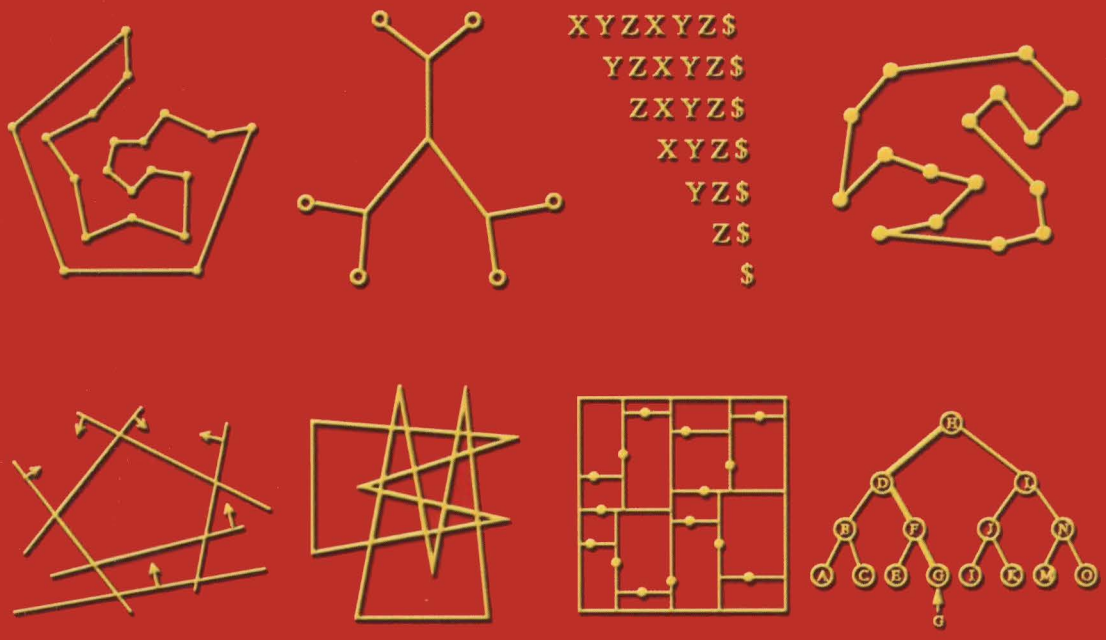
Второе издание

Алгоритмы

Руководство по разработке



Стивен Скиена

Springer

bhv®

The Algorithm Design Manual

Second Edition

Steven S. Skiena

The Algorithm Design Manual

Second Edition

Springer

Стивен С. Скиена

Алгоритмы
Руководство по разработке

2-е издание

Санкт-Петербург
«БХВ-Петербург»
2011

УДК 681.3.06
ББК 32.973.26-018.2
С42

Скиена С.

С42 Алгоритмы. Руководство по разработке. — 2-е изд.: Пер. с англ. — СПб.: БХВ-Петербург, 2011. — 720 с.: ил.

ISBN 978-5-9775-0560-4

Книга является наиболее полным руководством по разработке эффективных алгоритмов. Первая часть книги содержит практические рекомендации по разработке алгоритмов: приводятся основные понятия, дается анализ алгоритмов, рассматриваются типы структур данных, основные алгоритмы сортировки, операции обхода графов и алгоритмы для работы со взвешенными графами, примеры использования комбинаторного поиска, эвристических методов и динамического программирования. Вторая часть книги содержит обширный список литературы и каталог из 75 наиболее распространенных алгоритмических задач, для которых перечислены существующие программные реализации. Приведены многочисленные примеры задач.

Книгу можно использовать в качестве справочника по алгоритмам для программистов, исследователей и в качестве учебного пособия для студентов соответствующих специальностей.

Для программистов, исследователей и студентов

УДК 681.3.06
ББК 32.973.26-018.2

ISBN 978-1-84800-069-8 (англ.)
ISBN 978-5-9775-0560-4 (рус.)

Оглавление

Предисловие

Многие профессиональные программисты, с которыми я встречался, не очень хорошо подготовлены к решению проблем разработки алгоритмов. Это прискорбно, так как методика разработки алгоритмов является одной из важнейших *технологий* вычислительной техники. Создание правильных, эффективных и реализуемых алгоритмов для решения реальных задач требует от разработчика знаний в двух областях:

- *Методика.* Хорошие разработчики алгоритмов знают несколько фундаментальных приемов, в число которых входят работа со структурами данных, динамическое программирование, поиск в глубину, перебор с возвратами и эвристика. Пожалуй, самым важным техническим приемом является моделирование — искусство абстрагирования от запутанного реального приложения к четко сформулированной задаче, поддающейся алгоритмическому решению.
- *Ресурсы.* Хорошие разработчики алгоритмов пользуются коллективным опытом предыдущих поколений разработчиков. Вместо того, чтобы создавать "с нуля" алгоритм для стоящей перед ними задачи, они сначала узнают, что уже известно о ней и ищут существующие реализации решения, чтобы использовать их в качестве отправной точки. Они знают много классических алгоритмических задач, которые служат исходным материалом для моделирования практически любого приложения.

Эта книга была задумана как руководство по разработке алгоритмов, в котором я планировал изложить технологию разработки комбинаторных алгоритмов. Она рассчитана как на студентов, так и на профессионалов. Книга состоит из двух частей. Часть I является собой общее руководство по техническим приемам разработки и анализа компьютерных алгоритмов. Часть II предназначена для использования в качестве справочного и познавательного материала и состоит из каталога алгоритмических ресурсов, реализаций и обширного списка литературы.

Читателю

Меня очень обрадовал теплый прием, с которым было встречено первое издание книги "Руководство по разработке алгоритмов", опубликованное в 1997 г. Она была признана уникальным руководством по применению алгоритмических приемов для решения задач, которые часто возникают в реальной жизни. Но с тех пор многое изменилось в этом мире. Если считать, что начало современной разработке и анализу алгоритмов было положено приблизительно в 1970 г., то получается, что около 30% современной истории алгоритмов приходится на время, прошедшее после первой публикации руководства.

Читатели одобрили три аспекта руководства: каталог алгоритмических задач, истории из жизни и электронную версию книги. Эти элементы были сохранены в настоящем издании.

- *Каталог алгоритмических задач.* Не так-то просто узнать, что уже известно о стоящей перед вами задаче. Именно поэтому в книге имеется каталог 75 наиболее важных задач, часто возникающих в реальной жизни. Просматривая его, студент или специалист-практик может быстро выяснить название своей задачи и понять, что о ней известно и как приступить к ее решению. Чтобы облегчить идентификацию, каждая задача в каталоге сопровождается рисунками состояния "до и после", иллюстрирующими требуемые характеристики входа и выхода. За этот каталог задач один остроумный рецензент предложил назвать мою книгу "Автостопом по алгоритмам".

  Каталог задач является самой важной частью этой книги. Обновляя каталог для этого издания, я собрал отзывы и рекомендации ведущих мировых экспертов по каждой содержащейся в нем задаче. Особое внимание было уделено обновлению программных реализаций решений задач.

- *Истории из жизни.* На практике алгоритмические задачи редко возникают в начале работы над большим проектом. Наоборот, они обычно появляются в виде подзадач, когда становится ясно, что программист не знает, что делать дальше, или что принятое решение ошибочно.

  Чтобы продемонстрировать, как алгоритмические задачи возникают в реальной жизни, в материал книги были включены правдивые истории, описывающие наш опыт по решению практических задач. При этом преследовалась цель показать, что разработка и анализ алгоритмов представляют собой не отвлеченную теорию, а важный инструмент, требующий умелого обращения.

  В этом издании сохранены все первоначальные истории из жизни (обновленные по мере необходимости), а также были добавлены новые истории, имеющие отношение к внешней сортировке, работе с графами, методу имитации отжига и другим алгоритмическим областям.

- *Электронная версия.* Поскольку человек практичный обычно ищет готовую программу, а не алгоритм, в книге даются ссылки на все имеющиеся рабочие реализации алгоритмов. Для удобства эти реализации собраны на одном веб-сайте **http://www.cs.sunysb.edu/~algorith**. После публикации книги этот веб-сайт долгое время был одним из первых в результатах поиска в Google по слову "algorithm".

  Кроме этого, в книге даны рекомендации, как найти подходящий код для решения той или иной задачи. Наличие данных реализаций сводит проблему разработки алгоритма к правильному моделированию приложения и избавляет разработчика от необходимости разбираться в подробностях самого алгоритма. Этот подход применяется на всем протяжении книги.

Важно обозначить то, чего нет в этой книге. Мы не уделяем большого внимания математическому обоснованию алгоритмов и в большинстве случаев ограничиваемся неформальными рассуждениями. В этой книге вы не найдете ни одной теоремы. Более подробную информацию вы можете получить, изучив приведенные программы или справочный материал. Цель данного руководства в том, чтобы как можно быстрее указать читателю верное направление движения.

Преподавателю

Эта книга содержит достаточно материала для курса "Введение в алгоритмы". Предполагается, что читатель обладает знаниями, полученными при изучении таких курсов, как "Структуры данных" или "Теория вычислительных машин и систем".

На сайте **http://www.algorist.com** можно загрузить полный набор лекционных слайдов для преподавания материала этой книги. Кроме того, я выложил аудио- и видеолекции с использованием этих слайдов для преподавания полного курса продолжительностью в один семестр. Таким образом, вы можете через Интернет воспользоваться моей помощью в преподавании вашего курса.

Главное внимание в книге уделяется разработке алгоритмов, а их анализ стоит на втором плане. Книгу можно использовать как для обычных лекционных курсов, так и для активного обучения, при котором преподаватель не читает лекцию, а руководит решением реальных задач в группе студентов. Истории из жизни предоставляют введение в активный метод обучения.

Книга была полностью переработана с целью облегчить ее использование в качестве учебника. Для настоящего издания характерны:

- *подробное изложение материала.* По сравнению с предыдущим изданием, объем первой части книги был увеличен вдвое. Однако дополнительный материал не увеличивает количество обсуждаемых тем, а служит для более полного и тщательного изложения основ;
- *обсуждение основ.* Учебники по разработке алгоритмов обычно представляют общеизвестные алгоритмы как нечто само собой разумеющееся и не обсуждают идеи, лежащие в их основе, или слабые места других подходов. Истории из жизни, приводимые в этой книге, иллюстрируют процесс выбора алгоритма на примерах решения некоторых прикладных задач, но я применил аналогичный подход и к изложению материала, касающегося классических алгоритмов;
- *остановки для размышлений.* В этих разделах я изложил ход собственных рассуждений (включая тупиковые решения) в процессе выполнения конкретного домашнего задания. Разделы "Остановка для размышлений" разбросаны по всему тексту, чтобы повысить активность читателей по решению задач. Ответы к задачам даются тут же;
- *переработанные и новые домашние задания.* Настоящее издание книги содержит вдвое больше упражнений для домашней работы, чем предыдущее. Нечетко сформулированные задания были исправлены или заменены новыми. Каждой задаче присвоен уровень сложности от 1 до 10;
- *экзамены на основе материала книги.* Студентам моих курсов по изучению алгоритмов я обещаю, что все вопросы текущих контрольных работ и экзаменов в конце семестра будут взяты из домашних заданий в этой книге. Таким образом, студенты точно знают, что нужно изучать, чтобы успешно сдать экзамен. Для действенности этого подхода я тщательно подобрал количество, тип и сложность домашних заданий, следя за тем, чтобы количество задач было оптимальным;
- *разделы с подведением итогов.* В этих разделах делается акцент на основных понятиях, которые нужно усвоить в данной главе;

- *ссылки на задачи по программированию.* В конце упражнений для каждой главы даются ссылки на 3–5 задач по программированию, взятых с веб-сайта **http://www.programming-challenges.com**. Эти задания можно использовать, чтобы добавить практический компонент реализации алгоритмов к теоретическому курсу;
- *большой объем работающего программного кода вместо псевдокода.* В этой книге увеличено количество алгоритмов, написанных на реальных языках программирования (в частности, на языке C), за счет уменьшения объема псевдокода. Я считаю, что корректность и надежность проверенной работающей реализации дают ей преимущество над неформальным представлением простых алгоритмов. Полностью реализованные алгоритмы доступны на сайте **http://www.algorist.com**;
- *замечания к главам*. Каждая глава завершается кратким разделом с замечаниями, содержащими ссылки на основные источники информации и дополнительный справочный материал.

Благодарности

Обновление посвящения через десять лет после выхода первого издания книги заставляет задуматься о скоротечности времени. С тех пор Рени стала моей женой, а потом матерью наших двух детей, Бонни и Эбби. Мой отец ушел в мир иной, но моя мама и мои братья Лен и Роб продолжают играть важную роль в моей жизни. Я посвящаю эту книгу членам моей семьи, новым и старым, тем, кто с нами и тем, кто покинул нас.

Я бы хотел поблагодарить нескольких людей за их непосредственный вклад в новое издание. Эндрю Гон (Andrew Gaun) и Бетсон Томас (Betson Thomas) оказали мне помощь, в частности, при создании инфраструктуры нового сайта **http://www.cs.sunysb.edu/~algorith** и при решении некоторых вопросов по подготовке рукописи. Дэвид Грайз (David Gries) дал ценные рекомендации в объеме, превышающем мои ожидания. Химаншу Гупта (Himanshu Gupta) и Бин Танг (Bin Tang) отважно использовали рукописную версию этого издания в своих академических курсах. Я также выражаю благодарность редакторам издательства Springer-Verlag Уэйну Уиллеру (Wayne Wheeler) и Алану Уайлду (Allan Wylde).

Группа экспертов по разработке алгоритмов изучила материал книги, делясь со мной своими знаниями и извещая меня о новостях в этой области. Я благодарен всей группе, в состав которой входили:

Ами Амир (Ami Amir), Хёрв Бронниманн (Herve Bronnimann), Бернар Шазель (Bernard Chazelle), Крис Чу (Chris Chu), Скотт Коттон (Scott Cotton), Ефим Диниц (Yefim Dinitz), Комеи Фукуда (Komei Fukuda), Майкл Гудрич (Michael Goodrich), Ленни Хит (Lenny Heath), Сихат Имамоглу (Cihat Imamoglu), Тао Жянг (Tao Jiang), Дэвид Каргер (David Karger), Джузеппе Лиотта (Giuseppe Liotta), Альберт Мао (Albert Mao), Сильвано Мартелло (Silvano Martello), Кэтрин Мак-Геох (Catherine McGeoch), Курт Мельхорн (Kurt Mehlhorn), Скотт А. Митчелл (Scott A. Mitchell), Насер Мескини (Naceur Meskini), Джин Майерс (Gene Myers), Гонзало Наварро (Gonzalo Navarro), Стивен Норт (Stephen North), Джо О'Рурк (Joe O'Rourke), Майк Патерсон (Mike Paterson), Тео Павлидис (Theo Pavlidis), Сет Петти (Seth Pettie), Мишель Почиола (Michel Pocchiola), Барт Пренил (Bart Preneel), Томаш Радзик

(Tomasz Radzik), Эдвард Рейнголд (Edward Reingold), Фрэнк Раски (Frank Ruskey), Питер Сэндерс (Peter Sanders), Жоао Сетубал (Joao Setubal), Джонатан Шевчук (Jonathan Shewchuk), Роберт Скил (Robert Skeel), Дженз Стои (Jens Stoye), Торстен Суэл (Torsten Suel), Брюс Уотсон (Bruce Watson) и Ури Цвик (Uri Zwick).

Многие упражнения были созданы по подсказке коллег или под влиянием других работ. Восстановление первоначальных источников годы спустя является задачей не из легких, но на веб-сайте книги дается ссылка на первоисточник каждой задачи (насколько мне удавалось его вспомнить).

Было бы невежливо не поблагодарить людей, внесших важный вклад в первое издание книги. Рики Брэдли (Ricky Bradley) и Дарио Влах (Dario Vlah) создали мощную инфраструктуру для веб-сайта, логически стройную и легко расширяемую. Жонг Ли (Zhong Li) сделал большинство рисунков каталога задач с помощью графического редактора xfig. Ричард Крэндол (Richard Crandall), Рон Дэниэльсон (Ron Danielson), Такис Метаксас (Takis Metaxas), Дэйв Миллер (Dave Miller), Гири Нарасимхан (Giri Narasimhan) и Джо Закари (Joe Zachary) проверили черновые версии первого издания. Их содержательные отзывы и рекомендации помогли мне сформировать содержимое данного издания.

Большую часть моих знаний об алгоритмах я получил, изучая их совместно с моими аспирантами. Многие из них — Йо-Линг Лин (Yaw-Ling Lin), Сундарам Гопалакришнан (Sundaram Gopalakrishnan), Тинг Чен (Ting Chen), Фрэнсин Иванс (Francine Evans), Харальд Рау (Harald Rau), Рики Брэдли (Ricky Bradley) и Димитрис Маргаритис (Dimitris Margaritis) — являются персонажами историй, изложенных в книге. Мне всегда было приятно работать и общаться с моими друзьями и коллегами из Университета Стоуни Брук — Эсти Аркином (Estie Arkin), Майклом Бэндером (Michael Bender), Джи Гао (Jie Gao) и Джо Митчеллом (Joe Mitchell). И, наконец, хочу сказать спасибо Майклу Брокстайну (Michael Brochstein) и остальным жителям города за то, что познакомили меня с настоящей жизнью далеко за пределами Стоуни Брук.

Ограничение ответственности

Традиционно вину за любые недостатки в книге великодушно принимает на себя ее автор. Я же делать этого не намерен. Любые ошибки, недостатки или проблемы в этой книге являются виной кого-то другого; но я буду признателен, если вы поставите меня в известность о них, с тем, чтобы я знал, кто виноват.

Стивен С. Скиена
Кафедра вычислительной техники
Университет Стоуни Брук
Стоуни Брук, Нью-Йорк 11794-4400
http://www.cs.sunysb.edu/~skiena
Апрель 2008 г.

ЧАСТЬ I

Практическая разработка алгоритмов

ГЛАВА 1

Введение в разработку алгоритмов

Что такое алгоритм? Это процедура выполнения определенной задачи. Алгоритм является основополагающей идеей любой компьютерной программы.

Чтобы представлять интерес, алгоритм должен решать общую, корректно поставленную *задачу*. Определение задачи, решаемой с помощью алгоритма, дается описанием всего множества *экземпляров*, которые должен обработать алгоритм, и выхода, т. е. результата, получаемого после обработки одного из этих экземпляров. Описание одного из экземпляров задачи может заметно отличаться от формулировки общей задачи. Например, постановка *задача сортировки* делается таким образом:

ЗАДАЧА. Сортировка.

Вход. Последовательность из n элементов: a_1, ..., a_n.

Выход. Перестановка элементов входной последовательности таким образом, что для ее членов справедливо соотношение $a'_1 \leq a'_2 \leq ... \leq a'_{n-1} \leq a'_n$.

Экземпляром задачи сортировки может быть массив имен, например {Mike, Bob, Sally, Jill, Jan}, или набор чисел, например {154, 245, 568, 324, 654, 324}. Первым шагом к решению — определить общую задачу.

Алгоритм — это процедура, которая принимает любой из возможных входных экземпляров и преобразует его в соответствии с требованиями, указанными в условии задачи. Для решения задачи сортировки существует много разных алгоритмов. В качестве примера одного из таких алгоритмов можно привести метод *сортировки вставками*. Сортировка этим методом заключается во вставке в требуемом порядке элементов из неотсортированной части списка в отсортированную часть, первоначально содержащую один элемент. Реализация этого алгоритма на языке C представлена в листинге 1.1.

Листинг 1.1. Реализация алгоритма сортировки вставками

```
insertion_sort(item s[], int n)
{
   int i,j;                    /* Счетчики */
      for (i=1; i<n; i++) {
         j=i;
         while ((j>0) && (s[j] < s[j-1])) {
            swap(&s[j],&s[j-1]);
            j = j-1;
         }
      }
   }
```

А на рис. 1.1 показано применение этого алгоритма для сортировки определенного экземпляра — строки INSERTIONSORT.

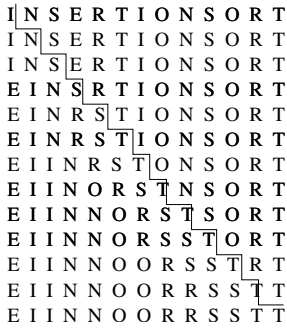


Рис. 1.1. Пример использования алгоритма сортировки вставками (шкала времени направлена вниз)

Обратите внимание на универсальность этого алгоритма. Его можно применять как для сортировки слов, так и для сортировки чисел, используя соответствующую операцию сравнения для определения, какое из двух значений поставить первым. Можно с легкостью убедиться, что этот алгоритм правильно сортирует любой возможный набор входных величин в соответствии с нашим определением задачи сортировки.

Хороший алгоритм должен обладать тремя свойствами: быть корректным, эффективным и легкореализуемым. Получение комбинации всех трех свойств сразу может оказаться недостижимой задачей. В производственной обстановке любая программа, которая предоставляет достаточно хорошие результаты и не замедляет работу системы, в большинстве случаев является приемлемой, независимо от того, возможны ли улучшения этих показателей. В этой области вопрос получения самых лучших возможных результатов или достижения максимальной эффективности обычно возникает только в случае серьезных проблем с производительностью или с законом.

В этой главе основное внимание уделяется вопросу корректности алгоритмов, а их эффективность рассматривается в *главе 2*. Способность определенного алгоритма правильно решить поставленную задачу, т. е. его корректность, редко поддается очевидной оценке. Алгоритмы обычно сопровождаются доказательством их правильности в виде объяснения, почему для каждого экземпляра задачи будет выдан требуемый результат. Но прежде чем продолжить обсуждение темы, мы продемонстрируем, что аргумент "это очевидно" никогда не является достаточным доказательством правильности алгоритма.

1.1. Оптимизация маршрута робота

Рассмотрим задачу, которая часто возникает на производстве и транспорте. Допустим, что нам нужно запрограммировать роботизированный манипулятор, который применяется для припаивания контактов интегральной схемы к контактным площадкам на печатной плате. Чтобы запрограммировать манипулятор для выполнения этой задачи, сначала нужно установить порядок, в котором манипулятор припаивает первый кон-

такт, потом второй, третий и т. д., пока не будут припаяны все. После обработки последнего контакта манипулятор возвращается к исходной позиции первого контакта для обработки следующей платы. Таким образом, маршрут манипулятора является замкнутым маршрутом, или циклом.

Так как роботы являются дорогостоящими устройствами, мы хотим минимизировать время, затрачиваемое манипулятором на обработку платы. Будет логичным предположить, что манипулятор перемещается с постоянной скоростью; соответственно, время перемещения от одной точки к другой прямо пропорционально расстоянию между точками. То есть, нам нужно найти алгоритмическое решение такой задачи:

ЗАДАЧА. Оптимизация маршрута робота.

Вход. Множество S из n точек, лежащих на плоскости.

Выход. Самый короткий маршрут посещения всех точек множества S.

Прежде чем приступать к программированию маршрута манипулятора, нам нужно разработать алгоритм решения этой задачи. На ум может прийти несколько подходящих алгоритмов. Но самым подходящим будет *эвристический алгоритм ближайшего соседа* (nearest-neighbor heuristic). Начиная с какой-либо точки p_0, мы идем к ближайшей к ней точке (соседу) p_1. От точки p_1 мы идем к ее ближайшему еще не посещенному соседу, таким образом исключая точку p_0 из числа кандидатов на посещение. Процесс повторяется до тех пор, пока не останется ни одной не посещенной точки, после чего мы возвращаемся в точку p_0, завершая маршрут. Псевдокод эвристического алгоритма ближайшего соседа представлен в листинге 1.2.

Листинг 1.2. Эвристический алгоритм ближайшего соседа

```
NearestNeighbor(P)
  Из множества P выбираем и посещаем произвольную начальную точку p0
  p = p0
  i = 0
  Пока остаются непосещенные точки
    i = i + 1
    Выбираем и посещаем непосещенную точку pi, ближайшую к точке pi-1
    Посещаем точку pi
  Возвращаемся в точку p0 от точки pn-1
```

Этот алгоритм можно рекомендовать к применению по многим причинам. Он прост в понимании и легко реализуется. Вполне логично сначала посетить близлежащие точки, чтобы сократить общее время прохождения маршрута. Алгоритм дает отличные результаты для входного экземпляра, показанного на рис. 1.2.

Алгоритм ближайшего соседа достаточно эффективен, т. к. в нем каждая пара точек (p_i, p_j) рассматривается, самое большее, два раза: первый раз при добавлении в маршрут точки p_i , а второй — при добавлении точки p_j. При всех этих достоинствах алгоритм имеет всего лишь один недостаток — он совершенно неправильный.

Неправильный? Да как он может быть неправильным? Поясню: несмотря на то, что алгоритм всегда создает маршрут, этот маршрут не обязательно будет самым коротким возможным маршрутом, или хотя бы приближающимся к таковому. Рассмотрим множество точек, расположенных в линию, как показано на рис. 1.3.

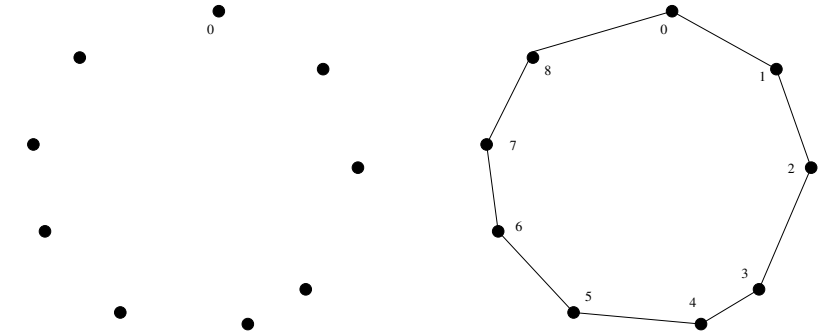


Рис. 1.2. Случай удачного входного экземпляра для эвристического алгоритма ближайшего соседа

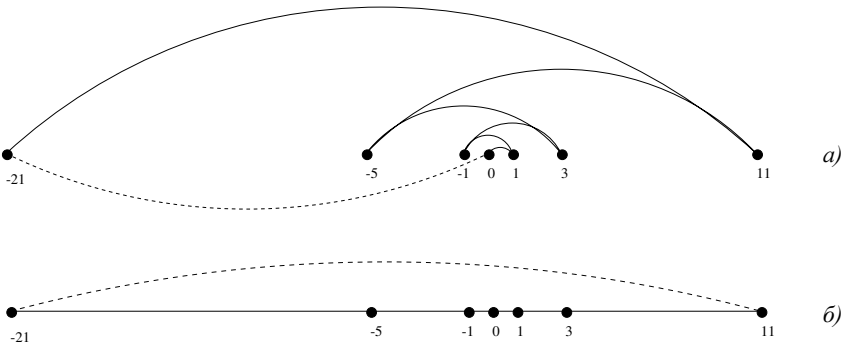


Рис. 1.3. Пример неудачного входного экземпляра для эвристического алгоритма ближайшего соседа *(а)* и оптимальное решение *(б)*

Цифры на рисунке обозначают расстояние от начальной точки до соответствующей точки справа или слева. Начав обход с точки 0 и посещая затем ближайшего непосещенного соседа текущей точки, мы будем метаться вправо-влево через нулевую точку, т. к. алгоритм не содержит указания, что нужно делать в случае одинакового расстояния между точками. Гораздо лучшее (более того, оптимальное) решение данного экземпляра задачи — начать обход с крайней левой точки и двигаться направо, посещая каждую точку, после чего возвратиться в исходное положение.

Представьте себе реакцию вашего начальника при виде манипулятора, мечущегося вправо-влево при выполнении такой простой задачи.

Можно сказать, что в данном случае проблема заключается в неудачном выборе отправной точки маршрута. Почему бы не начать маршрут с самой левой точки в качестве точки p_0? Это даст нам оптимальное решение данного экземпляра задачи.

Верно на все 100%, но лишь до тех пор, пока мы не развернем множество точек на 90 градусов, сделав все точки самыми левыми. А если к тому же немного сдвинуть первоначальную точку 0 влево, то она опять будет выбрана в качестве отправной. Теперь вместо дергания из стороны в сторону манипулятор будет скакать вверх-вниз, но время прохождения маршрута по-прежнему оставляет желать лучшего. Таким образом,

независимо от того, какую точку мы выберем в качестве исходной, алгоритм ближайшего соседа обречен на неудачу с некоторыми экземплярами задачи (т. е. с некоторыми множествами точек), и нам нужно искать другой подход. Условие, заставляющее всегда искать ближайшую точку, является излишне ограничивающим, т. к. оно принуждает нас выполнять нежелательные переходы. Задачу можно попробовать решить другим способом — соединяя пары самых близких точек, если такое соединение не вызывает никаких проблем, например, досрочного завершения цикла. Каждая вершина рассматривается как самостоятельная одновершинная цепочка. Соединив все вместе, мы получим одну цепочку, содержащую все точки. Соединив две конечные точки, мы получим цикл. На любом этапе выполнения этого эвристического алгоритма ближайших пар у нас имеется множество отдельных вершин и не имеющих общих вершин цепочек, которые можно соединить. Псевдокод соответствующего алгоритма показан в листинге 1.3.

Листинг 1.3. Эвристический алгоритм ближайших пар

```
ClosestPair(P)
  Пусть n — количество точек множества P.
  For i = 1 to n — 1 do
    d = ∞
    For каждой пары точек(s, t), не имеющих общих вершин цепей
      if dist(s, t) ≤ d then sₘ = s, tₘ = t и d = dist(s, t)
    Соединяем (sₘ,tₘ) ребром
  Соединяем две конечные точки ребром
```

Для экземпляра задачи, представленного на рис. 1.3, этот алгоритм работает должным образом. Сначала точка 0 соединяется со своими ближайшими соседями, точками 1 и –1. Потом соединение следующих ближайших пар точек выполняется поочередно влево-вправо, расширяя центральную часть по одному сегменту за проход. Эвристический алгоритм ближайших пар чуть более сложный и менее эффективный, чем предыдущий, но, по крайней мере, для данного экземпляра задачи он дает правильный результат.

Впрочем, это верно не для всех экземпляров. Посмотрите на результаты работы алгоритма на рис. 1.4, *а*. Данный входной экземпляр состоит из двух рядов равномерно расположенных точек. Расстояние между рядами $(1-e)$ несколько меньше, чем расстояние между смежными точками в рядах $(1+e)$. Таким образом, пары наиболее близких точек располагаются не по периметру, а напротив друг друга. Сперва противоположные точки соединяются попарно, а затем полученные пары соединяются поочередно по периметру. Общее расстояние маршрута алгоритма ближайших пар в этом случае будет равно $3(1-e)+2(1+e)+\sqrt{(1-e)^2+(2+2e)^2}$. Этот маршрут на 20% длиннее, чем маршрут на рис. 1.4, *б*, когда $e \approx 0$. Более того, есть входные экземпляры, дающие значительно худшие результаты, чем этот.

Таким образом, этот алгоритм тоже не годится. Какой из этих двух алгоритмов более эффективный? На этот вопрос нельзя ответить, просто посмотрев на них. Но очевидно, что оба алгоритма могут выдать очень плохие маршруты на некоторых с виду простых входных экземплярах.

Но каков же правильный алгоритм решения этой задачи? Можно попробовать перечислить все возможные перестановки множества точек, а потом выбрать перестановку,

сводящую к минимуму длину маршрута. Псевдокод этого алгоритма показан в листинге 1.4.

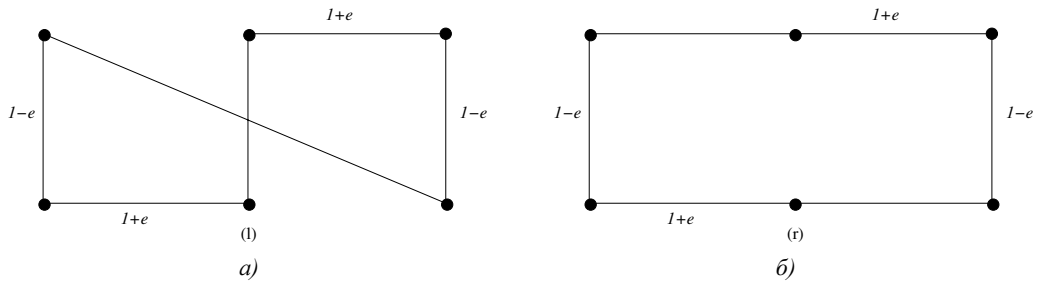


Рис. 1.4. Неудачный входной экземпляр для алгоритма ближайших пар *(а)* и оптимальное решение *(б)*

Листинг 1.4. Оптимальный алгоритм поиска маршрута

```
OptimalTSP(P)
  d = ∞
  For каждой перестановки Pi из общего числа перестановок n! множества
                                                              точек P
    If (cost(Pi) ≤ d) then d = cost(Pi) и Pmin = Pi
  Return Pmin
```

Так как рассматриваются все возможные упорядочения, то получение самого короткого маршрута гарантировано. Поскольку мы выбираем самую лучшую комбинацию, алгоритм правильный. В то же самое время он чрезвычайно медленный. Например, самый быстрый компьютер в мире не сможет в течение дня перечислить все 20! = 2 432 902 008 176 640 000 возможных перестановок 20 точек. А о реальных ситуациях, когда количество точек печатной платы достигает тысячи, можно и не говорить. Все компьютеры в мире, работая круглосуточно, не смогут даже приблизиться к решению этой задачи до конца существования Вселенной, а тогда решение этой задачи, скорее всего, уже не будет актуальным.

Поиском эффективного алгоритма решения этой задачи, называющейся *задачей коммивояжера* (traveling salesman problem, TSP), мы будем заниматься на протяжении большей части этой книги. Если же вам не терпится узнать решение уже сейчас, то вы можете посмотреть его в *разделе 16.4*.

Подведение итогов

Алгоритмы, которые всегда выдают правильное решение, коренным образом отличаются от *эвристических алгоритмов*, которые обычно выдают достаточно хорошие, но не гарантированные результаты.

1.2. Задача календарного планирования

Теперь рассмотрим задачу календарного планирования. Представьте, что вы кинозвезда и вам наперебой предлагают роли в разных кинофильмах, общее количество кото-

рых равно *n*. Каждое предложение имеет условие, что вы должны посвятить себя ему с первого до последнего дня съемок. Поэтому вы не можете сниматься одновременно в фильмах с полностью или частично накладывающимися периодами съемок.

Ваш критерий для принятия того или иного предложения довольно прост: вы хотите заработать как можно больше денег. Поскольку вам платят одинаково за каждый фильм, то вы стремитесь получить роли в как можно большем количестве фильмов, периоды съемок которых не конфликтуют.

На рис. 1.5 перечислены фильмы, в которых вам предлагают роли.

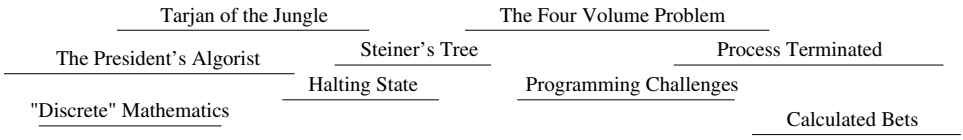


Рис. 1.5. Экземпляр задачи планирования непересекающихся календарных периодов

В данном случае очевидно, что вы можете сниматься, самое большее, в четырех фильмах — "Discrete Mathematics", "Programming Challenges", "Calculated Bets", а потом в "Halting State" или в "Steiner's Tree".

А в менее очевидных случаях вам (или вашему менеджеру) нужно будет решить следующую алгоритмическую задачу календарного планирования:

ЗАДАЧА. Планирование съемок в фильмах.

Вход. Множество *I* интервалов времени *n* в линейном порядке.

Выход. Самое большое подмножество непересекающихся интервалов времени, которое возможно во множестве *I*.

На ум может прийти несколько способов решения этой задачи. Один из них основан на представлении, что надо работать всегда, когда это возможно. Это означает, что вам нужно брать роль в фильме, съемки которого начинаются раньше всех других. Псевдокод этого алгоритма представлен в листинге 1.5.

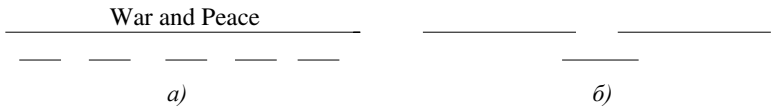
Листинг 1.5. Алгоритм первой возможной работы

```
EarliestJobFirst(I)
   Из множества фильмов I берем роль в фильме j с самым ранним началом
   съемок, период которых не пересекается с периодом ваших предыдущих
   обязательств. Поступаем таким образом до тех пор, пока больше
   не останется таких фильмов.
```

Этот подход выглядит логично, по крайней мере, до тех пор, пока вы не осознаете, что хватаясь за самую раннюю работу, можете пропустить несколько других, если первый фильм является сериалом. Пример такой ситуации показан на рис. 1.6, *а*, где самым ранним фильмом является киноэпопея "War and Peace", которая закрывает перед вами все другие перспективы.

Этот пример заставляет искать другое решение. Проблема с фильмом "War and Peace" заключается в том, что его съемки длятся слишком долго. В таком случае, может быть,

вам следует брать роли только в фильмах с самыми короткими периодами съемок? Разве не очевидно, что чем быстрее вы закончите сниматься в одном фильме, тем раньше можно начать сниматься в другом, максимизируя таким образом количество фильмов в любой выбранный период времени? Псевдокод этого алгоритма представлен в листинге 1.6.



а) *б)*

Рис. 1.6. Неудачные экземпляры задач для применения эвристики: самых ранних периодов *(а)*, самых коротких периодов *(б)*

Листинг 1.6. Алгоритм самого короткого периода

```
ShortestJobFirst(I)
  While (I ≠ Ø) do
    Из всего множества фильмов I берем фильм j с
    самым коротким периодом съемок
    Удаляем фильм j и любой другой фильм, период съемок которого
    пересекается с фильмом j, из множества доступных фильмов I
```

Но и этот подход окажется действенным только до тех пор, пока вы не увидите, что можете упустить возможность заработать больше (рис. 1.6, *б*). Хотя в данном случае потери меньше, чем в предыдущем случае, тем не менее вы получите только половину возможных заработков.

На данном этапе может показаться заслуживающим внимания алгоритм, который перебирает все возможные комбинации. Если отвлечься от проверки подмножеств интервалов (т. е. периодов съемок) на пересечение, этот алгоритм можно выразить псевдокодом, представленным в листинге 1.7.

Листинг 1.7. Алгоритм полного перебора

```
ExhaustiveScheduling(I)
  j = 0
  Smax = Ø
  For каждого из 2^n подмножеств Si множества интервалов I
    If (Si непересекающееся) и (size(Si) > j)
       then j = size(Si) и Smax = Si
  Return Smax
```

Но насколько эффективным будет такой алгоритм? Здесь ключевым ограничением является необходимость выполнить перечисление 2^n подмножеств n элементов. А положительным обстоятельством является то, что это *намного* лучше, чем перечисление всех $n!$ порядков n элементов, как предлагается в задаче оптимизации маршрута роботизированного манипулятора. В данном случае при $n = 20$ имеется около миллиона подмножеств, которые можно за несколько секунд перебрать на современном компью-

тере. Но когда выбор фильмов возрастет до $n = 100$, то 2^{100} будет намного больше, чем значение 20!, которое положило нашего робота на лопатки в предыдущей задаче.

Разница между задачей составления маршрута и задачей календарного планирования заключается в том, что для последней имеется алгоритм, который решает задачу и правильно, и эффективно. Для этого вам нужно брать роли только в фильмах с самым ранним окончанием съемок, т. е. выбрать такой временной интервал *x*, у которого правая конечная точка находится левее правых конечных точек всех прочих временных интервалов. Таким фильмом в нашем примере (см. рис. 1.5) является "Discrete Mathematics". Вполне возможно, что съемки других фильмов начались раньше, чем съемки фильма *x*, но все они должны пересекаться друг с другом (по крайней мере, частично), поэтому мы можем выбрать, самое большее, один фильм изо всей группы. Съемки фильма *x* закончатся раньше всего, поэтому остальные фильмы с накладывающимися съемочными периодами потенциально блокируют другие возможности, расположенные справа от них. Очевидно, что выбрав фильм *x*, вы никак не можете проиграть. Псевдокод эффективного алгоритма правильного решения задачи календарного планирования будет выглядеть так, как показано в листинге 1.8.

Листинг 1.8. Оптимальный алгоритм календарного планирования

```
OptimalScheduling(I)
  While (I ≠ Ø) do
    Из всего множества фильмов I выбираем фильм j с самым
          ранним окончанием съемок
    Удаляем фильм j и любой другой фильм, съемки которого
          пересекаются с фильмом j, из множества доступных фильмов I
```

Обеспечение оптимального решения для всего диапазона возможных входных экземпляров является трудной, но, как правило, выполнимой задачей. Важной частью процесса разработки такого алгоритма является поиск входных экземпляров задачи, которые опровергают наше допущение о правильности алгоритма-претендента на решение. Эффективные алгоритмы часто скрываются где-то совсем рядом, и в этой книге мы хотим помочь вам развить навыки их обнаружения.

Подведение итогов

Кажущиеся вполне логичными алгоритмы очень легко могут оказаться неправильными. Правильность алгоритма требует тщательного доказательства.

1.3. Обоснование правильности алгоритмов

Будем надеяться, что предшествующие примеры продемонстрировали вам всю сложность темы правильности алгоритмов. Правильные алгоритмы выделяются из общего числа с помощью специальных инструментов, главный из которых называется *доказательством*.

Адекватное математическое доказательство состоит из нескольких частей. Прежде всего, требуется ясная и четкая формулировка того, что вы пытаетесь доказать. Потом необходим набор предположений, которые всегда считаются верными и поэтому исполь-

зуются как часть доказательства. Далее, цепь умозаключений приводит нас от начальных предположений к конечному утверждению, которое мы пытаемся доказать. Наконец, небольшой черный квадрат ■ в тексте указывает на конец доказательства.

В этой книге формальным доказательствам не уделяется большого внимания, т. к. правильное формальное доказательство привести очень трудно, а неправильное может вас сильно дезориентировать. На самом деле, доказательство является *демонстрацией*. Доказательства полезны только тогда, когда они простые и незамысловатые — ясные и лаконичные аргументы, объясняющие, почему алгоритм удовлетворяет требованию нетривиальной правильности.

Правильные алгоритмы требуют тщательного изложения и определенных усилий для доказательства как их правильности, так и того факта, что они *не* являются неправильными. В последующих разделах мы разработаем инструменты для достижения этих целей.

1.3.1. Представление алгоритмов

Цепь логических умозаключений об алгоритме невозможно построить без тщательного описания последовательности шагов, которые необходимо выполнить. Для этой цели наиболее часто употребляются, по отдельности или в совокупности, три формы представления алгоритма: обычный язык, псевдокод и язык программирования. Самым загадочным из этих средств представления алгоритма является псевдокод; это средство лучше всего можно определить как язык программирования, который никогда не выдает сообщений о синтаксических ошибках. Все три способа являются полезными, т. к. существует естественное стремление к компромиссу между легкостью восприятия и точностью представления алгоритма. Наиболее простым для понимания "языком программирования" является обычный язык, но в то же время он наименее точен. С другой стороны, такие языки, как Java или C/C++, позволяют точно выразить алгоритм, но создавать и понимать алгоритмы на этих языках задача не из легких. В отношении сложности применения и понимания псевдокод представляет золотую середину между этими двумя крайностями.

Выбор самого лучшего способа представления алгоритма зависит от ваших предпочтений. Я, например, сначала описываю свои алгоритмические идеи на обычном языке, а затем перехожу на более формальный псевдокод наподобие языка программирования или даже на настоящий язык программирования для уточнения сложных деталей.

Не допускайте ошибку, которую часто делают мои студенты, — используют псевдокод, чтобы приукрасить плохо определенную идею и придать ей более формальный и солидный вид. При описании алгоритма следует стремиться к ясности. Например, алгоритм ExhaustiveScheduling (см. листинг 1.7) можно было бы выразить на обычном языке так, как показано в листинге 1.9.

Листинг 1.9. Алгоритм полного перебора

```
ExhaustiveScheduling(I)
  Протестировать все 2^n подмножеств множества I и возвратить самое
    большое подмножество непересекающихся интервалов.
```

Подведение итогов

В основе любого алгоритма лежит *идея*. Если в описании алгоритма не просматривается ясно ваша идея, значит, вы используете для ее выражения нотацию слишком низкого уровня.

1.3.2. Задачи и свойства

Чтобы продемонстрировать правильность алгоритма, одного его описания недостаточно. Нам также нужно подробное описание задачи, которая подлежит решению.

Постановка задачи состоит из двух частей: набора допустимых входных экземпляров и требований к выходу алгоритма. Невозможно доказать правильность алгоритма для нечетко поставленной задачи. Иными словами, поставьте неправильно задачу, и вы получите неправильный ответ.

Постановки некоторых задач допускают слишком широкий диапазон входных экземпляров. Допустим, что в нашей задаче календарного планирования съемки фильмов могут прерываться на некоторое время. Например, съемки фильма могут быть запланированы на сентябрь и ноябрь, а октябрь свободен. Тогда календарный план съемок для любого фильма будет состоять из *набора* временных интервалов. В этом случае мы можем браться за роли в двух фильмах с чередующимися, но не пересекающимися периодами съемок. Такую общую задачу календарного планирования нельзя решить с помощью алгоритма самого раннего окончания съемок. Более того, для решения этой общей задачи вообще *не существует* эффективного алгоритма.

Подведение итогов

Важным и заслуживающим внимания приемом разработки алгоритмов является сужение множества допустимых экземпляров задачи до тех пор, пока не будет найден правильный и эффективный алгоритм. Например, задачу общих графов можно свести к задаче деревьев, или двумерную геометрическую задачу свести к одномерной.

При указании требований выхода алгоритма часто допускаются две ошибки. Первая из них — плохая формулировка вопроса. Примером может служить вопрос о *наилучшем* маршруте между двумя точками на карте при отсутствии определения, что значит "наилучший". Лучший в каком смысле? В смысле самого короткого расстояния, самого короткого времени прохождения или, может быть, минимального количества поворотов?

Второй ошибкой является формулирование составных целей. Каждый из трех только что упомянутых критериев наилучшего маршрута является четко определенной целью правильного эффективного алгоритма оптимизации. Но в качестве требования к решению из этих критериев можно выбрать только один. Например, формулировка: "Найти самый короткий маршрут от точки к точке, содержащий количество поворотов не более чем в два раза превышающее необходимое" является вполне четкой постановкой задачи. Но решить такую задачу очень трудно, т. к. решение требует сложного анализа.

Для примеров постановки задач читателю настоятельно рекомендуется ознакомиться с постановкой каждой из 75 задач во второй части этой книги. Правильная постановка задачи является важной частью ее решения. Изучение постановок всех этих классических задач поможет вам распознать задачи, постановкой и решением которых кто-то уже занимался до вас.

1.3.3. Демонстрация неправильности алгоритма

Самый лучший способ доказать *неправильность* алгоритма — найти экземпляр задачи, для которого алгоритм выдает неправильный ответ. Такие экземпляры задачи называются *контрпримерами*. Никто не бросится на защиту алгоритма, для которого был предоставлен контрпример. Вполне разумно выглядящие алгоритмы можно моментально опровергнуть посредством очень простых контрпримеров. Хороший контрпример должен обладать двумя важными свойствами:

- *проверяемостью.* Чтобы продемонстрировать, что некий входной экземпляр задачи является контрпримером для определенного алгоритма, требуется вычислить ответ, который алгоритм выдаст для данного экземпляра, и предоставить лучший ответ, с тем, чтобы доказать, что алгоритм не смог его найти;
- *простотой.* Хороший контрпример не содержит ничего лишнего и ясно демонстрирует, *почему* именно предлагаемый алгоритм неправильный. Поэтому обнаруженный контрпример следует упростить. Контрпример на рис. 1.6, *а* можно упростить и улучшить, сократив количество пересекающихся периодов с четырех до двух.

Развитие навыков поиска контрпримеров будет стоить затраченного времени. Этот процесс в чем-то подобен разработке наборов тестов для проверки компьютерных программ, но в нем главную роль играет удачная догадка, а не перебор вариантов. Приведу несколько советов.

- *Ищите мелкомасштабные решения.* Обратите внимание, что мои контрпримеры для задач поиска маршрута содержат шесть или меньше точек, а для задач календарного планирования — только три временных интервала. Это обстоятельство указывает на то, что если алгоритм неправильный, то доказать это можно на очень простом экземпляре. Алгористы-любители нередко создают большой запутанный экземпляр, с которым потом не могут справиться. А профессионалы внимательно изучат несколько небольших экземпляров, т. к. их легче осмыслить и проверить.
- *Рассмотрите все решения.* Для наименьшего нетривиального значения n имеется только небольшое количество экземпляров. Например, существуют только три представляющих интерес способа расположения двух интервалов на прямой: неперекрывающиеся интервалы, частично перекрывающиеся интервалы и полностью перекрывающиеся интервалы. Все случаи размещения на прямой трех интервалов (включая контрпримеры для обоих эвристических алгоритмов календарного планирования) можно создать, по-разному добавляя третий интервал к этим двум, расположенным указанными тремя способами.
- *Ищите слабое звено.* Если рассматриваемый вами алгоритм работает по принципу "всегда берем самое большее" (так называемый *жадный алгоритм*), то подумайте, почему этот подход может быть неправильным.
- *Ищите ограничения.* "Жадный" эвристический алгоритм можно сбить с толку необычным методом предоставления входного экземпляра, содержащего одинаковые элементы. В этой ситуации алгоритму будет не на чем основывать свое решение и он, возможно, возвратит неоптимальное решение.
- *Рассматривайте крайние случаи.* Многие контрпримеры представляют собой комбинацию большого и малого, правого и левого, многих и немногих, далекого и

близкого. Обычно легче осмыслить и проверить примеры по краям диапазона, чем из его середины. Рассмотрим в качестве примера два скопления точек, расстояние d между которыми намного превышает расстояние между точками в любом из них. Длина оптимального маршрута коммивояжера в этой ситуации будет практически равна $2d$, независимо от количества посещаемых точек, т. к. длина маршрута внутри каждого скопления точек несущественна.

Подведение итогов

Лучший способ опровергнуть правильность эвристического алгоритма — испытать его на контрпримерах.

1.3.4. Индукция и рекурсия

Один факт, что для данного алгоритма не был найден контрпример, вовсе не означает, что алгоритм правильный. Для этого требуется доказательство или демонстрация правильности. Для доказательства правильности алгоритма часто применяется математическая индукция.

Мои первые впечатления о математической индукции были таковы, словно это какое-то шаманство. Вы берете формулу типа $\sum_{i=1}^{n} i = n(n+1)/2$, доказываете ее для базового случая, например, 1 или 2, потом допускаете, что утверждение справедливо для $n-1$, и на основе этого допущения доказываете формулу для общего n. Это называется доказательством? Полнейший абсурд!

Мои первые впечатления о рекурсии в программировании были точно такими же — чистое шаманство. Программа проверяет входной аргумент на равенство базовому значению, например, 1 или 2. При отрицательном результате такой проверки более сложный случай решается путем разбивки его на части и вызова *этой же программы в качестве подпрограммы* для решения этих частей. Это называется программой? Полнейший абсурд!

Причиной, благодаря которой как рекурсия, так и математическая индукция кажутся шаманством, является тот факт, что рекурсия и *есть* математическая индукция. В обеих имеются общие и граничные условия, при этом общее условие разбивает задачу на все более мелкие части, а граничное, или начальное, условие завершает рекурсию. Если вы понимаете одно из двух, — или рекурсию, или индукцию, — вы в состоянии понять, как работает другое.

Мне приходилось слышать, что программист — это математик, который умеет строить доказательства только методом индукции. Частично дело в том, что программисты — никудышные построители доказательств, но главным образом в том, что многие изучаемые ими алгоритмы являются или рекурсивными, или инкрементными (поэтапными).

Рассмотрим, например, правильность алгоритма сортировки вставками, представленного в начале этой главы. Его правильность можно *обосновать* методом индукции:

- базовый экземпляр содержит всего лишь один элемент, а по определению одноэлементный массив является полностью отсортированным;

- мы можем допустить, что после первых $n-1$ итераций сортировки вставками первые $n-1$ элементов массива A будут полностью отсортированы;
- чтобы определить, куда следует вставить последний элемент x, нам нужно найти уникальную ячейку между наибольшим элементом, не превышающим x, и наименьшим элементом, большим чем x. Для этого мы сдвигаем все бóльшие элементы назад на одну позицию, создавая место для элемента x в требуемой позиции. ■

Но к индуктивным доказательствам нужно относиться с большой осторожностью, т. к. в цепь рассуждений могут вкрасться трудно распознаваемые ошибки. Прежде всего, это *граничные ошибки*. Например, в приведенном выше доказательстве правильности алгоритма сортировки вставками мы самоуверенно заявили, что между двумя элементами массива A имеется однозначно определяемая ячейка, в которую можно вставить наш элемент x, когда массив в нашем базовом экземпляре содержит только одну ячейку. Поэтому для правильной обработки частных случаев вставки минимальных или максимальных элементов необходимо соблюдать бóльшую осторожность.

Другой, более распространенный, тип ошибок в индуктивных доказательствах связан с небрежным подходом к расширению экземпляра задачи. Добавление всего лишь одного элемента к экземпляру задачи может полностью изменить оптимальное решение. Соответствующий пример для задачи календарного планирования показан на рис. 1.7.

Рис. 1.7. Оптимальное решение (прямоугольники) до и после внесения изменений (пунктирная линия) в экземпляр задачи

После добавления нового временного интервала в экземпляр задачи новое оптимальное расписание может не содержать ни одного из временных интервалов любого оптимального решения, предшествующего изменению. Бесцеремонное игнорирование таких аспектов может вылиться в очень убедительное доказательство полностью неправильного алгоритма.

Подведение итогов

Математическая индукция обычно является верным способом проверки правильности рекурсивного алгоритма.

Остановка для размышлений: Правильность инкрементных алгоритмов

ЗАДАЧА. Доказать правильность рекурсивного алгоритма для инкрементации натуральных чисел, т. е. $y \rightarrow y+1$, представленного в листинге 1.10.

Листинг 1.10. Алгоритм для инкрементации натуральных чисел

```
Increment (y)
  if y = 0 then return(1) else
    if (y mod 2) = 1 then
      return(2·Increment(⌊y/2⌋))
    else return(y + 1)
```

Решение. Лично мне правильность этого алгоритма определенно *не* очевидна. Но т. к. это рекурсивный алгоритм, а я — программист, мое естественное побуждение будет доказать его методом индукции.

Абсолютно очевидно, что алгоритм правильно обрабатывает базовый случай, когда $y = 0$. Совершенно ясно, что $0 + 1 = 1$ и, соответственно, возвращается значение 1.

Теперь допустим, что функция работает правильно для общего случая, когда $y = n - 1$. На основе этого допущения нам нужно продемонстрировать правильность алгоритма для случая, когда $y = n$. Для половины случаев алгоритм доказывается легко, в частности для четных чисел (для которых `(y mod 2) = 0`), т. к. $y + 1$ возвращается явно.

Но для нечетных чисел решение зависит от результата, возвращаемого выражением `Increment([y/2])`. Здесь нам хочется воспользоваться нашим индуктивным допущением, но оно не совсем правильно. Мы сделали допущение, что функция `Increment` работает правильно, когда $y = n - 1$, но для значения y, равного приблизительно половине этого значения, мы этого не допускали. Теперь мы можем усилить наше допущение, объявив, что общий случай выдерживается для всех $y \leq n - 1$. Это усиление никоим образом не затрагивает сам принцип, но необходимо, чтобы установить правильность алгоритма.

Теперь случаи нечетных y (т. е. $y = 2m + 1$ для целого числа m) можно обработать, как показано в листинге 1.11.

Листинг 1.11. Алгоритм для инкрементации нечетных натуральных чисел

```
2•Increment([(2m + 1)/2]) = 2•Increment(⌊m + 1/2⌋)
                          = 2•Increment(m)
                          = 2(m + 1)
                          = 2m + 2 = y + 1
```

решая, таким образом, общий случай. ■

1.3.5. Суммирование

При анализе алгоритмов часто приходится прибегать к математическим формулам суммирования. А процесс доказательства правильности формулы суммирования представляет собой классический пример применения математической индукции. В конце этой главы дается несколько упражнений, в которых требуется доказать формулу с помощью индукции. Чтобы сделать эти упражнения более понятными, напомню основные принципы суммирования.

Формулы суммирования представляют собой краткие выражения, описывающие сложение сколь угодно большой последовательности чисел. В качестве примера можно привести такую формулу:

$$\sum_{i=1}^{n} f(i) = f(1) + f(2) + \ldots + f(n)$$

Суммирование многих алгебраических функций можно выразить простыми формулами в замкнутой форме. Например, поскольку сумма n единиц равна n, то:

$$\sum_{i=1}^{n} 1 = n$$

Сумму первых n целых чисел можно выразить через целые числа i и $(n - i + 1)$ следующим образом:

$$\sum_{i=1}^{n} i = \sum_{i=1}^{n/2} (i + (n - i + 1)) = n(n+1)/2$$

Очень пригодится в области анализа алгоритмов умение распознавать два основных класса формул суммирования:

- *Арифметические прогрессии.* Арифметическую прогрессию в виде формулы $S(n) = \sum_i^n i = n(n+1)/2$ можно встретить в анализе алгоритма сортировки методом выбора. По большому счету, важным фактом является наличие квадратичной суммы, а не то, что константа равняется 1/2. В общем, для $p \geq 1$:

  $$S(n, p) = \sum_{i}^{n} i^p = \Theta(n^{p+1})$$

  Таким образом, сумма квадратов кубичная, а сумма кубов — "четверичная" (если вы не против употребления такого слова). Нотация $\Theta(x)$ (тета большое) подробно рассматривается в *разделе 2.2*.

  Для $p < -1$ эта сумма всегда стремится к константе, даже когда $n \to \infty$.

- *Геометрические прогрессии.* В геометрических прогрессиях индекс цикла играет роль показателя степени, т. е.:

  $$G(n, a) = \sum_{i=0}^{n} a^i = a(a^{n+1} - 1)/(a - 1)$$

  Сумма прогрессии зависит от ее знаменателя, т. е. числа a. При $a < 1$ эта сумма стремится к константе, даже когда $n \to \infty$.

  Данная сходимость последовательности оказывается большим подспорьем в анализе алгоритмов. Это означает, что если количество элементов растет линейно, то их сумма необязательно будет расти линейно, а может быть ограничена константой. Например, $1 + 1/2 + 1/4 + 1/8 + \ldots \leq 2$ независимо от количества элементов последовательности.

  При $a > 1$ сумма стремительно возрастает при добавлении каждого нового элемента, например, $1 + 2 + 4 + 8 + 16 + 32 = 63$.

  В самом деле, для $a > 1$ $G(n, a) = \Theta(a^{a+1})$.

Остановка для размышлений. Формулы факториала

ЗАДАЧА. Докажите методом индукции, что $\sum_{i=1}^{n} i \times i! = (n+1)! - 1$.

Решение. Индукционная парадигма прямолинейна: сначала подтверждаем базовый случай (здесь мы принимаем $n = 1$, хотя случай $n = 0$ был бы еще более общим):

$$\sum_{i=1}^{1} i \times i! = 1 = (1+1)! - 1 = 2 - 1 = 1$$

Теперь допускаем, что данное утверждение верно для всех чисел вплоть до n. Для доказательства общего случая $n + 1$ видим, что если мы вынесем наибольший член из-под знака суммы

$$\sum_{i=1}^{n+1} i \times i! = (n+1) \times (n+1)! + \sum_{i=1}^{n} i \times i!$$

то получим левую часть нашего индуктивного допущения. Заменяя правую часть, получаем:

$$\begin{aligned}\sum_{i=1}^{n+1} i \times i! &= (n+1) \times (n+1)! + (n+1)! - 1 \\ &= (n+1)! \times ((n+1)+1) - 1 \\ &= (n+2)! - 1\end{aligned}$$

Этот общий прием выделения наибольшего члена суммы для выявления экземпляра индуктивного допущения лежит в основе всех таких доказательств. ■

1.4. Моделирование задачи

Моделирование является искусством формулирования приложения в терминах точно поставленных, хорошо понимаемых задач. Правильное моделирование является ключевым аспектом применения методов разработки алгоритмов решения реальных задач. Более того, правильное моделирование может сделать ненужным разработку или даже реализацию алгоритмов, соотнося ваше приложение с ранее решенной задачей. Правильное моделирование является ключом к эффективному использованию материала во второй части этой книги.

В реальных приложениях применяются реальные объекты. Вам, может быть, придется работать над системой маршрутизации сетевого трафика, планированием использования классных комнат учебного заведения или поиском закономерностей в корпоративной базе данных. Большинство же алгоритмов спроектировано для работы со строго определенными абстрактными структурами, такими как перестановки, графы или множества. Чтобы извлечь пользу из литературы по алгоритмам, вам нужно научиться выполнять постановку задачи абстрактным образом, в терминах процедур над фундаментальными структурами.

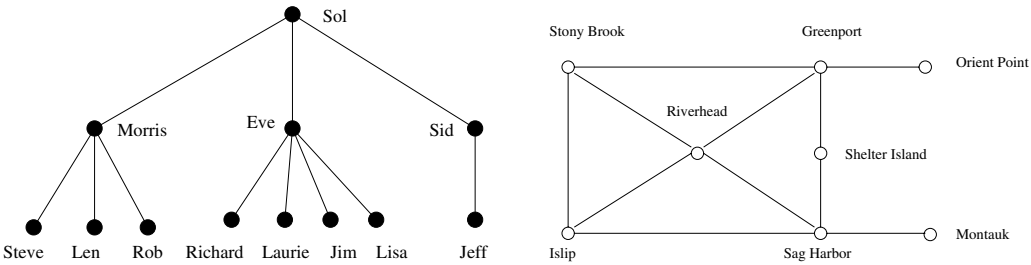
1.4.1. Комбинаторные объекты

Очень велика вероятность, что над решаемой вами алгоритмической задачей уже работали другие, хотя, возможно, и в совсем иных контекстах. Но не надейтесь узнать, что известно о вашей конкретной "задаче оптимизации процесса A", поискав в книге словосочетание "процесс A". Вам нужно сформулировать задачу оптимизации вашего процесса в терминах вычислительных свойств стандартных структур, таких как:

- *перестановка* — упорядоченное множество элементов. Например, {1, 4, 3, 2} и {4, 3, 2, 1} являются двумя разными перестановками одного множества целых чисел. Мы уже видели перестановки в задачах оптимизации маршрута манипулятора и

календарного планирования. Перестановки будут вероятным исследуемым объектом в задаче поиска "размещения", "маршрута", "границ" или "последовательности";

- *подмножество* — выборка из множества элементов. Например, множества {1, 3, 4} и {2} являются двумя разными подмножествами множества первых четырех целых чисел. В отличие от перестановок, порядок элементов подмножества не имеет значения, поэтому подмножества {1, 3, 4} и {4, 3, 1} являются одинаковыми. В проблеме календарного планирования нам пришлось иметь дело с подмножествами. Подмножества будут вероятным исследуемым объектом в задаче поиска "кластера", "коллекции", "комитета", "группы", "пакета" или "выборки";



а) *б)*

Рис. 1.8. Моделирование реальных структур с помощью деревьев и графов

- *дерево* — иерархическое представление взаимосвязей между объектами. Реальное применение деревьев показано на примере родословного древа семейства Скиена на рис. 1.8, *а*. Деревья будут вероятным исследуемым объектом в задаче поиска "иерархии", "отношений доминирования", "отношений предок/потомок" или "таксономии";
- *граф* — представление взаимоотношений между произвольными парами объектов. На рис. 1.8, *б* показана модель дорожной сети в виде графа, где вершины представляют населенные пункты, а ребра — дороги, соединяющие населенные пункты. Граф будет вероятным исследуемым объектом в задаче поиска "сети", "схемы", "инфраструктуры" или "взаимоотношений";
- *точка* — представление места в некотором геометрическом пространстве. Например, расположение автобусных остановок можно описать точками на карте (плоскости). Точка будет вероятным исследуемым объектом в задаче поиска "местонахождения", "позиции", "записи данных" или "расположения";
- *многоугольник* — представление области геометрического пространства. Например, с помощью полигона можно описать границы страны на карте. Многоугольники и многогранники будут вероятными исследуемыми объектами в задаче поиска "формы", "региона", "очертания" или "границы";
- *строка* — последовательность символов или шаблонов. Например, имена студентов можно представить в виде строк. Строка будет вероятным исследуемым объектом при работе с "текстом", "символами", "шаблонами" или "метками".

Для всех этих фундаментальных структур имеются соответствующие алгоритмы, которые представлены в каталоге задач во второй части этой книги. Важно ознакомиться с

этими задачами, т. к. они они изложены на языке, типичном для моделирования приложений. Чтобы научиться свободно владеть этим языком, просмотрите задачи в каталоге и изучите рисунки входа и выхода для каждой из них. Разобравшись в этих задачах, хотя бы на уровне рисунков и формулировок, вы будете знать, где искать возможный ответ в случае возникновения проблем в разработке вашего приложения.

В книге также представлены примеры успешного применения моделирования приложений в виде описаний решений реальных задач. Однако здесь необходимо высказать одно предостережение. Моделирование сводит разрабатываемое приложение к одной из многих существующих задач и структур. Такой процесс по своей природе является ограничивающим, и некоторые нюансы модели могут не соответствовать вашей конкретной задаче. Кроме того, встречаются задачи, которые можно моделировать несколькими разными способами.

Моделирование является всего лишь первым шагом в разработке алгоритма решения задачи. Внимательно отнеситесь к отличиям вашего приложения от потенциальной модели, но не спешите с заявлением, что ваша задача является уникальной и особенной. Временно игнорируя детали, которые не вписываются в модель, вы сможете найти ответ на вопрос, действительно ли они являются принципиально важными.

Подведение итогов

Моделирование разрабатываемого приложения в терминах стандартных структур и алгоритмов является важнейшим шагом в поиске решения.

1.4.2. Рекурсивные объекты

Научившись мыслить рекурсивно, вы научитесь определять большие сущности, состоящие из меньших сущностей *точно такого же типа, как и большие*. Например, если рассматривать дом как набор комнат, то при добавлении или удалении комнаты дом остается домом.

В мире алгоритмов рекурсивные структуры встречаются повсеместно. Более того, каждую из ранее описанных абстрактных структур можно считать рекурсивной. На рис. 1.9 видно, как легко они разбиваются на составляющие.

Перечислим возможные рекурсивные объекты.

- *Перестановки*. Удалив первый элемент перестановки $\{1,...,n\}$, мы получим перестановку остающихся $n - 1$ элементов.
- *Подмножества*. Каждое множество элементов $\{1,...,n\}$ содержит подмножество $\{1,...,n - 1\}$, являющееся результатом удаления элемента n, если такой имеется.
- *Деревья*. Что мы получим, удалив корень дерева? Правильно, коллекцию меньших деревьев. А что мы получим, удалив какой-либо лист дерева? Немного меньшее дерево.
- *Графы*. Удалите любую вершину графа, и вы получите меньший граф. Теперь разобьем вершины графа на две группы, правую и левую. Разрезав все ребра, соединяющие эти группы, мы получим два меньших графа и набор разорванных ребер.

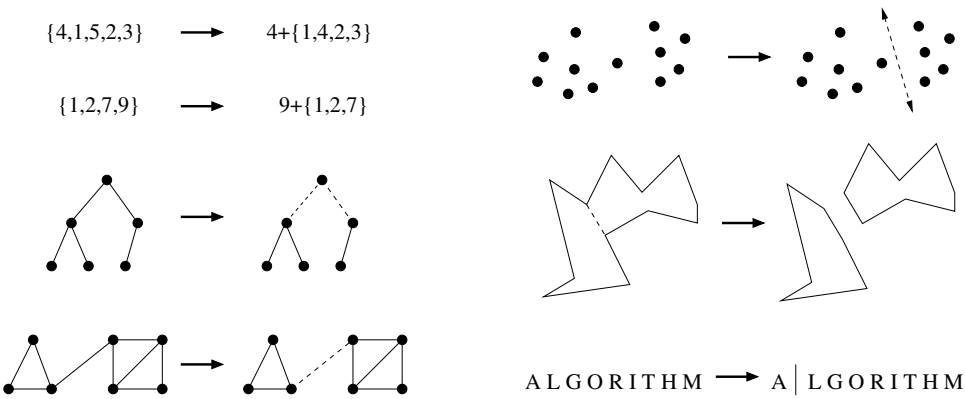


Рис. 1.9. Рекурсивное разложение комбинаторных объектов: перестановки, подмножества, деревья, графы, множества точек, многоугольники и строки

- *Точки*. Возьмем облако точек и разобьем его линией на две группы. Теперь у нас есть два меньших облака точек.
- *Многоугольники*. Соединив хордой две несмежные вершины простого многоугольника с n вершинами, мы получим два меньших многоугольника.
- *Строки*. Что мы получим, удалив первый символ строки? Другую, более короткую строку.

Для рекурсивного описания объектов требуются как правила разложения, так и базовые объекты, а именно спецификация простейшего объекта, далее которого разложение не идет. Такие базовые объекты обычно легко определить. Перестановки и подмножества нулевого количества объектов обозначаются как {}. Наименьшее дерево или граф состоят из одной вершины, а наименьшее облако точек состоит из одной точки. С многоугольниками немного посложнее; наименьшим многоугольником является треугольник. Наконец, пустая строка содержит нулевое количество знаков. Решение о том, содержит ли базовый объект нулевое количество элементов или один элемент, является, скорее, вопросом вкуса и удобства, нежели какого-либо фундаментального принципа.

Такие рекурсивные разложения применяются для определения многих алгоритмов, рассматриваемых в этой книге. Обращайте на них внимание.

1.5. Истории из жизни

Самый лучший способ узнать, какое огромное влияние тщательная разработка алгоритма может иметь на производительность, — ознакомиться с примерами из реальной жизни. Внимательно изучая опыт других людей, мы учимся использовать этот опыт в нашей собственной работе.

В различных местах этой книги приводится несколько рассказов об успешном (а иногда и неуспешном) опыте нашей команды в разработке алгоритмов решения реальных задач. Я надеюсь, что вы сможете перенять и усвоить опыт и знания, полученные нами

в процессе этих разработок, чтобы использовать их в качестве моделей для ваших собственных решений.

Все, изложенное в этих рассказах, действительно произошло. Конечно же, в изложении истории слегка приукрашены, и диалоги в них были отредактированы. Но я приложил все усилия, чтобы правдиво описать весь процесс, начиная от постановки задачи до выдачи решения, чтобы вы могли проследить его в действии.

В своих рассказах я пытаюсь уловить образ мышления алгориста в процессе решения задачи.

Эти истории из жизни обычно затрагивают, по крайней мере, одну, а часто несколько, задач из каталога во второй части книги. Когда такая задача встречается, дается ссылка на соответствующий раздел каталога. Таким образом подчеркиваются достоинства моделирования разрабатываемого приложения в терминах стандартных алгоритмических задач. Пользуясь каталогом задач, вы сможете в любое время получить известную информацию о решаемой задаче.

1.6. История из жизни. Моделирование проблемы ясновидения

Я занимался обычными делами, когда на меня нежданно-негаданно свалился этот запрос в виде телефонного звонка.

— Профессор Скиена, я надеюсь, что Вы сможете помочь мне. Я — президент компании Lotto System Group, Inc., и нам нужен алгоритм решения проблемы, возникающей в нашем последнем продукте.

— Буду рад помочь Вам, — ответил я. В конце концов, декан моего инженерного факультета всегда призывает преподавательский состав к более широкому взаимодействию с производством.

— Наша компания продает программу, предназначенную для развития способностей наших клиентов к ясновидению и предсказанию выигрышных лотерейных номеров[1]. Стандартный лотерейный билет содержит шесть номеров, выбираемых из большего количества последовательных номеров, например, от 1 до 44. Таким образом, шансы выигрыша любой комбинации номеров очень небольшие. Но после соответствующей тренировки наши клиенты могут мысленно увидеть, скажем, 15 номеров из 44 возможных, по крайней мере, четыре из которых будут на выигрышном билете. Вы все пока понимаете?

— Скорее нет, чем да, — сказал я, но тут вспомнил, что наш декан призывает нас взаимодействовать с производством.

— Наша проблема заключается в следующем. После того, как ясновидец сузил выбор номеров от 44 до 15, из которых он уверен в правильности, по крайней мере, четырех, нам нужно найти эффективный способ применить эту информацию. Допустим, угадавшему, по крайней мере, три правильных номера выдается денежный приз. Нам ну-

[1] Это реальный случай.

жен алгоритм, чтобы составить наименьший набор билетов, которые нужно купить, чтобы гарантировать выигрыш, по крайней мере, одного приза.

— При условии, что ясновидец не ошибается?

— Да, при условии, что ясновидец не ошибается. Нам нужна программа, которая распечатывает список всех возможных комбинаций выигрышных номеров билетов, которые он должен купить с минимальными затратами. Можете ли вы помочь нам с решением этой задачи?

Может быть, они и в самом деле были ясновидцами, т. к. они обратились как раз туда, куда надо. Определение наилучшего подмножества номеров билетов попадает в разряд комбинаторных задач. А точнее, это какой-то тип задачи о покрытии множества, в которой каждый покупаемый билет "покрывает" некоторые из возможных четырехэлементных подмножеств увиденного ясновидцем пятнадцатиэлементного множества. Определение наименьшего набора билетов для покрытия всех возможностей представляет собой особый экземпляр NP-полной задачи о *покрытии множества* (рассматривается в *разделе 18.1*) и считается вычислительно неразрешимой.

Данная задача действительно была особым экземпляром задачи о покрытии множества, полностью определяемая всего лишь четырьмя числами: размером n возможного множества S ($n \approx 15$), количеством номеров k на каждом билете ($k \approx 6$), количеством обещаемых ясновидцем выигрышных номеров j из множества S ($j = 4$) и, наконец, количеством совпадающих номеров l, необходимых, чтобы выиграть приз ($l = 3$). На рис. 1.10 показано покрытие экземпляра меньшего размера, где $n = 5$, $j = k = 3$, $l = 2$.

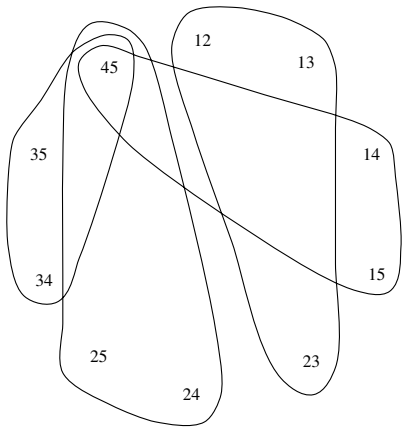


Рис. 1.10. Покрытие всех пар множества {1, 2, 3, 4, 5} номерами билетов {1, 2, 3}, {1, 4, 5}, {2, 4, 5}, {3, 4, 5}

— Хотя найти точный минимальный набор билетов с помощью эвристических методов будет трудно, я должен буду дать вам покрывающий набор номеров, достаточно близкий к самому дешевому, — ответил я ему. — Вам этого будет достаточно?

— При условии, что ваша программа создает лучший набор номеров, чем программа моего конкурента, это будет то, что надо. Его система не всегда гарантирует выигрыш. Очень признателен за вашу помощь, профессор Скиена.

— Один вопрос напоследок. Если с помощью вашей программы люди могут натренироваться выбирать выигрышные лотерейные билеты, то почему вы сами не пользуетесь ею, чтобы выигрывать в лотерею?

— Надеюсь встретиться с Вами в ближайшее время, профессор Скиена. Благодарю за помощь.

Я повесил трубку и начал обдумывать дальнейшие действия. Задача выглядела, как идеальный проект для какого-либо смышленого студента. После моделирования задачи посредством множеств и подмножеств основные компоненты решения выглядели довольно просто.

- Нам было нужна возможность генерировать все подмножества k номеров из потенциального множества S. Алгоритмы генерирования и ранжирования подмножеств представлены в *разделе 14.5*.
- Нам была нужна правильная формулировка, что именно означает требование иметь покрывающее множество приобретенных билетов. Очевидным критерием того, что требование выполнено, мог быть наименьший набор билетов, включающий, по крайней мере, один билет, содержащий каждое из $\binom{n}{l}$ l-подмножеств множества S, за которое может выдаваться приз.
- Нам нужно было отслеживать уже рассмотренные призовые комбинации. Нам нужны такие комбинации номеров билетов, чтобы покрыть как можно больше еще не охваченных призовых комбинаций. Текущие охваченные комбинации являются подмножеством всех возможных комбинаций. Структуры данных для подмножеств рассматриваются в *разделе 12.5*. Наилучшим кандидатом выглядел вектор битов, который бы за постоянное время давал ответ, охвачена ли уже данная комбинация.
- Нужен был механизм поиска, чтобы решить, какой следующий билет покупать. Для небольших множеств можно было проверить все возможные подмножества комбинаций и выбрать из них самую меньшую. Для множеств большего размера выигрышные комбинации номеров билетов для покупки можно было выбирать с помощью какого-либо процесса рандомизированного поиска наподобие имитации отжига (см. *раздел 7.5.3*), чтобы покрыть как можно больше комбинаций. Выполняя эту рандомизированную процедуру несколько раз и выбирая самые лучшие решения, мы смогли бы, скорее всего, получить хороший набор комбинаций номеров билетов.

Опуская детали механизма поиска, требуемый алгоритм можно выразить на псевдокоде, как показано в листинге 1.12.

Листинг 1.12. Поиск набора призовых комбинаций

```
LottoTicketSet (n,k,l)

  Инициализируем как "ложь" все элементы (n choose l)-элементного
                  вектора разрядов V
    While V содержит элементы "ложь"
      Выбираем k-элементное подмножество T множества {1,..., n}
      в качестве следующей комбинации номеров покупаемого билета
```

```
    For каждого из l-элементных подмножеств Ti множества T,
    V[rank(Ti)] = "истина"
  Выдаем набор комбинаций номеров билетов для покупки
```

Способный студент Фаяз Юнас (Fayyaz Younas) принял вызов реализовать алгоритм. На основе представленной основы он реализовал алгоритм поиска методом полного перебора и смог найти оптимальные решения задач, в которых $n \leq 5$. Для решения задачи с большим значением n он реализовал процедуру рандомизированного поиска, которую отлаживал, пока не остановился на самом лучшем варианте. Наконец наступил день, когда мы могли позвонить в компанию Lotto Systems Group и сказать им, что мы решили задачу.

— Результат работы нашей программы таков: оптимальным решением для $n = 15$, $k = 6$, $j = 4$, $l = 3$ будет покупка 28 билетов.

— Двадцать восемь билетов! — выразил недовольство президент. — В вашей программе, должно быть, есть ошибка. Вот пять билетов, которых будет достаточно, чтобы покрыть все варианты *дважды*: {2, 4, 8, 10, 13, 14}, {4, 5, 7, 8, 12, 15}, {1, 2, 3, 6, 11, 13}, {3, 5, 6, 9, 10, 15}, {1, 7, 9, 11, 12, 14}.

Мы повозились с этим примером немного и должны были признать, что он был прав. *Мы неправильно смоделировали задачу!* В действительности, нам не нужно было покрывать явно все возможные выигрышные комбинации. Объяснение представлено на рис. 1.11 в виде двухбилетного решения нашего предыдущего четырехбилетного примера.

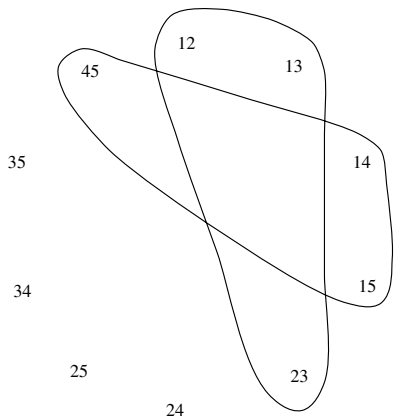


Рис. 1.11. Гарантирование выигрышной комбинации из двух номеров множества {l, 2, 3, 4, 5} при использовании только комбинации номеров {l, 2, 3} и {l, 4, 5}

Такие малообещающие комбинации, как {2, 3, 4} и {3, 4, 5}, имеют совпадающие пары комбинаций номеров в билетах, показанных на рис. 1.11. Мы пытались покрыть слишком много комбинаций, а дрожащие над каждой копейкой ясновидцы не желали платить за такую расточительность.

К счастью, у этой истории хороший конец. Общий принцип нашего поискового решения оставался применимым для реальной задачи. Все, что нужно было сделать, так это

исправить, какие подмножества засчитываются за покрытие данным набором комбинаций номеров билетов. Сделав это исправление, мы получили требуемые результаты. В компании Lotto Systems Group с благодарностью приняли нашу программу для внедрения в свой продукт. Будем надеяться, что они сорвут большой куш с ее помощью.

Мораль этой истории заключается в том, что необходимо удостовериться в правильности моделирования задачи, прежде чем пытаться решить ее. В нашем случае мы разработали правильную модель, но недостаточно глубоко проверили ее, перед тем, как начинать создавать программу на ее основе. Наше заблуждение было бы быстро выявлено, если бы, прежде чем приступать к решению реальной проблемы, мы проработали небольшой пример вручную и обсудили результаты с нашим клиентом. Но мы смогли выйти из этой ситуации с минимальными отрицательными последствиями благодаря правильности нашей первоначальной формулировки и использованию четко определенных абстракций для таких задач, как генерирование k-элементных подмножеств методом ранжирования, структуры данных множества и комбинаторного поиска.

Замечания к главе

В каждой хорошей книге, посвященной алгоритмам, отражается подход ее автора к их разработке. Тем, кто хочет ознакомиться с альтернативными точками зрения, особенно рекомендуется прочитать книги [CLRS01], [KT06] и [Man89].

Формальные доказательства правильности алгоритма являются важными и заслуживают более полного рассмотрения, чем можно предоставить в этой главе. Методы верификации программ обсуждаются в книге [Gri89].

Задача календарного планирования ролей в фильмах является особым случаем общей задачи *независимого множества*, которая рассматривается в *разделе 16.2*. В качестве входных экземпляров допускаются только интервальные графы, в которых вершины графа G можно представить линейными интервалами, а (i, j) является ребром G тогда и только тогда, когда интервалы пересекаются. Этот интересный и важный класс графов подробно рассматривается в книге [Gol04].

Колонка "Программистские перлы" Джона Бентли (Jon Bentley) является, наверное, самой известной коллекцией историй из жизни разработчиков алгоритмов. Колонка первоначально публиковалась в журнале "Communications of the ACM", а потом ее материалы были изданы в двух книгах, [Ben90] и [Ben99]. Еще одна прекрасная коллекция историй из жизни собрана в книге [Bro95]. Хотя эти истории имеют явный уклон в сторону разработки и проектирования программного обеспечения, они также представляют собой кладезь мудрости. Все программисты должны прочитать эти книги, чтобы получить и пользу, и удовольствие.

Наше решение задачи о покрытии множества лотерейных билетов более полно представлено в книге [YS96].

1.7. Упражнения

Поиск контрпримеров

1. [3] Докажите, что значение $a + b$ может быть меньшим, чем значение $\min(a, b)$.
2. [3] Докажите, что значение $a \times b$ может быть меньшим, чем значение $\min(a, b)$.

3. [5] Начертите сеть дорог с двумя точками *a* и *b*, такими, что маршрут между ними, преодолеваемый за кратчайшее время, не является самым коротким.

4. [5] Начертите сеть дорог с двумя точками *a* и *b*, самый короткий маршрут между которыми не является маршрутом с наименьшим количеством поворотов.

5. [4] Задача о рюкзаке: имея множество целых чисел $S = \{s_1, s_2, ..., s_n\}$ и целевое число T, найти такое подмножество множества S, сумма которого в точности равна T. Например, множество $S = \{1, 2, 5, 9, 10\}$ содержит такое подмножество, сумма элементов которого равна $T = 22$, но не $T = 23$.

   Найти контрпримеры для каждого из следующих алгоритмов решения задачи о рюкзаке, т. е., нужно найти такое множество S и число T, при которых подмножество, выбранное с помощью данного алгоритма, не до конца заполняет рюкзак, хотя правильное решение и существует:

   - вкладывать элементы множества S в рюкзак в порядке слева направо, если они подходят (т. е., алгоритм "первый подходящий");
   - вкладывать элементы множества S в рюкзак в порядке от наименьшего до наибольшего (т. е., используя алгоритм "первый лучший");
   - вкладывать элементы множества S в рюкзак в порядке от наибольшего до наименьшего.

6. [5] Задача о покрытии множества: имея семейство подмножеств $S_1, ..., S_m$ универсального множества $U = \{1,...,n\}$, найдите семейство подмножеств $T \subset S$ наименьшей мощности, чтобы $\bigcup_{t_i} \in T^{t_i} = U$. Например, для семейства подмножеств $S_1 = \{1, 3, 5\}$, $S_2 = \{2, 4\}$, $S_3 = \{1, 4\}$ и $S_4 = \{2, 5\}$ покрытием множества будет семейство подмножеств S_1 и S_2. Приведите контрпример для следующего алгоритма: выбираем самое мощное подмножество для покрытия, после чего удаляем все его элементы из универсального множества; повторяем добавление подмножества, содержащего наибольшее количество неохваченных элементов, пока все элементы не будут покрыты.

Доказательство правильности

7. [3] Докажите правильность следующего рекурсивного алгоритма умножения двух натуральных чисел для всех целочисленных констант $c \geq 2$:

```
function multiply(y, z)
               comment Return произведение yz.
        1.      if z = 0 then return(0) else
        2.      return(multiply(cy,[z/c]) + y·(z mod c))
```

8. [3] Докажите правильность следующего алгоритма вычисления полинома $P(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$:

```
function horner(A, x)
  p = An
  for i from n-1 to 0
    p = p * x + Ai
  return p
```

9. [3] Докажите правильность следующего алгоритма сортировки:

```
function bubblesort (A : list[1... n])
  var int i,j
```

```
for i from n to 1
  for j from 1 to i - 1
    if (A[j]>A[j+1]
      меняем местами значения A[j] и A[j + 1]
```

Математическая индукция

Для доказательства пользуйтесь методом математической индукции.

10. [3] Докажите, что $\sum_{i=1}^{n} i = n(n+1)/2$ для $n \geq 0$.

11. [3] Докажите, что $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$ для $n \geq 0$.

12. [3] Докажите, что $\sum_{i=1}^{n} i^3 = n^2(n+1)^2/4$ для $n \geq 0$.

13. [3] Докажите, что $\sum_{i=1}^{n} i(i+1)(i+2) = n(n+1)(n+2)(n+3)/4$.

14. [5] Докажите, что $\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$ для $n \geq 1$, $a \neq 1$.

15. [3] Докажите, что $\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{n+1}$ для $n \geq 0$.

16. [3] Докажите, что $n^3 + 2n$ делится на 3 для $n \geq 0$.

17. [3] Докажите, что дерево с n вершинами имеет в точности $n - 1$ ребер.

18. [3] Докажите, что сумма кубов первых n положительных целых чисел равна квадрату суммы этих целых чисел, т. е.,

$$\sum_{i=1}^{n} i^3 = (\sum_{i=1}^{n} i)^2$$

Приблизительные подсчеты

19. [3] Содержат ли все ваши книги, по крайней мере, миллион страниц? Каково общее количество страниц всех книг в вашей институтской библиотеке?

20. [3] Сколько слов содержит эта книга?

21. [3] Сколько часов составляет один миллион секунд? А сколько дней? Выполните все необходимые вычисления в уме.

22. [3] Сколько городов и поселков в Соединенных Штатах?

23. [3] Сколько кубических километров воды изливается из устья Миссисипи каждый день? Не пользуйтесь никакой справочной информацией. Опишите все предположения, сделанные вами для получения ответа.

24. [3] В каких единицах измеряется время доступа к жесткому диску, в миллисекундах (тысячных долях секунды) или микросекундах (миллионных долях секунды)? Сколько времени занимает доступ к слову в оперативной памяти вашего компьютера, больше или меньше микросекунды? Сколько инструкций может выполнить центральный про-

цессор вашего компьютера в течение года, если компьютер постоянно держать включенным?

25. [4] Алгоритм сортировки выполняет сортировку 1 000 элементов за 1 секунду. Сколько времени займет сортировка 10 000 элементов,
    - если время исполнения алгоритма прямо пропорционально n^2?
    - если время исполнения алгоритма, по грубым оценкам, пропорционально $n\log n$?

Проекты по реализации

26. [5] Реализуйте два эвристических алгоритма решения задачи коммивояжера из *раздела 1.1*. Какой из них выдает на практике более качественные решения? Можете ли вы предложить эвристический алгоритм, работающий лучше любого из них?
27. [5] Опишите способ проверки достаточности покрытия данным множеством комбинаций номеров лотерейных билетов из задачи в *разделе 1.6*. Напишите программу поиска хороших множеств комбинаций номеров билетов.

Задачи, предлагаемые на собеседовании

28. [5] Напишите функцию деления целых чисел, которая не использует ни оператор деления (/), ни оператор умножения (*). Функция должна быть быстродействующей.
29. [5] У вас есть 25 лошадей. В каждой скачке может участвовать не больше 5 лошадей. Требуется определить первую, вторую и третью по скорости лошадь. Найдите минимальное количество скачек, позволяющих решить эту задачу.
30. [3] Сколько настройщиков пианино во всем мире?
31. [3] Сколько бензоколонок в Соединенных Штатах?
32. [3] Сколько весит лед на хоккейном поле?
33. [3] Сколько километров дорог в Соединенных Штатах?
34. [3] Сколько раз, в среднем, нужно открыть наугад телефонный справочник Манхэттена, чтобы найти определенного человека?

Задачи по программированию

Эти задачи доступны на сайтах **http://www.programming-challenges.com** и **http://uva.onlinejudge.org**. Идентификатор задачи на соответствующем сайте указывается после названия задачи в форме "число/число". Так как сайты англоязычные, названия задач даются на исходном языке.

1. The $3n + 1$ problem. 110101/100.
2. The Trip. 110103/10137.
3. Australian Voting. 110108/10142.

ГЛАВА 2

Анализ алгоритмов

Алгоритмы являются принципиально важным компонентом информатики, т. к. их изучение не требует использования языка программирования или компьютера. Это означает необходимость в методах, позволяющих сравнивать эффективность алгоритмов, не прибегая к их реализации. Самыми значимыми из этих инструментов являются модель вычислений RAM и асимптотический анализ сложности наихудших случаев.

Для оценки производительности алгоритмов применяется асимптотическая нотация. Хотя практик может прийти в ужас от самой идеи теоретического анализа алгоритмов, этот материал представлен здесь в силу его исключительной ценности при работе с алгоритмами.

Этот способ оценки производительности является наиболее трудным материалом в данной книге. Но когда вы поймете основу этих идей на интуитивном уровне, вам будет намного легче разобраться в формальной составляющей.

2.1. Модель вычислений RAM

Разработка машинно-независимых алгоритмов основывается на гипотетическом компьютере, называющемся *машиной с произвольным доступом к памяти* (Random Access Machine) или RAM-машиной. Согласно этой модели наш компьютер работает таким образом:

- для исполнения любой *простой* операции (`+`, `*`, `-`, `=`, `if`, `call`) требуется ровно один временной шаг;
- циклы и подпрограммы не считаются простыми операциями, а состоят из нескольких простых операций. Нет смысла считать подпрограмму сортировки одношаговой операцией, т. к. для сортировки 1 000 000 элементов потребуется определенно намного больше времени, чем для сортировки десяти элементов. Время исполнения цикла или подпрограммы зависит от количества итераций или специфического характера подпрограммы;
- каждое обращение к памяти занимает один временной шаг. Кроме этого, наш компьютер обладает неограниченным объемом оперативной памяти. Кэш и диск в модели RAM не применяются.

Время исполнения алгоритма в RAM-модели вычисляется по общему количеству шагов, требуемых алгоритму для решения данного экземпляра задачи. Допуская, что наша RAM-машина исполняет определенное количество шагов/операций за секунду, количество шагов легко перевести в единицы времени.

Может показаться, что RAM-модель является слишком упрощенным представлением работы компьютеров. В конце концов, на большинстве процессоров умножение двух

чисел занимает больше времени, чем сложение, что не вписывается в первое предположение модели. Второе предположение может быть нарушено удачной оптимизацией цикла компилятором или гиперпотоковыми возможностями процессора. Наконец, время обращения к данным может значительно разниться в зависимости от расположения данных: в кэше, в оперативной памяти или на диске. Таким образом, по сравнению с настоящим компьютером, все три основные допущения для RAM-машины неверны.

Тем не менее, несмотря на эти несоответствия настоящему компьютеру, RAM-модель является *превосходной* моделью для понимания того, как алгоритм будет работать на настоящем компьютере. Она обеспечивает хороший компромисс, отражая поведение компьютеров и одновременно являясь простой в использовании. Эти характеристики делают RAM-модель полезной для практического применения.

Любая модель полезна лишь в определенных рамках. Возьмем, например, модель плоской Земли. Можно спорить, что это неправильная модель, т. к. еще древние греки знали, что в действительности Земля круглая. Но модель плоской Земли достаточно точна для закладки фундамента дома. Более того, в данном случае с моделью плоской Земли настолько удобнее работать, что использование модели сферической Земли[1] для этой цели даже не приходит в голову.

Та же самая ситуация наблюдается и в случае с RAM-моделью вычислений — мы создаем, вообще говоря, очень полезную абстракцию. Довольно трудно создать алгоритм, для которого RAM-модель выдаст существенно неверные результаты. Устойчивость RAM-модели позволяет анализировать алгоритмы машинно-независимым способом.

Подведение итогов

Алгоритмы можно изучать и анализировать, не прибегая к использованию конкретного языка программирования или компьютерной платформы.

2.1.1. Анализ сложности наилучшего, наихудшего и среднего случая

С помощью RAM-модели можно подсчитать количество шагов, требуемых алгоритму для исполнения любого экземпляра задачи. Но чтобы получить общее представление о том, насколько хорошим или плохим является алгоритм, нам нужно знать, как он работает со *всеми* экземплярами задачи.

Чтобы понять, что означает наилучший, наихудший и средний случай сложности алгоритма (т. е. время его исполнения в соответствующем случае), нужно рассмотреть исполнение алгоритма на всех возможных экземплярах входных данных. В случае задачи сортировки множество входных экземпляров состоит из всех возможных компоновок ключей n по всем возможным значениям n. Каждый входной экземпляр можно представить в виде точки графика (рис. 2.1), где ось x представляет размер входа задачи (для сортировки это будет количество элементов, подлежащих сортировке), а ось y — количество шагов, требуемых алгоритму для обработки данного входного экземпляра.

[1] В действительности, Земля не совсем сферическая, но модель сферической Земли удобна для работы с такими понятиями, как широта и долгота.

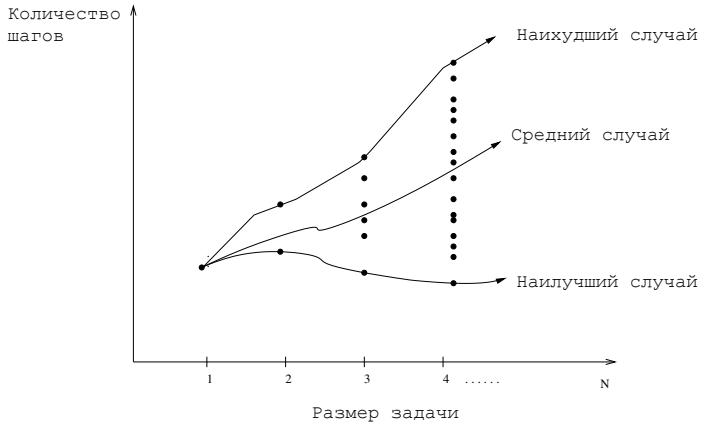


Рис. 2.1. Наилучший, наихудший и средний случай сложности алгоритма

Эти точки естественным образом выстраиваются столбцами, т. к. размер входа может быть только целым числом (т. е., сортировка 10,57 элементов лишена смысла). На графике этих точек можно определить три представляющих интерес функции:

- *сложность алгоритма в наихудшем случае* — это функция, определяемая максимальным количеством шагов, требуемых для обработки любого входного экземпляра размером *n*. Этот случай отображается кривой, проходящей через самую высшую точку каждого столбца;
- *сложность алгоритма в наилучшем случае* — это функция, определяемая минимальным количеством шагов, требуемых для обработки любого входного экземпляра размером *n*. Этот случай отображается кривой, проходящей через самую низшую точку каждого столбца;
- *сложность алгоритма в среднем случае* — это функция, определяемая средним количеством шагов, требуемых для обработки всех экземпляров размером *n*.

На практике наиболее важной является оценка сложности алгоритма в наихудшем случае. Для многих людей это противоречит здравому смыслу. Рассмотрим пример. Вы собираетесь посетить казино, имея в кармане *n* долларов. Каковы возможные исходы? В наилучшем случае вы выиграете само казино, но хотя такое развитие событий и возможно, вероятность его настолько ничтожна, что даже не стоит думать об этом. В наихудшем случае вы проиграете все свои *n* долларов; вероятность такого события удручающе высока и ее легко вычислить. Средний случай, когда типичный игрок проигрывает в казино 87,32% взятых с собой денег, трудно рассчитать и даже определить. Что именно означает *средний*? Глупые люди проигрывают больше, чем умные, так вы умнее или глупее, чем средний человек и насколько? Игроки в очко, которые умеют отслеживать сданные карты, в среднем выигрывают больше, чем игроки, которые не отказываются от нескольких бесплатных рюмок алкогольного напитка, регулярно предлагаемых симпатичными официантками. Чтобы избежать всех этих усложнений и получить самый полезный результат, мы и рассматриваем только наихудший случай.

Важно осознавать то, что в каждом случае сложность алгоритма определяется числовой функцией, соотносящей время с размером задачи. Эти функции определены так же

строго, как и любые другие числовые функции, будь то уравнение $y = x^2 - 2x + 1$ или цена акций Google в зависимости от времени. Но функции временной сложности настолько трудны для понимания, что перед началом работы их нужно упростить. Для этой цели используются асимптотические обозначения, в частности обозначение "*O*-большое".

2.2. Асимптотические обозначения

Временную сложность наилучшего, наихудшего и среднего случая для любого алгоритма можно представить как числовую функцию от размеров возможных экземпляров задачи. Но работать с этими функциями очень трудно, т. к. они обладают такими свойствами:

- *являются слишком волнистыми.* Время исполнения алгоритма, например, двоичного поиска, обычно меньше для массивов, имеющих размер $n = 2^k - 1$, где k — целое число. Эта особенность не имеет большого значения, но служит предупреждением, что *точная* функция временной сложности любого алгоритма вполне может иметь неровный график с небольшими выпуклостями и впадинами, как показано на рис. 2.2;

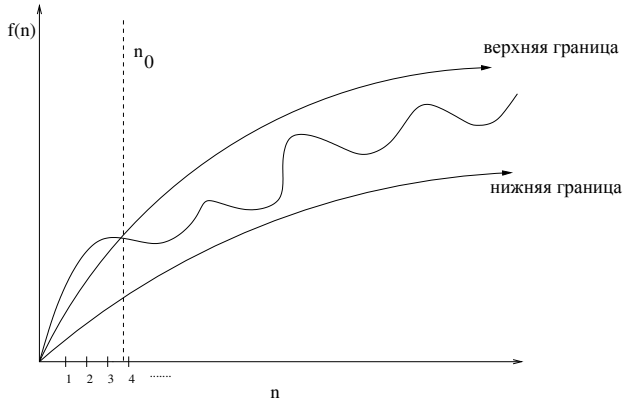


Рис. 2.2. Верхняя и нижняя границы, действительные для $n > n_0$, сглаживают волнистость сложных функций

- *требуют слишком много информации для точного определения.* Чтобы сосчитать точное количество инструкций *R*AM-машины, исполняемых в худшем случае, нужно, чтобы алгоритм был расписан в подробностях полной компьютерной программы. Более того, точность ответа зависит от маловажных деталей кодировки (например, был ли употреблен оператор `case` вместо вложенных операторов `if`). Точный анализ наихудшего случая, например, такого:

$$T(n) = 12754n^2 + 4353n + 834\lg_2 n + 13546$$

очевидно, был бы очень трудной задачей, решение которой не предоставляет нам никакой дополнительной информации, кроме той, что "с увеличением n временная сложность возрастает квадратически".

Оказывается, намного легче работать с верхней и нижней границами функций временной сложности, используя для этого асимптотические обозначения (*O*-большое и Ω-большое соответственно). Асимптотические обозначения позволяют упростить анализ, поскольку игнорируют детали, которые не влияют на сравнение эффективности алгоритмов.

В частности, в асимптотических обозначениях игнорируется разница между постоянными множителями. Например, в анализе с применением асимптотического обозначения функции $\mathbf{f}(n) = 2n$ и $g(n) = n$ являются идентичными. Это вполне логично в нашей ситуации. Допустим, что определенный алгоритм на языке C исполняется вдвое быстрее, чем тот же алгоритм на языке Java. Этот постоянный множитель, равняющийся двум, не предоставляет нам никакой информации собственно об алгоритме, т. к. в обоих случаях исполняется один и тот же алгоритм. При сравнении алгоритмов такие постоянные коэффициенты не принимаются во внимание.

Формальные определения, связанные с асимптотическими обозначениями, выглядят таким образом:

- $f(n) = O(g(n))$ означает, что функция $f(n)$ *ограничена сверху* функцией $c \cdot g(n)$. Иными словами, существует такая константа c, для которой $f(n) \le c \cdot g(n)$ при достаточно большом значении n (т. е. $n \ge n_0$ для некоторой константы n_0);
- $f(n) = \Omega(g(n))$ означает, что функция $f(n)$ *ограничена снизу* функцией $c \cdot g(n)$. Иными словами, существует такая константа c, для которой $f(n) \ge c \cdot g(n)$ для всех $n \ge n_0$;
- $f(n) = \Theta(g(n))$ означает, что функция $f(n)$ *ограничена сверху* функцией $c_1 \cdot g(n)$, а *снизу* функцией $c_2 \cdot g(n)$ для всех $n \ge n_0$. Иными словами, существуют константы c_1 и c_2, для которых $f(n) \le c_1 \cdot g(n)$ и $f(n) \ge c_2 \cdot g(n)$. Следовательно, функция $g(n)$ дает нам хорошие ограничения для функции $f(n)$.

Графическая иллюстрация этих определений дается на рис. 2.3.

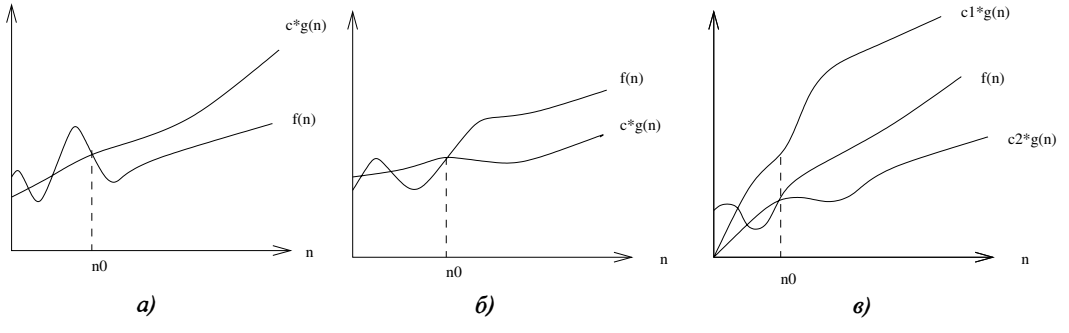


Рис. 2.3. Графическая иллюстрация асимптотических обозначений: *O*-большое *(а)*, Ω-большое *(б)*, Θ-большое *(в)*

В каждом из этих определений фигурирует константа n_0, после которой эти определения всегда верны. Нас не интересуют небольшие значения n, т. е. значения слева от n_0. В конце концов, нам безразлично, что один алгоритм может отсортировать, скажем, шесть или восемь элементов быстрее, чем другой. Мы ищем алгоритм для быстрой сортировки 10 000 или 1 000 000 элементов. В этом отношении асимптотические обо-

значения позволяют нам игнорировать несущественные детали и концентрироваться на общей картине.

> ***Подведение итогов***
>
> Анализ наихудшего случая и асимптотические обозначения являются инструментами, которые существенно упрощают задачу сравнения эффективности алгоритмов.

Обязательно убедитесь, что вы понимаете смысл асимптотических обозначений, изучив последующие примеры. Конкретные значения констант c и n_0 были выбраны потому, что они хорошо иллюстрируют ситуацию, но можно использовать и другие значения этих констант с точно таким же результатом. Вы можете выбрать любые другие значения констант, при которых сохраняется первоначальное неравенство; в идеале, следует выбирать такие значения, при которых очевидно, что неравенство соблюдается:

$3n^2 - 100n + 6 = O(n^2)$, т. к. выбрано $c = 3$ и $3n^2 > 3n^2 - 100n + 6$;

$3n^2 - 100n + 6 = O(n^3)$, т. к. выбрано $c = 1$ и $n^3 > 3n^2 - 100n + 6$ при $n > 3$;

$3n^2 - 100n + 6 \neq O(n)$, т. к. для любого значения c выбрано $cn < 3n^2$ при $n > c$;

$3n^2 - 100n + 6 = \Omega(n^2)$, т. к. выбрано $c = 2$ и $2n^2 < 3n^2 - 100n + 6$ при $n > 100$;

$3n^2 - 100n + 6 \neq \Omega(n^3)$, т. к. выбрано $c = 3$ и $3n^2 - 100n + 6 < n^3$ при $n > 3$;

$3n^2 - 100n + 6 = \Omega(n)$, т. к. для любого значения c выбрано $cn < 3n^2 - 100n + 6$ при $n > 100c$;

$3n^2 - 100n + 6 = \Theta(n^2)$, т. к. применимо как O, так и Ω;

$3n^2 - 100n + 6 \neq \Theta(n^3)$, т. к. применимо только O;

$3n^2 - 100n + 6 \neq \Theta(n)$, т. к. применимо только Ω.

Асимптотические обозначения позволяют получить приблизительное представление о равенстве функций при их сравнении. Выражение типа $n^2 = O(n^3)$ может выглядеть странно, но его значение всегда можно уточнить, пересмотрев его определение в терминах верхней и нижней границ. Возможно, это обозначение будет более понятным, если в данном случае рассматривать символ равенства (=) как означающий "одна из функций, принадлежащих к множеству функций". Очевидно, что n^2 является одной из функций, принадлежащих множеству функций $O(n^3)$.

Остановка для размышлений. Возвращение к определению

ЗАДАЧА. Верно ли равенство $2^{n+1} = \Theta(2^n)$?

Решение. Для разработки оригинальных алгоритмов требуется умение и вдохновение. Но при использовании асимптотических обозначений лучше всего подавить все свои творческие инстинкты. Все задачи по асимптотическим обозначениям можно правильно решить, работая с первоначальным определением.

- Верно ли равенство $2^{n+1} = O(2^n)$? Очевидно, что $f(n) = O(g(n))$ тогда и только тогда, когда существует такая константа c, при которой для всех достаточно больших зна-

чений n функция $f(n) \le c \cdot g(n)$. Существует ли такая константа? Заметим, что $2^{n+1} = 2 \cdot 2^n$, откуда следует, что $2 \cdot 2^n \le c \cdot 2^n$ для всех $c \ge 2$.

- Верно ли равенство $2^{n+1} = \Omega(2^n)$? Согласно определению, $f(n) = \Omega(g(n))$ тогда и только тогда, когда существует такая константа $c > 0$, при которой для всех достаточно больших значений n функция $f(n) \ge c \cdot g(n)$. Это условие удовлетворяется для любой константы $0 < c \le 2$. Границы O-большое и Ω-большое совместно подразумевают $2^{n+1} = \Theta(2^n)$. ■

Остановка для размышлений. Квадраты

ЗАДАЧА. Верно ли равенство $(x + y)^2 = O(x^2 + y^2)$?

Решение. При малейших трудностях в работе с асимптотическим обозначением немедленно возвращаемся к его определению, согласно которому это выражение действительно тогда и только тогда, когда мы можем найти такое значение c, при котором $(x + y)^2 \le c(x^2 + y^2)$.

Лично я первым делом раскрыл бы скобки в левой части уравнения, т. е. $(x + y)^2 = x^2 + 2xy + y^2$. Если бы в развернутом выражении отсутствовал член $2xy$, то вполне очевидно, что неравенство соблюдалось бы для любого значения $c > 1$. Но т. к. этот член имеется, то нам нужно рассмотреть его связь с выражением $x^2 + y^2$. Если $x \le y$, то $2xy \le 2y^2 \le 2(x^2 + y^2)$. А если $x \ge y$, то $2xy \le 2x^2 \le 2(x^2 + y^2)$. В любом случае теперь мы можем ограничить это выражение двойным значением функции с правой стороны. Это означает, что $(x + y)^2 \le 3(x^2 + y^2)$, поэтому результат остается в силе. ■

2.3. Скорость роста и отношения доминирования

Используя асимптотические обозначения, мы пренебрегаем постоянными множителями, не учитывая их при вычислении функций. При таком подходе функции $f(n) = 0{,}001n^2$ и $g(n) = 1000n^2$ для нас одинаковы, несмотря на то, что значение функции $g(n)$ в миллион раз больше значения функции $f(n)$ для любого n.

Причина, по которой достаточно грубого анализа, предоставляемого обозначением O-большое, приводится в табл. 2.1, в которой перечислены наиболее распространенные функции и их значения для нескольких значений n. В частности, здесь можно увидеть время исполнения $f(n)$ операций алгоритмов на быстродействующем компьютере, исполняющем каждую операцию за одну наносекунду (10^{-9} секунд). На основе представленной в таблице информации можно сделать следующие выводы:

- время исполнения всех этих алгоритмов примерно одинаково для значений $n = 10$;
- любой алгоритм с временем исполнения $n!$ становится бесполезным для значений $n \ge 20$;
- диапазон алгоритмов с временем исполнения 2^n несколько шире, но они также становятся непрактичными для значений $n > 40$;
- алгоритмы с квадратичным временем исполнения n^2 применяются при $n \le 10\,000$, после чего их производительность начинает резко ухудшаться. Эти алгоритмы, скорее всего, будут бесполезны для значений $n > 1\,000\,000$;

- алгоритмы с линейным и логарифмическим временем исполнения остаются полезными при обработке миллиарда элементов;
- в частности, алгоритм $O(\lg n)$ без труда обрабатывает любое вообразимое количество элементов.

Таблица 2.1. Скорость роста основных функций

$n/f(n)$	$\lg n$	n	$n\lg n$	n^2	2^n	$n!$
10	0,003 мкс	0,01 мкс	0,033 мкс	0,1 мкс	1 мкс	3,63 мс
20	0,004 мкс	0,02 мкс	0,086 мкс	0,4 мкс	1 мс	77,1 лет
30	0,005 мкс	0,03 мкс	0,147 мкс	0,9 мкс	1 с	8.4×10^{15} лет
40	0,005 мкс	0,04 мкс	0,213 мкс	1,6 мкс	18,3 мин	
50	0,006 мкс	0,05 мкс	0,282 мкс	2,5 мкс	13 дней	
100	0,007 мкс	0,1 мкс	0,644 мкс	10 мкс	4×10^{13} лет	
1 000	0,010 мкс	1,00 мкс	9,966 мкс	1 мс		
10 000	0,013 мкс	10 мкс	130 мкс	100 мс		
100 000	0,017 мкс	0,10 мс	1,67 мс	10 с		
1 000 000	0,020 мкс	1 мс	19,93 мс	16,7 мин		
10 000 000	0,023 мкс	0,01 с	0,23 с	1,16 дней		
100 000 000	0,027 мкс	0,10 с	2,66 с	115,7 дней		
1 000 000 000	0,030 мкс	1 с	29,90 с	31,7 лет		

Из этого можно сделать основной вывод, что, даже игнорируя постоянные множители, мы получаем превосходное общее представление о годности алгоритма для решения задачи определенного размера. Алгоритм с временем исполнения $f(n) = n^3$ секунд победит алгоритм с временем исполнения $g(n) = 1\,000\,000 \cdot n^2$ секунд только при $n < 1\,000\,000$. На практике такая громадная разница между постоянными множителями алгоритмов встречается намного реже, чем задачи по обработке большего объема входных данных.

2.3.1. Отношения доминирования

Посредством асимптотических обозначений функции разбиваются на классы, в каждом из которых сгруппированы функции с эквивалентным асимптотическим обозначением. Например, функции $f(n) = 0{,}34n$ и $g(n) = 234{,}234n$ принадлежат к одному и тому же классу, а именно к классу порядка $\Theta(n)$. Кроме того, когда функции f и g принадлежат к разным классам, они являются разными относительно нашего обозначения. Иными словами, справедливо либо $f(n) = O(g(n))$, либо $g(n) = O(f(n))$, но не оба равенства одновременно.

Говорят, что функция с более быстрым темпом роста *доминирует* над менее быстро растущей функцией точно так же, как более быстро растущая страна в итоге начинает доминировать над отстающей. Когда функции f и g принадлежат к разным классам

(т. е. $f(n) \neq \Theta(g(n))$), говорят, что функция f *доминирует* над функцией g, когда $f(n) = O(g(n))$. Это отношение иногда обозначается как $g \gg f$.

К счастью, процесс базового анализа алгоритмов обычно порождает лишь небольшое количество классов функций, достаточное для покрытия почти всех алгоритмов, рассматриваемых в этой книге. Далее приводятся эти классы в порядке возрастания доминирования.

- *Функции-константы*, $f(n) = 1$. Такие функции могут измерять трудоемкость сложения двух номеров, распечатывания какого-либо текста или рост таких функций, как $f(n) = \min(n,100)$. По большому счету, зависимость от параметра n отсутствует.
- *Логарифмические функции*, $f(n) = \log n$. Логарифмическая временная сложность проявляется в таких алгоритмах, как двоичный поиск. С увеличением n такие функции возрастают довольно медленно, но быстрее, чем функции-константы (которые вообще не возрастают). Логарифмы рассматриваются более подробно в *разделе 2.6*.
- *Линейные функции*, $f(n) = n$. Такие функции измеряют трудоемкость просмотра каждого элемента в массиве элементов один раз (или два раза, или десять раз), например, для определения наибольшего или наименьшего элемента или для вычисления среднего значения.
- *Суперлинейные функции*, $f(n) = n\lg n$. Этот важный класс функций возникает в таких алгоритмах, как Quicksort и Mergesort. Эти функции возрастают лишь немного быстрее, чем линейные (см. табл. 2.1), но достаточно быстро, чтобы составить другой класс доминирования.
- *Квадратичные функции*, $f(n) = n^2$. Эти функции измеряют трудоемкость просмотра большинства или всех пар элементов в универсальном множестве из n элементов. Они возникают в таких алгоритмах, как сортировка вставками или сортировка методом выбора.
- *Кубические функции*, $f(n) = n^3$. Эти функции возникают при перечислении всех триад элементов в универсальном множестве из n элементов. Они также возникают в определенных алгоритмах динамического программирования, которые рассматриваются в *главе 8*.
- *Показательные функции*, $f(n) = c^n$, *константа* $c > 1$. Эти функции возникают при перечислении всех подмножеств множества из n элементов. Как мы видели в табл. 2.1, экспоненциальные алгоритмы быстро становятся бесполезными с увеличением количества элементов n. Впрочем, не так быстро, как функции из следующего класса.
- *Факториальные функции*, $f(n) = n!$. Факториальные функции определяют все перестановки n элементов.

Тонкости отношений доминирования рассматриваются в *разделе 2.9.2*, но в действительности вы должны помнить лишь следующее отношение:

$$n! \gg 2^n \gg n^3 \gg n^2 \gg n\log n \gg n \gg \log n \gg 1$$

Подведение итогов

Хотя анализ алгоритмов высокого уровня может порождать экзотические функции, в большинстве практических алгоритмов применяется лишь небольшой ассортимент функций временной сложности, которого является вполне достаточно.

2.4. Работа с асимптотическими обозначениями

Для работы с асимптотическими обозначениями нужно повторить тему упрощения алгебраических выражений, которая изучается в школе. Большинство полученных в школе знаний также применимо и для асимптотических обозначений.

2.4.1. Сложение функций

Сумма двух функций определяется доминантной функцией, а именно:

$O(f(n)) + O(g(n)) = O(\max(f(n),(g(n)))$

$\Omega(f(n)) + \Omega(g(n)) = \Omega(\max(f(n),(g(n)))$

$\Theta(f(n)) + \Theta(g(n)) = \Theta(\max(f(n),(g(n)))$

Это обстоятельство очень полезно при упрощении выражений, т. к. оно подразумевает, что $n^3 + n^2 + n + 1 = O(n^3)$. По сравнению с доминантным членом все остальное является несущественным.

На интуитивном уровне это объясняется таким образом. По крайней мере, половина общего объема суммы $f(n) + g(n)$ должна предоставляться большей функцией. По определению, при $n \to \infty$ доминантная функция будет предоставлять большую часть объема суммы функций. Соответственно, отбросив меньшую функцию, мы уменьшим значение суммы максимум в два раза, что равносильно постоянному множителю 1/2. Предположение, что $f(n) = O(n^2)$ и $g(n) = O(n^2)$ также влечет за собой, что $f(n) + g(n) = O(n^2)$.

2.4.2. Умножение функций

Умножение можно рассматривать, как повторяющееся сложение. Рассмотрим умножение на любую константу $c > 0$, будь это 1,02 или 1 000 000. Умножение функции на константу не может повлиять на ее асимптотическое поведение, т. к. в анализе функции $c \cdot f(n)$ с применением обозначения "O-большое" мы можем умножить ограничивающие константы на $1/c$, что даст нам необходимые константы для анализа. Таким образом:

$O(cf(n)) \to O(f(n))$

$\Omega(cf(n)) \to \Omega(f(n))$

$\Theta(cf(n)) \to \Theta(f(n))$

Конечно же, во избежание неприятностей, константа c должна быть строго положительной, т. к. мы можем свести на нет даже самую быстрорастущую функцию, умножив ее на ноль.

С другой стороны, когда обе функции произведения возрастают, то обе являются важными. Функция $O(n!\log n)$ доминирует над функцией $n!$ точно так же, как и функция $\log n$ доминирует над константой 1. В общем,

$O(f(n))*O(g(n)) \to O(f(n)*g(n))$

$\Omega(f(n))*\Omega(g(n)) \to \Omega(f(n)*g(n))$

$\Theta(f(n))*\Theta(g(n)) \to \Theta(f(n)*g(n))$

Остановка для размышлений. Транзитивность

ЗАДАЧА. Доказать, что отношение O-большое обладает транзитивностью, т. е. если $f(n) = O(g(n))$ и $g(n) = O(h(n))$, тогда $f(n) = O(h(n))$.

Решение. Работая с асимптотическими обозначениями, мы всегда обращаемся к определению. Нам нужно показать, что $f(n) \leq c_3 h(n)$ при $n > n_3$ при условии, что $f(n) \leq c_1 g(n)$ и $g(n) \leq c_2 h(n)$ при $n > n_1$ и $n > n_2$ соответственно. Из этих неравенств получаем: $f(n) \leq c_1 g(n) \leq c_1 c_2 h(n)$ при $n > n_3 = \max(n_1, n_2)$. ■

2.5. Оценка эффективности

Как правило, общую оценку времени исполнения алгоритма можно легко получить при наличии точного письменного изложения алгоритма. В этом разделе рассматривается несколько примеров, возможно даже более подробно, чем необходимо.

2.5.1. Сортировка методом выбора

В этом разделе анализируется алгоритм сортировки методом выбора. При сортировке этим способом определяется наименьший неотсортированный элемент и помещается в конец отсортированной части массива. Процедура повторяется до тех пор, пока все элементы массива не будут отсортированы. Графическая иллюстрация работы алгоритма представлена на рис. 2.4, а соответствующий код на языке C в листинге 2.1.

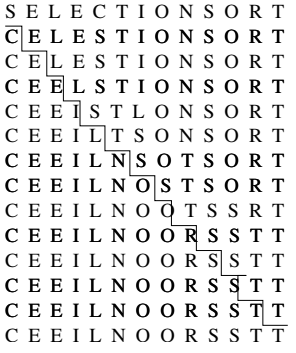


Рис. 2.4. Графическая иллюстрация работы алгоритма сортировки методом выбора

Листинг 2.1. Реализация алгоритма сортировки методом выбора на языке C

```
selection_sort(int s[], int n)
{
  int i,j;                  /* Счетчики */
  int min;                  /* Указатель наименьшего элемента */

  for (i=0; i<n; i++) {
    min=i;
    for (j=i+1; j<n; j++)
```

```
      if (s[j] < s[min]) min=j;
    swap(&s[i],&s[min]);
  }
}
```

Внешний цикл исполняется n раз. Внутренний цикл исполняется $n - i - 1$ раз, где i — счетчик внешнего цикла. Точное количество исполнений оператора `if` определяется следующей формулой:

$$S(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} n - i - 1$$

Это формула сложения целых чисел в убывающем порядке, начиная с $n - 1$, т. е.:

$$S(n) = (n-1) + (n-2) + (n-3) + ... + 2 + 1$$

Какие выводы мы можем сделать на основе такой формулы? Чтобы получить точное значение, необходимо применить методы, рассматриваемые в *разделе 1.3.5*. Но при работе с асимптотическими обозначениями нас интересует только *степень* выражения. Один из подходов — считать, что мы складываем $n - 1$ элементов, среднее значение которых равно приблизительно $n/2$. Таким образом мы получаем $S(n) \approx n(n-1)/2$.

Другим подходом будет использование верхней и нижней границ. Мы имеем не более n элементов, значение каждого из которых не превышает $n - 1$. Таким образом, $S(n) \leq \leq n(n-1) = O(n^2)$. Также мы имеем $n/2$ элементов, чье значение больше чем $n/2$. Соответственно, $S(n) \geq (n/2) \times (n/2) = \Omega(n^2)$. Все это говорит нам, что время исполнения равно $\Theta(n^2)$, т. е. сложность сортировки методом выбора является квадратичной.

2.5.2. Сортировка вставками

Основное практическое правило при асимптотическом анализе гласит, что время исполнения алгоритма в наихудшем случае получается умножением наибольшего возможного количества итераций каждого вложенного цикла. Рассмотрим, например, алгоритм сортировки вставками из листинга 1.1, внутренние циклы которого приведены в листинге 2.2.

Листинг 2.2. Внутренние циклы алгоритма сортировки вставками на языке C

```
for (i=1; i<n; i++) {
     j=i;
     while ((j>0) && (s[j] < s[j-1])) {
          swap(&s[j] ,&s[j-1]);
          j = j-i;
     }
}
```

Сколько итераций осуществляет внутренний цикл `while`? На этот вопрос сложно дать однозначный ответ, т. к. цикл может быть остановлен досрочно, если произойдет выход за границы массива (`j>0`) или элемент окажется на должном месте в отсортированной части массива (`s[j]<s[j-1]`). Так как в анализе наихудшего случая мы ищем верхнюю границу времени исполнения, то мы игнорируем досрочное завершение и полага-

ем, что количество исполняемых в этом цикле итераций всегда будет i. Более того, мы можем допустить, что *всегда* исполнятся n итераций, т. к. $i < n$. А т. к. внешний цикл исполняется n раз, то алгоритм сортировки вставками должен быть квадратичным, т. е. $O(n^2)$.

Такой грубый анализ методом округления всегда оказывается результативным, в том смысле, что полученная верхняя граница времени исполнения (O-большое) всегда будет правильной. Иногда этот результат может быть даже завышен в худшую сторону, т. е. в действительности время исполнения худшего случая окажется меньшим, чем результат, полученный при анализе. Тем не менее, я настоятельно рекомендую этот подход в качестве основы для простого анализа алгоритмов.

2.5.3. Сравнение строк

Сравнение комбинаций символов — основная операция при работе с текстовыми строками. Далее приводится алгоритм для реализации функции поиска определенного текста, которая является обязательной частью любого веб-браузера или текстового редактора.

ЗАДАЧА. Найти подстроку в строке.

Вход. Текстовая строка t и строка для поиска p (рис. 2.5).

Выход. Содержит ли строка t подстроку p, и, если содержит, в каком месте?

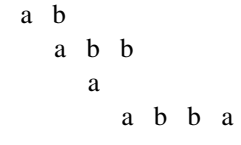


Рис. 2.5. Пример: поиск подстроки *abba* в тексте *aababba*

Пример практического применения этого алгоритма — поиск упоминания определенной фамилии в новостях. Для данного экземпляра задачи текстом t будет статья, а строкой p для поиска — указанная фамилия.

Эта задача решается с помощью довольно простого алгоритма (листинг 2.3), который допускает, что строка p может начинаться в любой возможной позиции в тексте t, и выполняет проверку, действительно ли, начиная с этой позиции, текст содержит искомую строку.

Листинг 2.3. Реализация алгоритма алгоритма поиска строки в тексте

```
int findmatch(char *p, char *t)
{
    int i,j;                    /* Счетчики */
    int m, n;                   /* Длины строк */
    m = strlen(p);
    n = strlen(t);
```

```
    for (i=0; i<=(n-m); i=i+1) {
        j=0;
        while ((j<m) && (t[i+j]==p[j]))
            j = j+1;
        if (j == m) return(i);
    }
    return(-1);
}
```

Каким будет время исполнения этих двух вложенных циклов в наихудшем случае? Внутренний цикл `while` исполняется максимум m раз, а возможно, и намного меньше, если поиск заканчивается неудачей. Кроме оператора `while`, внешний цикл содержит еще два оператора. Внешний цикл исполняется самое большее $n - m$ раз, т. к. после продвижения направо по тексту оставшийся фрагмент будет короче искомой строки. Общая временная сложность — произведение значений оценки временной сложности внешнего и вложенного циклов, что дает нам время исполнения в худшем случае $O((n - m)(m + 2))$.

При этом мы не учитываем время, потраченное на определение длины строк с помощью функции `strlen`. Так как мы не знаем, каким образом реализована эта функция, мы можем лишь строить предположения относительно времени ее работы. Если мы явно считаем количество символов, пока не достигнем конца строки, то отношение между временем исполнения этой операции и длиной строки будет линейным. Значит, время работы будет равно $O(n + m + (n - m)(m + 2))$.

С помощью асимптотических обозначений это выражение можно упростить. Так как $m + 2 = \Theta(m)$, выражение " + 2" не представляет интереса, поэтому останется только $O(n + m + (n - m)m)$. Выполнив умножение, получаем выражение $O(n + m + nm - m^2)$, которое выглядит довольно непривлекательно.

Но мы знаем, что в любой представляющей интерес задаче $n \geq m$, т. к. невозможно, чтобы искомая строка p была длиннее, чем текст t, в котором выполняется ее поиск. Одним из следствий этого обстоятельства является отношение $(n + m) \leq 2n = \Theta(n)$. Таким образом, формула времени исполнения для наихудшего случая упрощается дальше до $O(n + nm - m^2)$.

Еще два замечания. Обратите внимание, что $n \leq nm$, т. к. для любой представляющей интерес строки поиска $m \geq 1$. Таким образом, $n + nm = \Theta(nm)$, и мы можем опустить дополняющее n, упростив формулу анализа до $O(nm - m^2)$.

Кроме того, заметьте, что член $-m^2$ отрицательный, вследствие чего он только уменьшает значение выражения внутри скобок. Так как "O-большое" задает верхнюю границу, то любой отрицательный член можно удалить, не искажая оценку верхней границы. Тот факт, что $n \geq m$ подразумевает, что $mn \geq m^2$, поэтому отрицательный член недостаточно большой, чтобы аннулировать любой другой оставшийся член. Таким образом, время исполнения этого алгоритма в худшем случае можно выразить просто как $O(nm)$.

Накопив достаточно опыта, вы сможете выполнять такой анализ алгоритмов в уме, даже не прибегая к изложению алгоритма в письменной форме. В конце концов, одной из составляющих разработки алгоритма для решения предоставленной задачи является перебор в уме разных способов и выбор самого лучшего из них. Такое умение приоб-

ретается с опытом, но если у вас недостаточно практики и вы не понимаете, почему время исполнения данного алгоритма равно $O(f(n))$, то сначала распишите подробно формулу, а потом выполните последовательность логических рассуждений, как было продемонстрировано в этом разделе.

2.5.4. Умножение матриц

При анализе алгоритмов с вложенными циклами часто приходится иметь дело с вложенными операциями суммирования. Рассмотрим задачу умножения матриц:

ЗАДАЧА. Умножить матрицы.

Вход. Матрица A (размером $x \times y$) и матрица B (размером $y \times z$).

Выход. Матрица C размером $x \times z$, где $C[i][j]$ является скалярным произведением строки i матрицы A и столбца j матрицы B.

Умножение матриц является одной из основных операций в линейной алгебре; пример задачи на умножение матриц рассматривается в *разделе 13.3*. А в листинге 2.4 приводится пример реализации простого алгоритма для умножения матриц с использованием вложенных циклов.

Листинг 2.4. Умножение матриц

```
for (i=1; i<=x; i++)
     for (j=1; j<=y; j++) {
          C[i][j] = 0;
          for (k=1; k<=z; k++)
               C[i][j] += A[i][k] * B[k][j];
     }
```

Анализ временной сложности этого алгоритма выполняется следующим образом. Количество операций умножения $M(x, y, z)$ определяется такой формулой:

$$M(x,y,z) = \sum_{i=1}^{x} \sum_{j=1}^{y} \sum_{k=1}^{z} 1$$

Суммирование выполняется справа налево. Сумма z единиц равна z, поэтому можно написать:

$$M(x,y,z) = \sum_{i=1}^{x} \sum_{j=1}^{y} z$$

Сумма y членов z вычисляется так же просто. Она равна yz, тогда $M(x,y,z) = \sum_{i=1}^{x} yz$.

Наконец, сумма x членов yz равна xyz.

Таким образом, время исполнения этого алгоритма для умножения матриц равно $O(xyz)$. В общем случае, когда все три измерения матриц одинаковы, это сводится к $O(n^3)$, т. е. алгоритм имеет кубическую формулу времени исполнения.

2.6. Логарифмы и их применение

Слово "логарифм" — почти анаграмма слова "алгоритм". Но наш интерес к логарифмам вызван не этим обстоятельством. Возможно, что кнопка калькулятора с обозначением "log" — единственное место, где вы сталкиваетесь с логарифмами в повседневной жизни. Также возможно, что вы уже не помните назначение этой кнопки. *Логарифм* — это функция, обратная показательной. То есть, выражение $b^x = y$ эквивалентно выражению $x = \log_b y$. Более того, из определения логарифма следует, что $b^{\log_b y} = y$.

Показательные функции возрастают чрезвычайно быстро, как может засвидетельствовать любой, кто когда-либо выплачивал долг по кредиту. Соответственно, функции, обратные показательным, т. е. логарифмы, возрастают довольно медленно. Логарифмические функции возникают в любом процессе, содержащем деление пополам. Давайте рассмотрим несколько таких примеров.

2.6.1. Логарифмы и двоичный поиск

Двоичный поиск является хорошим примером алгоритма с временной логарифмической сложностью $O(\log n)$. Чтобы найти определенного человека по имени p в телефонной книге, содержащей n имен, мы сравниваем имя p с выбранным именем посередине книги (т. е. с $n/2$-м именем), скажем, Monroe, Marilyn. Независимо от того, находится ли имя p перед выбранным именем (например, Dean, James) или после него (например, Presley, Elvis), после этого сравнения мы можем отбросить половину всех имен в книге. Этот процесс повторяется с половиной книги, содержащей искомое имя и т. д., пока не останется всего лишь одно имя, которое и будет искомым. По определению количество таких делений равно $\log_2 n$. Таким образом, чтобы найти любое имя в телефонной книге Манхэттена (содержащей миллион имен), достаточно выполнить всего лишь двадцать сравнений. Потрясающе, не так ли?

Идея двоичного поиска является одной из наиболее плодотворных в области разработки алгоритмов. Эта мощь становится очевидной, если мы представим, что в окружающем нас мире имеются только неотсортированные телефонные книги. Как видно из табл. 1.1, алгоритмы с временной сложностью $O(\log n)$ можно применять для решения задач с практически неограниченным размером входных данных.

2.6.2. Логарифмы и деревья

Двоичное дерево высотой в один уровень может иметь две концевые вершины (листа), а дерево высотой в два уровня может иметь до четырех листов. Какова высота h двоичного дерева, имеющего n листов? Обратите внимание, что количество листов удваивается при каждом увеличении высоты дерева на один уровень. Таким образом, зависимость количества листов n от высоты дерева h выражается формулой $n = 2^h$, откуда следует, что $h = \log_2 n$.

Теперь перейдем к общему случаю. Рассмотрим деревья, которые имеют d потомков (для двоичных деревьев $d = 2$). Такое дерево высотой в один уровень может иметь d количество листов, а дерево высотой в два уровня может иметь d^2 количество листов. Количество листов на каждом новом уровне можно получить, умножая на d количество

листов предыдущего уровня. Таким образом, количество листов n выражается формулой $n = d^h$, т. е. высота находится по формуле $h = \log_d n$ (рис. 2.6).

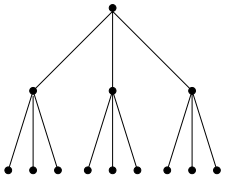


Рис. 2.6. Дерево высотой h и количеством потомков d для каждого узла имеет d^h листьев.

В данном случае $h = 2$, $d = 3$

Из вышеизложенного можно сделать вывод, что деревья небольшой высоты могут иметь очень много листьев. Это обстоятельство является причиной того, что двоичные деревья лежат в основе всех быстро обрабатываемых структур данных.

2.6.3. Логарифмы и биты

Положим, имеются две однобитовые комбинации (0 и 1) и четыре двухбитовые комбинации (00, 01, 10 и 11). Сколько битов w потребуется, чтобы представить любую из n возможных разных комбинаций, будь то один из n элементов или одно из целых чисел от 1 до n?

Ключевым наблюдением здесь является то обстоятельство, что нужно иметь, по крайней мере, n разных битовых комбинаций длиной w. Так как количество разных битовых комбинаций удваивается с добавлением каждого бита, то нам нужно, по крайней мере, w битов, где $2^w = n$, т. е. нам нужно $w = \log_2 n$ битов.

2.6.4. Логарифмы и умножение

Логарифмы имели особенно большую важность до распространения карманных калькуляторов. Применение логарифмов было самым легким способом умножения больших чисел вручную, либо с помощью логарифмической линейки, либо с использованием таблиц.

Но и сегодня логарифмы остаются полезными для выполнения операций умножения, особенно для возведения в степень. Вспомните, что $\log_a(xy) = \log_a(x) + \log_a(y)$, т. е., что логарифм произведения равен сумме логарифмов сомножителей. Прямым следствием этого является формула:

$$\log_a n^b = b \cdot \log_a n$$

Выясним, как вычислить a^b для любых a и b, используя функции $\exp(x)$ и $\ln(x)$ на карманном калькуляторе, где $\exp(x) = e^x$ и $\ln(x) = \log_e(x)$. Мы знаем, что:

$$a^b = \exp(\ln(a^b)) = \exp(b \ln a)$$

Таким образом, задача сводится к одной операции умножения с однократным вызовом каждой из этих функций.

2.6.5. Быстрое возведение в степень

Допустим, что нам нужно вычислить *точное* значение a^n для достаточно большого значения n. Такие задачи, в основном, возникают в криптографии при проверке числа на простоту (см. *раздел 13.8*). Проблемы с точностью не позволяют нам воспользоваться ранее рассмотренной формулой возведения в степень.

Самый простой алгоритм выполняет $n-1$ операций умножения ($a \times a \times ... \times a$). Но можно указать лучший способ решения этой задачи, приняв во внимание, что $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$. Если n четное, тогда $a^n = (a^{n/2})^2$. А если n нечетное, то тогда $a^n = a(a^{\lfloor n/2 \rfloor})^2$. В любом случае значение показателя степени было уменьшено наполовину, а вычисление сведено к, самое большее, двум операциям умножения. Таким образом, для вычисления конечного значения будет достаточно $O(\lg n)$ операций умножения. Псевдокод соответствующего алгоритма показан в листинге 2.5.

Листинг 2.5. Алгоритм быстрого возведения в степень

```
function power(a, n)
     if (n = 0) return(1)
     x = power (a, ⌊n/2⌋)
     if (n is even) then return(x²)
          else return(a × x²)
```

Этот простой алгоритм иллюстрирует важный принцип "разделяй и властвуй". Разделение задачи на (по возможности) равные подзадачи, всегда окупается. Этот принцип применим и в реальном мире. Когда значение n отлично от 2, то входные данные не всегда можно разделить точно пополам, но разница в один элемент между двумя половинами не вызовет никакого серьезного нарушения баланса.

2.6.6. Логарифмы и сложение

Гармоническое число представляет собой особый случай арифметической прогрессии, а именно $H(n) = S(n, -1)$. Это сумма обратных величин первых n последовательных чисел натурального ряда:

$$H(n) = \sum_{i=1}^{n} 1/i \sim \ln n$$

Гармонические числа помогают объяснить, откуда берутся логарифмы в алгебраических операциях. Например, ключевым фактором в анализе среднего случая сложности алгоритма быстрой сортировки Quicksort является следующее суммирование:

$$S(n) = n\sum_{i=1}^{n} 1/i$$

Применение тождества гармонического числа сразу же упрощает это выражение до $\Theta(n \log n)$.

2.6.7. Логарифмы и система уголовного судопроизводства

В табл. 2.2 приводится пример использования логарифмов.

Таблица 2.2. Рекомендуемые наказания в федеральных судах США за преступления финансового мошенничества

Понесенные убытки	Повышение уровня наказания
(A) 2 000 долларов или меньше	Уровень не повышается
(B) Свыше 2 000 долларов	Повысить на один уровень
(C) Свыше 5 000 долларов	Повысить на два уровня
(D) Свыше 10 000 долларов	Повысить на три уровня
(E) Свыше 20 000 долларов	Повысить на четыре уровня
(F) Свыше 40 000 долларов	Повысить на пять уровней
(G) Свыше 70 000 долларов	Повысить на шесть уровней
(H) Свыше 120 000 долларов	Повысить на семь уровней
(I) Свыше 200 000 долларов	Повысить на восемь уровней
(J) Свыше 350 000 долларов	Повысить на девять уровней
(K) Свыше 500 000 долларов	Повысить на десять уровней
(L) Свыше 800 000 долларов	Повысить на одиннадцать уровней
(M) Свыше 1 500 000 долларов	Повысить на двенадцать уровней
(N) Свыше 2 500 000 долларов	Повысить на тринадцать уровней
(O) Свыше 5 000 000 долларов	Повысить на четырнадцать уровней
(P) Свыше 10 000 000 долларов	Повысить на пятнадцать уровней
(Q) Свыше 20 000 000 долларов	Повысить на шестнадцать уровней
(R) Свыше 40 000 000 долларов	Повысить на семнадцать уровней
(S) Свыше 5 000 000 долларов	Повысить на восемнадцать уровней

Эта таблица из Федерального руководства по вынесению наказаний используется во всех федеральных судах Соединенных Штатов. Изложенные в ней рекомендуемые уровни наказания являются попыткой стандартизировать выносимые приговоры, чтобы за преступления одинаковой категории разные суды приговаривали к одинаковому наказанию. Для этого специальная комиссия выработала сложную функцию для оценки тяжести преступления и соотнесения его со сроком заключения. Данная функция представлена посредством табл. 2.2, в которой перечислены соотношения между понесенным убытком в долларах и соответствующим повышением базового наказания. Обратите внимание, что наказание повышается на один уровень при приблизительном удвоении суммы незаконно присвоенных денег. Это означает, что уровень наказания (который практически линейно связан со сроком заключения) возрастает логарифмически по отношению к сумме незаконно присвоенных денег.

Задумайтесь на минуту о последствиях этого обстоятельства. Несомненно, многие нечистые на руку руководители фирм уже задумывались на эту тему. Все вышеизложенное означает, что общий срок заключения возрастает *чрезвычайно* медленно по отношению к росту суммы украденных денег. Срок заключения за пять ограблений винно-водочных магазинов на общую сумму 50 000 долларов будет значительно выше, чем за однократное завладение посредством мошеннических операций суммой в 1 000 000 долларов. Соответственно, выгода от обогащения подобным образом на действительно крупные суммы будет еще больше. Мораль логарифмического роста функций ясна: уж если воровать, так миллионы.

Подведение итогов

Логарифмические функции возникают при решении задач с повторяющимся делением или удваиванием входных данных.

2.7. Свойства логарифмов

Как мы уже видели, выражение $b^x = y$ эквивалентно выражению $x = \log_b y$. Член b называется *основанием* логарифма. Особый интерес представляют следующие основания логарифмов:

- **основание b = 2.** *Двоичный логарифм*, обычно обозначаемый как $\lg x$, является логарифмом по основанию 2. Мы уже видели, что логарифмами с этим основанием выражается временная сложность алгоритмов, использующих многократное деление пополам (т. е. двоичный поиск) или умножение на два (т. е. листья деревьев). В большинстве случаев, когда речь идет о применении логарифмов в алгоритмах, подразумеваются двоичные логарифмы;
- **основание b = e.** *Натуральный логарифм*, обычно обозначаемый как $\ln x$, является логарифмом по основанию $e = 2.71828...$ Обратной к функции натурального логарифма является экспоненциальная функция $\exp(x) = e^x$. Суперпозиция этих функций дает нам формулу $\exp(\ln x) = x$;
- **основание b = 10.** Менее распространенными на сегодняшний день являются логарифмы по основанию 10, или *десятичные логарифмы*. До появления карманных калькуляторов логарифмы с этим основанием применялись на логарифмических линейках и таблицах алгоритмов.

Мы уже видели одно важное свойство логарифмов, а именно, что $\log_a(xy) = \log_a(x) + \log_a(y)$.

Другим важным фактом, который нужно запомнить, является то, что логарифм по одному основанию легко преобразовать в логарифм по другому основанию. Для этого применяется следующая формула:

$$\log_a b = \frac{\log_c b}{\log_c a}$$

Таким образом, чтобы изменить основание a логарифма $\log_a b$ на c, логарифм просто нужно разделить на $\log_c a$. В частности, функцию натурального логарифма можно с легкостью преобразовать в функцию десятичного логарифма и наоборот.

Из этих свойств логарифмов следуют два важных с арифметической точки зрения следствия.

- *Основание логарифма не оказывает значительного влияния на скорость роста функции.* Сравните следующие три значения: $\log_2(1\,000\,000) = 19.9316$, $\log_3(1\,000\,000) = 12.5754$ и $\log_{100}(1\,000\,000) = 3$. Как видите, большое изменение в основании логарифма сопровождается малыми изменениями в значении логарифма. Чтобы изменить у логарифма основание a на c, первоначальный логарифм нужно разделить на $\log_c a$. Этот коэффициент теряется в нотации "O-большое", когда a и c являются константами. Таким образом, игнорирование основания логарифма при анализе алгоритма обычно оправдано.
- *Логарифмы уменьшают значение любой функции.* Скорость роста логарифма любой полиномиальной функции определяется как $O(\lg n)$. Это вытекает из равенства:

$$\log_a n^b = b \cdot \log_a n$$

Эффективность двоичного поиска в широком диапазоне задач является прямым следствием этого свойства. Обратите внимание, что двоичный поиск в отсортированном массиве из n^2 элементов требует всего лишь вдвое больше сравнений, чем в массиве из n элементов.

Логарифмы уменьшают значение любой функции. С факториалами трудно выполнять какие-либо вычисления, если не пользоваться логарифмами, и тогда формула

$$n! = \prod_{i=1}^{n} i \rightarrow \log n! = \sum_{i=1}^{n} \log i = \Theta(n \log n)$$

становится еще одной причиной появления логарифмов в анализе алгоритмов.

Остановка для размышлений. Важно ли деление точно пополам

ЗАДАЧА. Насколько больше операций сравнения потребуется при двоичном поиске, чтобы найти совпадение в списке из миллиона элементов, если вместо деления точно пополам делить список в отношении 1/3 к 2/3?

Решение. Не намного больше, чем при делении списка точно пополам, а именно $\log_{3/2}(1\,000\,000)$ 35 операций сравнения в самом худшем случае, что не намного больше, чем $\log_2(1\,000\,000)$ 20 операций сравнения при делении пополам. Эффективность алгоритма двоичного поиска определяется логарифмической временной сложностью, а не основанием логарифма. ■

2.8. История из жизни. Загадка пирамид

По выражению его глаз я должен был догадаться, что услышу что-то необычное.

— Мы хотим выполнять вычисления до 1 000 000 000 на многопроцессорном суперкомпьютере, но для этого нам нужен более быстрый алгоритм.

Мне приходилось видеть такой взгляд раньше. Глаза были пусты от того, что этот человек уже давно ни о чем не думал, полагая, что мощные суперкомпьютеры избавляют его от необходимости разрабатывать сложные алгоритмы. Во всяком случае, это было так, пока задача не стала достаточно сложной.

— Я работаю с лауреатом Нобелевской премии над решением знаменитой задачи теории чисел с помощью компьютера. Вы знакомы с проблемой Уоринга?

Я имел некоторое представление о теории чисел.

— Конечно. В проблеме Уоринга ставится вопрос, можно ли выразить каждое целое число, по крайней мере, одним способом, как сумму квадратов, самое большее, четырех целых чисел. Например, $78 = 8^2 + 3^2 + 2^2 + 1^2 = 7^2 + 5^2 + 2^2$. Я помню, как в институтском курсе теории чисел мне приходилось доказывать, что для выражения любого целого числа будет достаточно квадратов четырех целых чисел. Да, это знаменитая задача, но она была решена около 200 лет тому назад.

— Нет, нас интересует другая версия проблемы Уоринга, с использованием пирамидальных чисел. Пирамидальным называется число, которое можно представить в виде $(m^3 - m)/6$ при $m \geq 2$. Примером нескольких первых пирамидальных чисел будут 1, 4, 10, 20, 35, 56, 84, 120, 165. С 1928 года существует гипотеза, что любое целое число можно выразить суммой, самое большее, пяти пирамидальных чисел. Мы хотим с помощью суперкомпьютера доказать эту гипотезу для всех чисел от 1 до 1 000 000 000.

— Миллиард любых вычислений займет значительное время, — предупредил я. — Критичным будет время, затраченное на вычисление минимального представления каждого числа, т. к. это придется делать миллиард раз. Вы думали, какой тип алгоритма использовать?

— Мы уже написали свою программу и испытали ее на многопроцессорном суперкомпьютере. На небольших числах она работает очень быстро, но при работе с числами, большими 100 000, время значительно увеличивается.

Все ясно, — подумал я. Этот "компьютероман" открыл асимптотический рост. Никаких сомнений, что он использовал алгоритм квадратичной временной сложности и столкнулся с проблемами, как только число n стало достаточно большим.

— Нам нужна более быстрая программа, чтобы дойти до одного миллиарда. Можете ли вы помочь нам с этой задачей? Конечно же, мы будем выполнять ее на нашем многопроцессорном суперкомпьютере.

Поскольку я люблю такой тип задач — разработку алгоритмов для ускорения времени работы программ — я согласился подумать над этим и принялся за работу.

Я начал с просмотра программы, написанной моим посетителем. Он создал массив всех $\Theta(n^{1/3})$ пирамидальных чисел от 1 до n включительно[1]. Для каждого числа k в этом диапазоне методом полного перебора выполнялась проверка, являлось ли оно суммой двух пирамидальных чисел. При отрицательном результате проверки выполнялась проверка, было ли число суммой трех пирамидальных чисел, потом четырех и, наконец, пяти, пока не находился ответ. Приблизительно 45% целых чисел можно выразить как сумму трех пирамидальных чисел. Большинство из оставшихся 55% чисел можно представить в виде суммы четырех пирамидальных чисел и, как правило,

---

[1] Почему $n^{1/3}$? Вспомните, что пирамидальные числа выражаются формулой $(m^3 - m)/6$. Наибольшее число m, такое что результат не превышает n, приблизительно равно $\sqrt[3]{6n}$; поэтому количество таких чисел выражается формулой $\Theta(n^{1/3})$.

разными способами. Известно только 241 целое число, представляемое суммой пяти пирамидальных чисел, самое большое из которых равно 343 867. Для примерно половины из n чисел этот алгоритм проверял все трехэлементные комбинации и, по крайней мере, некоторые четырехэлементные. Таким образом, общее время исполнения этого алгоритма было, как минимум, $O(n \times (n^{1/3})^3) = O(n^2)$, где $n = 1\ 000\ 000\ 000$. Неудивительно, что на больших числах (превышающих 100 000) эта программа замедляла работу.

Любое решение, работающее на больших числах значительно быстрее предложенного, должно избегать явной проверки всех трехэлементных комбинаций. Для каждого значения k нам нужно наименьшее множество пирамидальных чисел, сумма которых в точности равна k. Эта задача называется задачей о рюкзаке и рассматривается в *разделе 13.10*. В нашем случае весу предметов соответствует набор пирамидальных чисел, не превышающих n, с тем дополнительным ограничением, что рюкзак вмещает ровно k предметов (т. е. чисел, в нашем случае).

В стандартном подходе к решению задачи о рюкзаке заранее вычисляются суммы меньших подмножеств, которые потом используются в вычислении больших подмножеств. Если у нас имеется таблица, содержащая суммы двух чисел, и мы хотим узнать, можно ли выразить число k в виде суммы трех чисел, мы можем перефразировать постановку задачи и спросить, можно ли выразить число k в виде суммы одного числа и одной из сумм в данной таблице.

Поэтому мне нужно было составить таблицу всех целых чисел, меньших, чем n, которые можно выразить в виде суммы двух из 1 818 пирамидальных чисел, меньших чем 1 000 000 000. Таких чисел может быть самое большее $1\ 818^2 = 3\ 305\ 124$. Более того, если мы уберем повторяющиеся суммы и суммы, большие, чем целевое число, у нас останется меньше чем половина чисел. Создание отсортированного массива этих чисел не составит никакого труда. Назовем эту отсортированную структуру данных таблицей сумм двух пирамидальных чисел.

Поиск минимального разложения данного числа k начинается с проверки, является ли оно одним из 1 818 пирамидальных чисел. Если не является, то тогда выполняется проверка, не находится ли оно в таблице сумм двух пирамидальных чисел. Чтобы проверить, можно ли выразить число k в виде суммы трех пирамидальных чисел, нужно было всего лишь проверить, что $k - p[i]$ находится в таблице сумм двух пирамидальных чисел при $1 \le i \le 1\ 818$. Эту проверку можно было быстро выполнить посредством двоичного поиска. Чтобы проверить, можно ли выразить число k в виде суммы четырех пирамидальных чисел, нужно было всего лишь проверить, что $k - two[i]$ находится в таблице сумм двух пирамидальных чисел для любого $1 \le i \le |two|$. Но так как почти любое число k можно выразить как сумму многих комбинаций четырех пирамидальных чисел, эта проверка не займет много времени, и доминирующей составляющей времени общей проверки будет время, затраченное на проверку трехэлементных сумм. Временная сложность проверки, является ли число k суммой трех пирамидальных чисел, оценивается как $O(n^{1/3}\lg n)$, а для всего множества n целых чисел — как $O(n^{4/3}\lg n)$. По сравнению с временной сложностью $O(n^2)$ алгоритма клиента для $n = 1\ 000\ 000\ 000$, мой алгоритм был в 30 000 раз быстрее.

Первый прогон реализации этого алгоритма на моем далеко не новом компьютере PARC ELC занял около 20 минут для $n = 1\ 000\ 000$. После этого я экспериментировал с представлением наборов чисел разными структурами данных и с разными алгоритмами

для поиска в этих структурах. В частности, я попробовал использовать вместо отсортированных массивов хэш-таблицы и двоичные векторы, поэкспериментировал с разновидностями двоичного поиска, такими как интерполяционный поиск (см. *раздел 14.2*). Наградой за эту работу стало менее чем трехминутное время исполнения программы на множестве чисел $n = 1\,000\,000$, что в шесть раз лучше времени исполнения первоначальной программы.

Завершив основную работу, я занялся настройкой программы, чтобы немного повысить производительность. В частности, т. к. 1 — это пирамидальное число, то для любого числа k, когда $k - 1$ было суммой трех пирамидальных чисел, я не вычислял сумму четырех пирамидальных чисел. В результате использования только этого приема общее время исполнения программы было сокращено еще на 10%. Наконец, с помощью профилировщика я выполнил несколько низкоуровневых настроек, чтобы еще чуть-чуть повысить производительность. Например, заменив лишь одну вызываемую процедуру встроенным кодом, я сбросил еще 10% с времени исполнения.

После этого я передал программу заказчику. Он использовал ее самым неподходящим образом, о чем я расскажу в *разделе 7.10*.

Подготавливая эту историю из жизни более чем десять лет спустя, я достал свою программу из архивов и запустил ее на моем теперешнем настольном компьютере, SunBlade 150. Скомпилированная с помощью компилятора gcc без какой бы то ни было оптимизации программа обработала 1 000 000 чисел за 27 секунд. А время исполнения программы, скомпилированной с четвертым уровнем оптимизации, составило всего лишь 14 секунд, что заставляет отдать должное качеству оптимизатора. Время исполнения на моем настольном компьютере улучшилось приблизительно в три раза за четыре года ко времени публикации первого издания этой книги и еще в 5,3 раза за последние 11 лет. Такое улучшение производительности типично для большинства настольных компьютеров.

Основная цель моего рассказа состоит в том, чтобы показать громадный потенциал повышения скорости вычислений за счет улучшения эффективности алгоритмов, по сравнению с довольно скромным повышением производительности, получаемым за счет установки более дорогого оборудования. Применив более эффективный алгоритм, я повысил скорость вычислений приблизительно в 30 тысяч раз. Суперкомпьютер моего заказчика, стоивший миллион долларов, был оснащен 16 процессорами, каждый из которых мог выполнять целочисленные вычисления в пять раз быстрее, чем мой настольный компьютер стоимостью в 3 000 долларов. Но при использовании всей этой техники скорость выполнения моей программы возросла менее чем в 100 раз. Очевидно, что в данном случае применение более эффективного алгоритма имеет преимущество над использованием более мощного оборудования, что справедливо для любой задачи с достаточно большим вводом.

2.9. Анализ высшего уровня (*)

В идеальном случае мы все умели бы свободно обращаться с математическими методами асимптотического анализа. Точно так же, в идеале, мы все были бы богатыми и красивыми. А поскольку жизнь далека от идеала, математические методы асимптотического анализа требуют определенных знаний и практики.

В этом разделе выполняется обзор основных методов и функций, применяемых в анализе алгоритмов на высшем уровне. Это факультативный материал — он не используется нигде в первой части этой книги. В то же самое время, знание этого материала будет большим подспорьем в понимании некоторых функций временной сложности, рассматриваемых во второй части.

2.9.1. Малораспространенные функции

Основные классы функций временной сложности алгоритмов были представлены в *разделе 2.3.1*. Но в расширенном анализе алгоритмов также возникают и менее распространенные функции. И хотя такие функции нечасто встречаются в этой книге, вам будет полезно знать, что они означают и откуда происходят. Далее приводятся некоторые из таких функций и их краткое описание.

- *Обратная функция Аккермана* $f(n) = \alpha(n)$. Эта функция появляется в подробном анализе нескольких алгоритмов.

  В данной книге точное определение этой функции и причины ее возникновения не рассматриваются. Будет достаточно воспринимать эту функцию как технический термин для самой медленно возрастающей функции сложности алгоритмов. В отличие от функции-константы $f(n) = 1$, эта функция достигает бесконечности при $n \to \infty$, но она определенно не торопится сделать это. Для любого значения n в физической вселенной значение функции будет меньше 5, т. е. $\alpha(n) < 5$.

- $f(n) = \log\log n$. Смысл функции "log log" очевиден по ее имени — это логарифм логарифма числа n. Одним из естественных примеров ее возникновения будет двоичный поиск в отсортированном массиве из всего лишь $\lg n$ элементов.

- $f(n) = \log n / \log\log n$. Эта функция возрастает немного медленнее, чем функция $\log n$, т. к. она содержит в знаменателе еще более медленно растущую функцию.

  Чтобы понять, как возникла эта функция, рассмотрим корневое дерево с количеством потомков d, имеющее n листьев. Высота двоичных корневых деревьев, т. е. деревьев, у которых $d = 2$, определяется следующей формулой:

  $$n = 2^h \to h = \lg n$$

  получающейся в результате логарифмирования обеих частей равенства. Теперь рассмотрим высоту дерева, когда количество потомков равно $d = \log n$. Тогда высота определяется такой формулой: $n = (\log n) \to h = \log n / \log\log n$.

- $f(n) = \log^2 n$. Это произведение логарифмических функций, т. е. $(\log n) \times (\log n)$. Такая функция может возникнуть, если мы хотим сосчитать просмотренные биты в процессе двоичного поиска в множестве из n элементов, каждый из которых является целым числом в диапазоне от 1 до, например, n. Для представления каждого из этих целых чисел требуется $\lg(n^2) = 2\lg n$ бит, а поскольку количество чисел в множестве поиска равно $\lg n$, то общее количество бит будет равно $2\lg^2 n$.

  Функция "логарифм в квадрате" обычно возникает при разработке сложных гнездовых структур, где каждый узел в, скажем, двоичном дереве представляет другую структуру данных, возможно, упорядоченную по другому ключу.

- $f(n) = \sqrt{n}$. Функция квадратного корня встречается не так уж редко, но представляет класс "сублинейных полиномов", т. к. $\sqrt{n} = n^{1/2}$. Такие функции возникают при построении d-мерных сеток, содержащих n точек. Площадь квадрата размером $\sqrt{n} \times \sqrt{n}$ равна n, а объем куба размером $n^{1/3} \times n^{1/3} \times n^{1/3}$ также составляет n. В общем, объем d-мерного гиперкуба со стороной $n^{1/d}$ составляет n.
- $f(n) = n^{(1+\varepsilon)}$. Греческая буква ε обозначает константу, которая может быть сколь угодно малой, но при этом не равняется нулю.

  Она может возникнуть в таких обстоятельствах. Допустим, время исполнения алгоритма равно $2^c n^{(1 + 1/c)}$ и мы можем выбирать любое значение для c. При $c = 2$ время исполнения будет $4n^{3/2}$ или $O(n^{3/2})$. При $c = 3$ время исполнения будет $8n^{4/3}$ или $O(n^{4/3})$, что уже лучше. Действительно, чем больше значение c, тем лучше становится показатель степени.

  Но c нельзя сделать как угодно большим, до того как член 2^c станет доминировать. Вместо этого мы обозначаем время исполнения этого алгоритма как $O(n^{1+\varepsilon})$ и предоставляем пользователю определить наилучшее значение для ε.

2.9.2. Пределы и отношения доминирования

Отношения доминирования между функциями являются следствием теории пределов, изучаемой в курсе высшей математики. Говорят, что функция $f(n)$ доминирует над функцией $g(n)$, если $\lim_{n\to\infty} g(n)/f(n) = 0$.

Рассмотрим это определение в действии. Допустим, что $f(n) = 2n^2$ и $g(n) = n^2$. Очевидно, что $f(n) > g(n)$ для всех n, но не доминирует над ней, т. к.

$$\lim_{n\to\infty} g(n) / f(n) = \lim_{n\to\infty} n^2 / 2n^2 = \lim_{n\to\infty} 1/2 \neq 0$$

Этого следовало ожидать, т. к. обе функции принадлежат к одному и тому же классу $\Theta(n^2)$. Теперь рассмотрим функции $f(n) = n^3$ и $g(n) = n^2$. Так как

$$\lim_{n\to\infty} g(n) / f(n) = \lim_{n\to\infty} n^2 / n^3 = \lim_{n\to\infty} 1/n = 0$$

то доминирует многочлен более высокого уровня. Это справедливо для любых двух многочленов, а именно n^a доминирует над n^b, если $a > b$, т. к.

$$\lim_{n\to\infty} n^b / n^a = \lim_{n\to\infty} n^{b-a} \to 0$$

Таким образом, $n^{1,2}$ доминирует над $n^{1,1999999}$.

Перейдем к показательным функциям: $f(n) = 3^n$ и $g(n) = 2^n$. Так как

$$\lim_{n\to\infty} g(n) / f(n) = 2^n / 3^n = \lim_{n\to\infty} (2/3)^n = 0$$

значит, доминирует функция с большим основанием.

Возможность доказать отношение доминирования зависит от возможности доказать пределы. Давайте рассмотрим одну важную пару функций. Любой многочлен (скажем,

$f(n) = n^{\varepsilon}$) доминирует над логарифмическими функциями (например, $g(n) = \lg n$). Так как $n = 2^{\lg n}$, то $f(n) = (2^{\lg n})^{\varepsilon} = 2^{\varepsilon \lg n}$. Теперь рассмотрим следующее тождество:

$$\lim_{n \to \infty} g(n) / f(n) = \lg n / 2^{\varepsilon \lg n}$$

В действительности при $n \to \infty$ оно стремится к нулю.

Подведение итогов

Рассматривая совместно функции, приведенные в этом разделе, и функции, обсуждаемые в *разделе 2.3.1*, мы видим полную картину порядка доминирования функций:

$$n! \gg c^n \gg n^3 \gg n^2 \gg n^{1+\varepsilon} \gg n \log n \gg n \gg \sqrt{n} \gg \log^2 n \gg \log n \gg \log n / \log\log n \gg$$
$$\gg \log\log n \gg \alpha(n) \gg 1$$

Замечания к главе

В других работах по алгоритмам уделяется значительно больше внимания формальному анализу алгоритмов, чем в этой книге, поэтому читателей со склонностью к теоретическим выкладкам я отсылаю к другим изданиям. В частности, анализ алгоритмов рассматривается на более глубоком уровне в книгах [CLRS01] и [KT06].

В книге [GKP89] дается интересное и всестороннее изложение математического аппарата для анализа алгоритмов. Хорошее введение в теорию чисел, включая проблему Уоринга, упомянутую в *разделе 2.8*, дается в книге [NZ80].

Понятие доминирования также порождает обозначение *о-малое*. Говорят, что $f(n) = o(g(n))$ тогда и только тогда, когда $g(n)$ доминирует над $f(n)$. Среди прочего, обозначение *о*-малое полезно для формулировки задач. Требование представить алгоритм с временной сложностью $o(n^2)$ означает, что нам нужен алгоритм с функцией временной сложности лучшей, чем квадратичная, и что будет приемлемой временная сложность, выражаемая функцией $O(n^{1,999}\log^2 n)$.

2.10. Упражнения

Анализ программ

1. [3] Какое значение возвращает следующая функция? Ответ должен быть в форме функции числа n. Найдите время исполнения в наихудшем случае, используя обозначение O-большое.

```
function mystery(n)
  r:=0
  for i := 1 to n − 1 do
    for j := i + 1 to n do
      for k := 1 to j do
        r := r + 1
  return(r)
```

2. [3] Какое значение возвращает следующая функция? Ответ должен быть в форме функции числа n. Найдите время исполнения в наихудшем случае, используя обозначение O-большое.

```
function pesky(n)
 r:=0
 for i := 1 to n do
   for j := 1 to i do
     for k :=j to i + j do
       r := r + 1
 return(r)
```

3. [5] Какое значение возвращает следующая функция? Ответ должен быть в форме функции числа *n*. Найдите время исполнения в наихудшем случае, используя обозначение *O*-большое.

```
function prestiferous(n)
 r :=0
 for i := 1 to n do
   for j := 1 to i do
     for k := j to i + j do
       for l := 1 to i + j − k do
         r := r + 1
 return(r)
```

4. [8] Какое значение возвращает следующая функция? Ответ должен быть в форме функции числа *n*. Найдите время исполнения в наихудшем случае, используя обозначение *O*-большое.

```
function conundrum(n)
r:=0
for i := 1 to n do
  for i := i + 1 to n do
    for k := i + j − 1 to n do
      r := r + 1
return(r)
```

5. [5] Допустим, что для вычисления многочлена $p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ используется следующий алгоритм:

```
p:=a0;
xpower := 1;
for i := 1 to n do
 xpower := x * xpower;
 p := p + ai * xpower
end
```

- Сколько операций умножения выполняется в наихудшем случае? А сколько операций сложения?
- Сколько операций умножения выполняется в среднем?
- Можно ли улучшить этот алгоритм?

6. [3] Докажите правильность следующего алгоритма для вычисления максимального значения в массиве $A[1..n]$:

```
function max(A)
  m:=A[1]
  for i := 2 to n do
    if A[i] > m then m := A[i]
  return (m)
```

Упражнения по асимптотическим обозначениям

7. [3] Верно или неверно?

   а) $2^{n+1} = O(2^n)$

   б) $2^{2n} = O(2^n)$

8. [3] Для каждой из следующих пар функций функция $f(n)$ является членом одного из множеств функций $O(g(n))$, $\Omega(g(n))$ или $\Theta(g(n))$. Определите, членом какого множества является функция в каждом случае и обоснуйте свой вывод.

   а) $f(n) = \log n^2$; $g(n) = \log n + 5$

   б) $f(n) = \sqrt{n}$; $g(n) = \log n^2$

   в) $f(n) = \log^2 n$; $g(n) = \log n$

   г) $f(n) = n$; $g(n) = \log^2 n$

   д) $f(n) = n\log n + n$; $g(n) = \log n$

   е) $f(n) = 10$; $g(n) = \log 10$

   ж) $f(n) = 2^n$; $g(n) = 10n^2$

   з) $f(n) = 2^n$; $g(n) = 3^n$

9. [3] Для каждой из следующих пар функций $f(n)$ и $g(n)$ определите, справедливо ли $f(n) = O(g(n))$ и $g(n) = O(f(n))$.

   а) $f(n) = (n^2 - n)/2$; $g(n) = 6n$

   б) $f(n) = n + 2\sqrt{n}$; $g(n) = n^2$

   в) $f(n) = n\log n$; $g(n) = n\sqrt{n}$ /2

   г) $f(n) = n + \log n$; $g(n) = \sqrt{n}$

   д) $f(n) = 2(\log n)^2$; $g(n) = \log n + 1$

   е) $f(n) = 4n\log n + n$; $g(n) = (n^2 - n)/2$

10. [3] Докажите, что $n^3 - 3n^2 - n + 1 = \Theta(n^3)$.

11. [3] Докажите, что $n^2 = O(2^n)$.

12. [3] Для каждой из следующих пар функций $f(n)$ и $g(n)$ найдите положительную константу c, при которой $f(n) \le c \cdot g(n)$ для всех $n > 1$.

    а) $f(n) = n^2 + n + 1$, $g(n) = 2n^3$

    б) $f(n) = n\sqrt{n} + n^2$, $g(n) = n^2$

    в) $f(n) = n^2 - n + 1$, $g(n) = n^2/2$

13. [3] Докажите, что если $f_1(n) = O(g_1(n))$ и $f_2(n) = O(g_2(n))$, тогда $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$.

14. [3] Докажите, что если $f_1(N) = \Omega(g_1(n))$ и $f_2(n) = \Omega(g_2(n))$, тогда $f_1(n) + f_2(n) = \Omega(g_1(n) + g_2(n))$.

15. [3] Докажите, что если $f_1(n) = O(g_1(n))$ и $f_2(n) = O(g_2(n))$, тогда $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.

16. [5] Докажите, что для всех $k \ge 1$ и всех множеств констант $\{a_k, a_{k-1}, ..., a_1, a_0\} \in R$ верно $a_k n^k + a_{k-1} n^{k-1} + ... + a_1 n + a_0 = O(n^k)$.

17. [5] Докажите, что для любых вещественных констант a и b, $b > 0$ верно $(n + a)^b = \Theta(n^b)$.

18. [5] Упорядочите указанные в таблице функции в возрастающем порядке. При наличии двух или более функций одинакового порядка укажите их.

n	2^n	$n\lg n$	$\ln n$
$n - n^3 + 7n^5$	$\lg n$	$\sqrt{n}$	e^n
$n^2 + \lg n$	n^2	2^{n-1}	$\lg \lg n$
n^3	$(\lg n)^2$	$n!$	$n^{1+\varepsilon}$, где $0 < \varepsilon < 1$

19. [5] Упорядочите указанные в таблице функции в возрастающем порядке. При наличии двух или более функций одинакового порядка укажите их.

$\sqrt{n}$	n	2^n
$n\log n$	$n - n^3 + 7n^5$	$n^2 + \log n$
n^2	n^3	$\log n$
$n^{1/3} + \log n$	$(\log n)^2$	$n!$
$\ln n$	$n/\log n$	$\log \log n$
$(1/3)^n$	$(3/2)^n$	6

20. [5] Найдите две функции $f(n)$ и $g(n)$, которые удовлетворяют перечисленным условиям. Если таких f и g нет, укажите на этот факт в ответе.

а) $f(n) = o(g(n))$ и $f(n) \neq \Theta(g(n))$

б) $f(n) = \Theta(g(n))$ и $f(n) = o(g(n))$

в) $f(n) = \Theta(g(n))$ и $f(n) \neq O(g(n))$

г) $f(n) = \Omega(g(n))$ и $f(n) \neq O(g(n))$

21. [5] Верно или неверно?

а) $2n^2 + 1 = O(n^2)$

б) $\sqrt{n} = O(\log n)$

в) $\log n = O(\sqrt{n})$

г) $n^2(1 + \sqrt{n}) = O(n^2 \log n)$

д) $3n^2 + \sqrt{n} = O(n^2)$

е) $\sqrt{n} \log n = O(n)$

ж) $\log n = O(n^{-1/2})$

22. [5] Для каждой из следующих пар функции $f(n)$ и $g(n)$ укажите, какие из перечисленных равенств справедливы: $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, $f(n) = \Theta(g(n))$.

а) $f(n) = n^2 + 3n + 4$, $g(n) = 6n + 7$

б) $f(n) = n\sqrt{n}$, $g(n) = n^2 - n$

в) $f(n) = 2^n - n^2$, $g(n) = n^4 + n^2$

23. [3] Ответьте на следующие вопросы и обоснуйте свой ответ.

а) Если доказано, что время исполнения алгоритма в наихудшем случае определяется как $O(n^2)$, возможно ли, что для некоторых входных экземпляров это время будет определяться как $O(n)$?

б) Если доказано, что время исполнения алгоритма в наихудшем случае определяется как $O(n^2)$, возможно ли, что для всех входных экземпляров это время будет определяться как $O(n)$?

в) Если доказано, что время исполнения алгоритма в наихудшем случае определяется как $\Theta(n^2)$, возможно ли, что на некоторых входных экземплярах это время будет определяться как $O(n)$?

г) Если доказано, что время исполнения алгоритма в наихудшем случае определяется как $\Theta(n^2)$, возможно ли, что для всех входных экземпляров это время будет определяться как $O(n)$?

д) Принадлежит ли функция $f(n)$ множеству $\Theta(n^2)$ (т. е. $f(n) = \Theta(n^2)$), где $f(n) = 100n^2$ для четных n и $f(n) = 20n^2 - n\log_2 n$ для нечетных n?

24. [3] Определите, верны ли следующие тождества, либо укажите, что это определить невозможно. Обоснуйте свой ответ.

а) $3^n = O(2^n)$

б) $\log 3^n = O(\log 2^n)$

в) $3^n = \Omega(2^n)$

г) $\log 3^n = \Omega(\log 2^n)$

25. [5] Для каждой из следующих функций $f(n)$ найдите простую функцию $g(n)$, для которой $f(n) = \Theta(g(n))$.

а) $f(n) = \sum_{i=1}^{n} \frac{1}{}$

б) $f(n) = \sum_{i=1}^{n} \left\lceil \frac{1}{i} \right\rceil$

в) $f(n) = \sum_{i=1}^{n} \log i$

г) $f(n) = \log(n!)$

26. [5] Расположите следующие функции в возрастающем асимптотическом порядке:

$$f_1(n) = n^2 \log_2 n,\ f_2(n) = n(\log_2 n)^2,\ f_3(n) = \sum_{i=0}^{n} 2^i,\ f_4(n) = n^2 \log_2 (\sum_{i=0}^{n} 2^i)$$

27. [5] Расположите следующие функции в возрастающем асимптотическом порядке. При наличии двух или более функций одинакового порядка укажите их.

$$f_1(n) = \sum_{i=1}^{n} \sqrt{i},\ f_2(n) = (\sqrt{n}) \log n,\ f_3(n) = n\sqrt{\log n},\ f_4(n) = 12n^{\frac{3}{2}} + 4n$$

28. [5] Для каждой из следующих функций $f(n)$ найдите простую функцию $g(n)$, такую что $f(n) = \Theta(g(n))$. (Вообще говоря, вы должны уметь доказывать полученный результат, предоставляя соответствующие параметры, но это не требуется для данного задания.)

а) $f(n) = \sum_{i=1}^{n} 3i^4 + 2i^3 - 19i + 20$

б) $f(n) = \sum_{i=1}^{n} 3(4^i) + 2(3^i) - i^{19} + 20$

в) $f(n) = \sum_{i=1}^{n} 5^i + 3^{2i}$

29. [5] Какое из следующих уравнений верно?

а) $\sum_{i=1}^{n} 3^i = \Theta(3^{n-1})$

б) $\sum_{i=1}^{n} 3^i = \Theta(3^n)$

в) $\sum_{i=1}^{n} 3^i = \Theta(3^{n+1})$

30. [5] Для каждой из следующих функций $f(n)$ найдите простую функцию $g(n)$, при которой $f(n) = \Theta(g(n))$

а) $f_1(n) = (1000)2^n + 4^n$

б) $f_2(n) = n + n\log n + \sqrt{n}$

в) $f_3(n) = \log(n^{20}) + (\log n)^{10}$

г) $f_4(n) = (0,99)^n + n^{100}$

31. [5] Для каждой пары выражений (A, B) в таблице укажите, какой именно функцией является A для B — функцией O, o, Ω, w или Θ. Обратите внимание, что для любой из этих пар возможны несколько, один или ни одного варианта. Укажите все правильные отношения.

	A	B
а)	n^{100}	2^n
б)	$(\lg n)^{12}$	$\sqrt{n}$
в)	$\sqrt{n}$	$n^{\cos(\pi n/8)}$
г)	10^n	100^n
д)	$n^{\lg n}$	$(\lg n)^n$
е)	$\lg(n!)$	$n\lg n$

Суммирование

32. [5] Докажите, что:

$$1^2 - 2^2 + 3^2 - 4^2 + \ldots + (-1)^{k-1}k^2 = (-1)^{k-1}k(k+1)/2$$

33. [5] Определите формулу суммы для чисел строки i следующего треугольника и докажите ее правильность. Каждый элемент строки является суммой части строки из трех элементов непосредственно над ним. Отсутствующие элементы полагаются равными нулю.

```
                1
            1   1   1
        1   2   3   2   1
    1   3   6   7   6   3   1
1   4  10  16  19  16  10   4   1
```

34. [3] Допустим, что рождественские праздники длятся n дней. Сколько точно подарков прислала мне *"любовь моя верная"*? (Если вы не понимаете, о чем речь, выясните смысл вопроса самостоятельно.)

35. [5] Рассмотрим следующий фрагмент кода.

```
for i=1 to n do
   for j=i to 2*i do
      output ''foobar''
```

Пусть $T(n)$ означает, сколько раз слово "foobar" печатается в зависимости от значения n.

а) Выразите $T(n)$ в виде суммы (точнее, в виде двух вложенных сумм).

б) Упростите выражение суммы. Полностью распишите все преобразования.

36. [5] Рассмотрим следующий фрагмент кода.

```
for i=1 to n/2 do
   for j=i to n-i do
      for k=1 to j  do
         output ''foobar''
```

Допустим, что n — четное. Пусть $T(n)$ означает, сколько раз слово "foobar" выводится в зависимости от значения n.

а) Выразите функцию $T(n)$ в виде трех вложенных сумм.

б) Упростите выражение суммирования. Полностью распишите все преобразования.

37. [6] На уроках арифметики в школе нам говорили, что $x \times y$ означает, что число x нужно написать y раз подряд и сосчитать сумму, т. е. $5 \times 4 = 5 + 5 + 5 + 5 = 20$. Выразите в виде функции от n и b временную сложность умножения двух чисел, которые в b-ичной системе счисления состоят из n цифр (люди работают в десятичной системе счисления, а компьютеры — в двоичной) методом многократного сложения. Будем считать, что умножение или сложение однозначных чисел занимает $O(1)$ времени. (Подсказка: подумайте, насколько большим может быть число y в зависимости от n и b?)

38. [6] На уроках арифметики нас также учили умножать большие числа поразрядно, т. е. $127 \times 211 = 127 \times 1 + 127 \times 10 + 127 \times 200 = 26\,397$. Выразите в виде функции от n временную сложность умножения этим методом двух чисел из n цифр. Будем считать, что умножение или сложение однозначных чисел занимает $O(1)$ времени.

Логарифмы

39. [5] Докажите следующие тождества:

а) $\log_a(xy) = \log_a x + \log_b y$.

б) $\log_a x^y = y \log_a x$.

в) $\log_a x = \dfrac{\log_b x}{\log_b a}$

г) $x^{\log_b y} = y^{\log_b x}$

40. [3] Докажите, что $\lceil \lg(n+1) \rceil = \lfloor \lg n \rfloor + 1$.

41. [3] Докажите, что двоичное представление числа $n \geq 1$ содержит $\lfloor \lg_2 n \rfloor + 1$ битов.

42. [5] В одной из моих научных статей я привел пример алгоритма сортировки методом сравнений с временной сложностью $O(n \log(\sqrt{n}))$. Почему это возможно, если нижняя граница сортировки определена как $\Omega(n \log n)$?

Задачи, предлагаемые на собеседовании

43. [5] Имеется множество S из n чисел. Из этого множества нужно выбрать такое подмножество S' из k чисел, чтобы вероятность вхождения каждого элемента из множества S в подмножество S' была одинаковой (т. е., чтобы вероятность выбора каждого элемента была равна k/n). Выбор нужно сделать за один проход по числам множества S. Решите задачу также для случая, когда число n неизвестно?

44. [5] Нужно сохранить тысячу элементов данных в тысяче узлов. В каждом узле можно сохранить копии только трех разных элементов. Разработайте схему создания копий, позволяющую минимизировать потерю данных при выходе узлов из строя. Сколько утерянных элементов данных нужно ожидать в случае выхода из строя трех произвольных узлов?

45. [5] Имеется следующий алгоритм поиска наименьшего числа в массиве чисел $A[0,...,n]$. Для хранения текущего минимального числа используется переменная *tmp*. Начиная с ячейки массива $A[0]$ значение переменной *tmp* сравнивается по порядку со значениями ячеек $A[1]$, $A[2]$, ..., $A[N]$. Если значение ячейки $A[i]$ окажется меньше значения *tmp* ($A[i] < tmp$), то оно записывается в переменную *tmp* ($tmp = A[i]$). Сколько нужно ожидать таких операций присваивания?

46. [5] Имеется 100-этажное здание и несколько небольших шариков из камня. Нужно определить самый нижний этаж, чтобы брошенный с него шарик разбился. Сколько времени понадобится для определения этого этажа при неограниченном количестве шариков? При наличии только двух шариков?

47. [5] Вам дали 10 кошельков с золотыми монетами. Монеты в девяти из этих кошельков весят по 10 грамм каждая, а в оставшемся кошельке — на 1 грамм меньше. С помощью цифровых весов нужно найти кошелек с легкими монетами, выполнив лишь одно взвешивание.

48. [5] У вас есть восемь шариков одинакового размера. Семь из них имеют одинаковый вес, восьмой шарик чуть тяжелее остальных. Нужно найти этот шарик, выполнив лишь два взвешивания.

49. [5] Допустим, что планируется слить n компаний в одну. Сколько имеется разных способов для осуществления этого слияния?

50. [5] Число Рамануджана-Харди — это число, представимое в виде суммы двух кубов двумя различными способами. Иными словами, существуют четыре разных числа a, b, c

и d, для которых $a^3 + b^3 = c^3 + d^3$. Сгенерируйте все числа Рамануджана-Харди для a, b, c, $d < n$.

51. [7] Шести пиратам нужно поделить между собой 300 долларов следующим способом. Самый старший пират предлагает, как нужно разделить деньги, после чего пираты голосуют по его предложению. Если предложение одобрено, по крайней мере половиной пиратов, то деньги распределяются в соответствии с предложенным способом. В противном случае автора предложения убивают, и следующий по старшинству пират предлагает свой способ. Процесс повторяется. Ответьте, каков будет результат этого деления, и обоснуйте. То есть, сколько пиратов останется в живых, и каким образом будут распределены деньги? Все пираты обладают хорошим умом, и самая приоритетная задача каждого — остаться в живых, а следующая по важности — получить как можно большую долю денег.

52. [7] Вариант предыдущей задачи. Пираты делят только один неделимый доллар. Кто получит этот доллар, и сколько пиратов будет убито?

Задачи по программированию

Эти задачи доступны на сайтах **http://www.programming-challenges.com** и **http://uva.onlinejudge.org**.

1. Primary Arithmetic. 110501/10035.
2. A Multiplication Game. 110505/847.
3. Light, More Light. 110701/10110.

ГЛАВА 3

Структуры данных

Замену структуры данных в медленно работающей программе сравнить с пересадкой органа. Такие важные классы *абстрактных типов данных*, как контейнеры, словари и очереди с приоритетами, могут реализовываться посредством разных, но функционально эквивалентных *структур данных*. Замена структуры данных одного типа структурой данных другого типа не влияет на правильность программы, т. к. предполагается, что одна правильная реализация заменяется другой правильной реализацией. Но в реализации другого типа данных могут применяться операции с иными временными отношениями, в результате чего общая производительность программы может значительно повыситься. Подобно ситуации с больным, нуждающимся в пересадке одного органа, для повышения производительности программы может оказаться достаточным заменить лишь один ее компонент.

Конечно, лучше с рождения иметь здоровое сердце, чем жить в ожидании донорского. То же самое справедливо и в случае со структурами данных. Наибольшую пользу от применения хороших структур данных можно получить, лишь заложив их использование в программу с самого начала. При написании этой книги предполагалось, что ее читатели уже имеют знания об элементарных структурах данных и манипуляциях указателями.

Но так как в сегодняшних курсах по структурам данных внимание фокусируется больше на абстракции данных и на объектно-ориентированном программировании, чем на деталях представления структур в памяти, то мы повторим этот материал здесь, чтобы убедиться в том, что вы его полностью понимаете.

Как и при изучении большинства предметов, при изучении структур данных важнее хорошо освоить основной материал, чем бегло ознакомиться с более сложными понятиями. Здесь мы обсудим три фундаментальных абстрактных типа данных, *контейнеры*, *словари* и *очереди с приоритетами*, и рассмотрим, как они реализуются посредством массивов и списков. Более сложные реализации структур данных рассматриваются в релевантной задаче в каталоге задач.

3.1. Смежные и связные структуры данных

В зависимости от реализации (посредством массивов или указателей) структуры данных можно четко разбить на два типа — *смежные* и *связные*.

- *Смежные структуры данных* реализованы в виде непрерывных блоков памяти. К ним относятся массивы, матрицы, кучи и хэш-таблицы.
- *Связные структуры данных* реализованы в отдельных блоках памяти, связанных вместе с помощью *указателей*. К этому виду структур данных относятся списки, деревья и списки смежных вершин графов.

В этом разделе дается краткий обзор сравнительных характеристик смежных и связных структур данных. Разница между ними более тонкая, чем может показаться с первого взгляда, поэтому я призываю не игнорировать этот материал, даже если вы и знакомы с этими типами структур данных.

3.1.1. Массивы

Массив представляет собой основную структуру данных смежного типа. Записи данных в массивах имеют постоянный размер, что позволяет с легкостью найти любой элемент по его *индексу* (или адресу).

Хорошей аналогией массива будет улица с домами, где каждый элемент массива соответствует дому, а индекс элемента — номеру дома. Считая, что все дома одинакового размера и пронумерованы последовательно от 1 до n, можно определить точное местонахождение каждого дома по его адресу[1].

Перечислим достоинства массивов.

- *Постоянное время доступа при условии наличия индекса.* Так как индекс каждого элемента массива соответствует определенному адресу в памяти, то при наличии соответствующего индекса доступ к произвольному элементу массива осуществляется практически мгновенно.
- *Эффективное использование памяти.* Массивы содержат только данные, поэтому память не тратится на указатели и другую форматирующую информацию. Кроме этого, для элементов массива не требуется использовать метку конца записи, т. к. все элементы массива имеют одинаковый размер.
- *Локальность в памяти.* Одна из самых распространенных идиом программирования — обработка элементов структуры данных в цикле. Массивы хорошо подходят для операций такого типа, поскольку обладают отличной локальностью в памяти. В современных компьютерных архитектурах физическая непрерывность последовательных обращений к данным помогает воспользоваться высокоскоростной *кэш-памятью.*

Недостатком массивов является то, что их размер нельзя изменять в процессе исполнения программы. Попытка обращения к $(n + 1)$-му элементу массива размером n элементов немедленно вызовет аварийное завершение программы. Этот недостаток можно компенсировать объявлением массивов очень больших размеров, но это может повлечь за собой чрезмерные затраты памяти, что опять наложит ограничения на возможности программы.

В действительности, размеры массива можно изменять во время исполнения программы посредством приема, называющегося *динамическим выделением памяти.* Допустим, мы начнем с одноэлементного массива размером m и будем удваивать его каждый раз до $2m$, когда предыдущий размер становится недостаточным. Этот процесс состоит из выделения памяти под новый непрерывный массив размером $2m$, копирования со-

[1] Данная аналогия неприменима в случае с нумерацией домов в Японии. Там дома нумеруют в порядке их возведения, а не физического расположения. Поэтому найти какой-либо дом в Японии по его адресу, не имея карты, очень трудно.

держимого старого массива в нижнюю половину нового и возвращения памяти старого массива в систему распределения памяти.

Очевидным расточительством в этой процедуре является операция копирования содержимого старого массива в новый при каждом удвоении размера массива. Это порождает вопрос: сколько раз может возникнуть необходимость перекопировать элемент массива после n вставок? Давайте разберемся. Первый вставленный элемент нужно будет перекопировать при расширении массива после первой, второй, четвертой, восьмой и т. д. вставок. Для расширения массива до размера в n элементов потребуется $\log_2 n$ удваиваний. Но большинство элементов не подвергается слишком большому числу перемещений. Более того, элементы с $(n/2 + 1)$ по n будут перемещены, самое большее, один раз, а могут и вообще не перемещаться.

Если половина элементов перемещается один раз, четверть элементов два раза и т. д., то общее число перемещений определяется следующей формулой:

$$M = \sum_{i=1}^{\lg n} i \cdot n/2^i = n\sum_{i=1}^{\lg n} i/2^i \le n\sum_{i=1}^{\infty} i/2^i = 2n$$

Таким образом, каждый из n элементов массива, в среднем, перемещается только два раза, а общая временная сложность управления динамическим массивом определяется той же самой функцией $O(n)$, какая справедлива для работы с одним статическим массивом достаточного размера.

Самой главной проблемой при использовании динамических массивов является отсутствие гарантии постоянства времени доступа *в наихудшем случае*. Теперь все обращения будут быстрыми, за исключением тех относительно нечастых обращений, вызывающих удвоение массива. Зато у нас есть уверенность, что n-е обращение к массиву будет выполнено достаточно быстро, чтобы *общее* затраченное усилие осталось таким же $O(n)$.

3.1.2. Указатели и связные структуры данных

Указатели позволяют удерживать воедино связные структуры. Указатель — это адрес ячейки памяти. Переменная, содержащая указатель на элемент данных, может предоставить бóльшую гибкость в работе с этими данными, чем просто их копия. В качестве примера указателя можно привести номер сотового телефона, который позволяет связаться с владельцем телефона независимо от его местоположения внутри зоны действия сети.

Так как синтаксис и возможности указателей значительно различаются в разных языках программирования, то мы начнем с краткого обзора указателей в языке C. Переменная указателя `p` содержит адрес памяти, по которому находится определенный блок данных[1]. В языке C указатель при объявлении получает тип, соответствующий типу данных, на которые этот указатель может ссылаться. Операция, обозначаемая `*p`, называется *разыменованием указателя* и возвращает значение элемента данных, рас-

[1] В языке C разрешается прямая манипуляция адресами такими способами, которые могут привести Java-программистов в ужас, но в этой книге мы воздержимся от применения подобных методов.

положенного по адресу, содержащемуся в переменной указателя p. А операция, обозначаемая &x, называется *взятием адреса* и возвращает адрес (т. е. указатель) данной переменной x. Специальное значение NULL применяется для обозначения неинициализированного указателя или указателя на последний элемент структуры данных.

Все связные структуры данных имеют определенные общие свойства, что видно из следующего объявления типа связного списка (листинг 3.1).

Листинг 3.1. Объявление структуры связанного списка

```
typedef struct list {
   item_type item;     /* Данные */
   struct list *next;  /* Указатель на следующий узел */
} list;
```

В частности:

- каждый узел в нашей структуре данных (структура list) содержит одно или несколько полей, предназначенных для хранения данных (поле item);
- каждый узел также содержит поле указателя на следующий узел (поле next). Это означает, что в связных структурах данных большой объем используемой ими памяти должен отдаваться под хранение указателей, а не полезных данных;
- наконец, требуется указатель на начало структуры, чтобы мы знали, откуда начинать обращение к ней.

Простейшей связной структурой является список, пример которого показан на рис. 3.1.

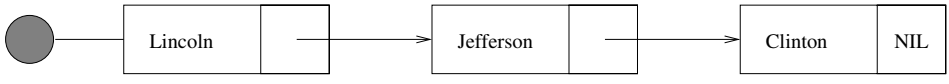


Рис. 3.1. Пример связного списка с полями данных и указателями

Списки поддерживают три основных операции: *поиск* (search), *вставку* (insert) и *удаление* (delete). В *двунаправленных* или *двусвязных* списках (doubly-linked list) каждый узел содержит указатель как на следующий, так и на предыдущий узел. Это упрощает определенные операции за счет дополнительного поля указателя в каждом узле.

Поиск элемента в связном списке

Поиск элемента x в связном списке можно выполнять итеративным или рекурсивным методом. В листинге 3.2 приведен пример реализации рекурсивного поиска.

Листинг 3.2. Рекурсивный поиск элемента в связанном списке

```
list *search_list(list *l, item_type x)
{
     if (l == NULL) return(NULL);
     if (l->item == x)
          return(l);
```

```
    else
        return( search_list(l->next, x) );
}
```

Если список содержит элемент x, то он находится либо в начале списка, либо в меньшей оставшейся части списка. В конце концов, задача сводится к поиску в пустом списке, который, очевидно, не может содержать элемент x.

Вставка элемента в связный список

Код вставки элементов в однонаправленный связный список представлен в листинге 3.3.

Листинг 3.3. Вставка элемента в однонаправленный связанный список

```
void insert_list(list **l, item_type x)
{
   list *p;          /* Временный указатель */
   p = malloc( sizeof(list) );
   p->item = x;
   p->next = *l;
   *l = p;
}
```

Так как нам не требуется содержать элементы списка в каком-либо определенном порядке, то мы можем вставлять каждый новый элемент туда, куда его проще всего вставить. Вставка элемента в начало списка позволяет избежать необходимости обхода списка, хотя и требует обновления указателя (переменная l) на начало списка.

Обратите внимание на две особенности языка C. Функция malloc возвращает указатель на блок памяти достаточного размера, выделенный для нового узла, в котором будет храниться элемент x. Двойная звездочка (**l) означает, что переменная l является указателем на указатель на узел списка. Таким образом, строчка кода *l=p; копирует значение p в блок памяти, на который ссылается переменная l, которая является внешней переменной, обеспечивающей доступ к началу списка.

Удаление элемента из связного списка

Удаление элемента из связного списка является более сложной операцией. Сначала нам нужно найти указатель на элемент списка, предшествующий удаляемому элементу. Это выполняется рекурсивным способом (листинг 3.4).

Листинг 3.4. Поиск указателя на элемент, предшествующий удаляемому

```
list *predecessor_list(list *l, item_type x)
{
   if ((l == NULL) || (l->next == NULL)) {
     printf("Error: predecessor sought on null list.\n");
       return(NULL);
   }
```

```
   if ((l->next)->item == x)
     return(l);
   else
     return( predecessor_list(l->next, x) );
}
```

Найти элемент, предшествующий удаляемому, нужно потому, что он содержит указатель `next` на следующий, в данном случае удаляемый, узел, который нужно обновить после удаления узла. После проверки существования узла, подлежащего удалению, собственно операция удаления не представляет собой ничего сложного. При удалении первого элемента связного списка нужно быть особенно внимательным, чтобы не забыть обновить указатель (переменную `l`) на начало списка (листинг 3.5).

Листинг 3.5. Удаление элемента связанного списка

```
delete_list(list **l, item_type x)
{
   list *p;                  /* Указатель на узел*/
   list *pred;               /* Указатель на предшествующий узел */
   list *search_list(), *predecessor_list();
   p = search_list(*l,x);
   if (p != NULL) {
     pred = predecessor_list(*l,x);
     if (pred == NULL)     /* Соединяем список */
        *l = p->next;
     else
        pred->next = p->next;
           free(p);              /* Освобождение памяти узла */
  }
}
```

В языке C требуется явное освобождение памяти, поэтому после удаления узла необходимо освободить занимаемую им память, чтобы возвратить ее в систему.

3.1.3. Сравнение

Связные списки имеют следующие преимущества над статическими массивами:

- переполнение в связных структурах невозможно, если только сама память не переполнена;
- операции вставки и удаления элементов проще соответствующих операций над непрерывными списками (т. е. массивами);
- при работе с большими записями перемещение указателей происходит легче и быстрее, чем перемещение самих записей.

Недостатки связных списков таковы:

- связным структурам необходимо дополнительное место для хранения указателей;
- в связном списке нет эффективного произвольного доступа к элементам;
- массивы обладают лучшей локальностью в памяти и более эффективны в использовании кэш-памяти, чем связные списки.

Подведение итогов

Динамическое выделение памяти обеспечивает гибкость в выборе способа и момента использования этого ограниченного ресурса.

Напоследок заметим, что списки и массивы можно рассматривать, как рекурсивные объекты.

- *Списки.* После удаления первого элемента связного списка мы имеем такой же связный список, только меньшего размера. То же самое справедливо для строк, поскольку в результате удаления символов из строки получается более короткая строка.
- *Массивы.* Отделение первых k элементов от массива из n элементов дает нам два массива меньших размеров, а именно размером k и $n - k$ элементов соответственно.

Знание этого свойства позволяет упростить обработку списков и создавать эффективные алгоритмы типа "разделяй и властвуй", такие как быстрая сортировка (quicksort) и двоичный поиск.

3.2. Стеки и очереди

Термин *контейнер* (container) обозначает структуру данных, позволяющую хранить и извлекать данные *независимо от содержимого*. В противоположность контейнерам, словари представляют собой абстрактные типы данных, которые извлекаются по ключевому значению или содержимому. Словари рассматриваются в *разделе 3.3*.

Контейнеры различаются по поддерживаемому ими типу извлечения данных. В двух наиболее важных типах контейнеров порядок извлечения зависит от порядка помещения:

- *Стеки.* Извлечение данных осуществляется в порядке LIFO ("last in, first out", "последним вошел — первым вышел"). Стеки легко реализуются и обладают высокой эффективностью. По этой причине стеки удобно применять в случаях, когда порядок извлечения данных не имеет никакого значения, например, при обработке пакетных заданий. Операции вставки и извлечения данных для стеков называются *push* (запись в стек) и *pop* (снятие со стека):
  - *push(x,s)* — вставить элемент x на верх (в конец) стека s;
  - *pop(s)* — извлечь (и удалить) верхний (последний) элемент из стека s.

  Порядок LIFO возникает во многих реальных ситуациях. Например, из набитого битком вагона метро пассажиры выходят в порядке LIFO. Продукты из холодильника нередко вынимаются в этом же порядке, с игнорированием сроков годности. По крайней мере, такой порядок применяется в моем холодильнике. В алгоритмах порядок LIFO обычно возникает при выполнении рекурсивных операций.
- *Очереди.* Очереди поддерживают порядок извлечения FIFO ("first-in, first-out", "первым вошел — первым вышел"). Использование этого порядка определенно самый справедливый способ управления временем ожидания обслуживания. Компьютер обрабатывает задачи в порядке FIFO, чтобы минимизировать максимальное время ожидания. Обратите внимание, что *среднее* время ожидания будет одинаковым, независимо от применяемого порядка, будь то FIFO или LIFO. Но так как данные мно-

гих вычислительных приложений не теряют актуальность бесконечно долго, вопрос максимального времени ожидания становится чисто академическим.

Очереди реализовать несколько труднее, чем стеки, и поэтому их применение больше подходит для приложений, в которых порядок извлечения данных является важным, например, эмуляции определенных процессов. Применительно к очередям операции вставки и извлечения данных называются *enqueue* (поставить в очередь) и *dequeue* (вывести из очереди) соответственно:

- *enqueue(x,q)* — вставить элемент x в конец очереди q;
- *dequeue(q)* — извлечь (и удалить) элемент в начале очереди q.

Далее в книге мы увидим применение очередей в качестве основной структуры данных для управления поиском в ширину в графах.

На практике стеки и очереди можно реализовать посредством массивов или связных списков. Ключевой вопрос состоит в том, известна ли верхняя граница размера контейнера заранее, т. е. имеется ли возможность использовать статические массивы.

3.3. Словари

Тип данных *словарь* позволяет доступ к данным по содержимому. Словари применяются для хранения данных, которые можно быстро найти в случае надобности. Далее приводится список основных операций, поддерживаемых в словарях:

- *search(D,k)* — возвращает указатель на элемент словаря D с ключом k, если такой элемент существует;
- *insert(D,x)* — добавляет элемент, на который указывает x, в словарь D;
- *delete(D,x)* — удаляет из словаря D элемент, на который указывает x.

Некоторые словари также поддерживают другие полезные операции:

- *max(D)* и *min(D)* — возвращает указатель на элемент множества D, имеющий наибольший или наименьший ключ. Это позволяет использовать словарь в качестве очереди с приоритетами, которая рассматривается в *разделе 3.5*;
- *predecessor(D,k)* и *successor(D,k)* — возвращает элемент из отсортированного массива D, предшествующий элементу с ключом k или стоящий сразу после него соответственно. Это позволяет обрабатывать в цикле все элементы этой структуры данных.

С помощью только что перечисленных словарных операций можно выполнять многие распространенные задачи обработки данных. Допустим, нам нужно удалить все повторяющиеся имена из списка рассылки, затем отсортировать и распечатать получившийся список. Для выполнения этой задачи инициализируем пустой словарь D, для которого ключом поиска будет служить имя записи. Потом считываем записи из списка рассылки и для каждой записи выполняем операцию *search* в словаре на предмет наличия в нем данного имени. Если имя отсутствует в словаре, то вставляем его с помощью операции *insert*. Обработав весь список, извлекаем результаты из словаря. Для этого, начиная с наименьшего элемента словаря (операция *min(D)*), выполняем операцию *successor*, пока не дойдем до наибольшего элемента (операция *max(D)*), обойдя таким образом по порядку все элементы словаря.

Определяя подобные задачи в терминах абстрактных операций над словарем, мы отвлекаемся от деталей представления структуры данных и концентрируемся на решении задачи.

Далее в этом разделе мы внимательно рассмотрим простые реализации словаря на основе массивов и связных списков. Более мощные реализации, такие как использование двоичных деревьев поиска (см. *раздел 3.4*) и хэш-таблицы (см. *раздел 3.7*), также являются привлекательными вариантами. Подробно словарные структуры данных рассматриваются в *разделе 12.1*. Читателям настоятельно рекомендуется просмотреть этот раздел, чтобы получить лучшее представление об имеющихся возможностях.

Остановка для размышлений. Сравнение реализаций словаря (I)

ЗАДАЧА. Определите асимптотическое время исполнения в наихудшем случае для каждой из семи основных словарных операций (*search*, *insert*, *delete*, *successor*, *predecessor*, *minimum* и *maximum*) для структуры данных, реализованной в виде:

- неотсортированного массива;
- отсортированного массива.

Решение. Эта (а также следующая) задача демонстрирует компромиссы, на которые приходится идти при разработке структур данных. Конкретное представление данных может обеспечить эффективную реализацию одних операций за счет снижения эффективности других.

В решении этой задачи мы будем полагать, что кроме массивов, упомянутых в формулировке, мы имеем доступ к нескольким переменным, в частности, к переменной n, содержащей текущее количество элементов массива. Обратите внимание на необходимость обновлять эти переменные в операциях, изменяющих их значения (например, *insert* и *delete*), и на то, что стоимость такого сопровождения нужно включать в стоимость этих операций.

Сложность основных словарных операций для неотсортированных и отсортированных массивов показана в табл. 3.1.

Таблица 3.1. Сложность основных словарных операций для массивов

Словарная операция	Неотсортированный массив	Отсортированный массив
search(L,k)	$O(n)$	$O(\log n)$
insert(L,x)	$O(1)$	$O(n)$
delete(L,x)	$O(1)$*	$O(n)$
successor(L,x)	$O(n)$	$O(1)$
predecessor(L,x)	$O(n)$	$O(1)$

Чтобы понять, почему операция имеет данную сложность, необходимо выяснить, каким образом она реализуется. Рассмотрим сначала эти операции для *неотсортированного* массива A.

- При выполнении операции *search* ключ поиска k сравнивается с (возможно) каждым элементом неотсортированного массива. Таким образом, в наихудшем случае, когда ключ k отсутствует в массиве, время поиска будет линейным.
- При выполнении операции *insert* значение переменной n увеличивается на единицу, после чего элемент x копируется в n-ю ячейку массива $A[n]$. Основная часть массива не затрагивается этой операцией, поэтому время ее исполнения будет постоянным.
- Операция *delete* несколько сложнее, что обозначается звездочкой (*) в табл. 3.1. По определению, в этой операции передается указатель x на элемент, который нужно удалить, поэтому нам не нужно тратить время на поиск этого элемента. Но удаление элемента из массива оставляет в нем промежуток, который нужно заполнить. Этот промежуток можно было бы заполнить, сдвинув все элементы массива, следующие за ним, т. е. элементы с $A[x + 1]$ по $A[n]$, вверх на одну позицию, но в случае удаления первого элемента эта операция займет время $\Theta(n)$. Поэтому будет намного лучше просто записать в ячейку $A[x]$ содержимое последней ячейки $A[n]$ массива и уменьшить значение n на единицу. Время исполнения этой операции будет постоянным.
- Определения операций *predecessor* и *successor* даются для отсортированного массива. Но результатом выполнения этих операций в неотсортированном массиве не будет элемент $A[x - 1]$ (или $A[x + 1]$), т. к. физически предыдущий (или следующий элемент) не обязательно будет таковым логически. В данном случае элементом, предшествующим элементу $A[x]$, будет наибольший из элементов, меньших $A[x]$. Аналогично, следующим элементом за $A[x]$ будет наименьший из элементов, бóльших $A[x]$. Чтобы найти предшествующий или следующий элемент в неотсортированном массиве A, необходимо просмотреть все его n элементов, что занимает линейное время.
- Операции *minimum* и *maximum* также определены только для отсортированного массива. Поэтому, чтобы найти наибольший или наименьший элемент в неотсортированном массиве, необходимо просмотреть все его элементы, что также занимает линейное время.

Реализация словаря в отсортированном массиве переворачивает наши представления о трудных и легких операциях. В данном случае нахождение нужного элемент посредством двоичного поиска занимает время $O(\log n)$, т. к. мы знаем, что средний элемент находится в ячейке массива $A[n/2]$. А поскольку верхняя и нижняя половины массива также отсортированы, то поиск может продолжаться рекурсивно в соответствующей половине. Количество делений массива пополам, требуемое для получения половины из одного элемента, равно $[\lg n]$.

Отсортированность массива полезна и для других словарных операций. Наименьший и наибольший элементы находятся в ячейках $A[1]$ и $A[n]$ соответственно, а элементы, идущие непосредственно до и после элемента $A[x]$, — в ячейках $A[x - 1]$ и $A[x + 1]$ соответственно.

Но операции вставки и удаления элементов становятся более трудоемкими, т. к. создание места для нового элемента или заполнение промежутка после удаления может потребовать перемещения многих элементов массива. Вследствие этого обе операции исполняются за линейное время. ■

Подведение итогов

При разработке структуры данных должны быть сбалансированы скорости выполнения всех поддерживаемых ею операций. Структура, обеспечивающая максимально быстрое выполнение как операции A, так и операции B, вполне может оказаться не самой подходящей (в смысле скорости работы) для этих операций, взятых в отдельности.

Остановка для размышлений. Сравнение реализаций словаря (11)

ЗАДАЧА. Определите асимптотическое время исполнения в наихудшем случае для каждой из семи основных словарных операций для структуры данных, реализованной в виде:

- однонаправленного связного неотсортированного списка;
- двунаправленного связного неотсортированного списка;
- однонаправленного связного отсортированного списка;
- двунаправленного связного отсортированного списка.

Решение. При оценке производительности необходимо принимать во внимание два момента: тип связности списка (однонаправленная или двунаправленная) и его упорядоченность (отсортированный или неотсортированный). Производительность этих операций для каждого типа структуры данных показана в табл. 3.2. Операции, оценка производительности которых представляет особые трудности, помечены звездочкой (*).

Таблица 3.2. Производительность словарных операций в связных структурах данных

Словарная операция	Односвязный неотсортированный список	Двусвязный неотсортированный список	Односвязный отсортированный список	Двусвязный отсортированный список
search(L,sk)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
insert(L,x)	$O(1)$	$O(1)$	$O(n)$	$O(n)$
delete(L,x)	$O(n)$*	$O(1)$	$O(n)$*	$O(1)$
successor(L,x)	$O(n)$	$O(n)$	$O(1)$	$O(1)$
predecessor(L,x)	$O(n)$	$O(n)$	$O(n)$*	$O(1)$
minimum(L)	$O(n)$	$O(n)$	$O(1)$	$O(1)$
maximum(L)	$O(n)$	$O(n)$	$O(1)$*	$O(1)$

Так же, как и в случае с неотсортированными массивами, операция поиска неизбежно будет медленной, а операции модифицирования — быстрыми.

- Операции *insert/delete*. Здесь сложность возникает при удалении элемента из односвязного списка. По определению операции *delete* передается указатель x на элемент, который нужно удалить. Однако в действительности нам нужен указатель на предшествующий элемент, т. к. именно его поле указателя необходимо обновить после удаления. Следовательно, требуется найти этот элемент, а его поиск в одно-

связном списке занимает линейное время. Данная проблема не возникает в двусвязном списке, т. к. мы можем сразу же получить предшественника удаляемого элемента.

Удаление элемента из отсортированного двусвязного списка выполняется быстрее, чем из отсортированного массива, т. к. связать предшествующий и последующий элементы списка дешевле, чем заполнять промежуток в массиве, перемещая оставшиеся элементы. Но удаление из односвязного отсортированного списка тоже усложняется необходимостью поиска предшествующего элемента.

- Операция *search*. Упорядоченность элементов не дает таких преимуществ при поиске в связном списке, как при поиске в массиве. Мы больше не можем выполнять двоичный поиск, т. к. мы не можем получить доступ к среднему элементу, не обратившись ко всем предшествующим ему элементам. Однако упорядоченность приносит определенную пользу в виде быстрого завершения неуспешного поиска. Действительно, если мы не нашли запись *Abbott*, дойдя до *Costello*[1], то можно сделать вывод, что такой записи не существует вообще. Тем не менее, поиск в наихудшем случае занимает линейное время.
- Операции *predecessor* и *successor*. Реализация операции *predecessor* усложняется уже упомянутой проблемой поиска указателя на предшествующий элемент. В отсортированных списках обоих типов логически следующий элемент эквивалентен следующему узлу и, поэтому, время обращения к нему постоянно.
- Операция *maximum*. Наибольший элемент находится в конце списка, и чтобы добраться до него, обычно потребуется время $\Theta(n)$ как в односвязном, так и в двусвязном списке.

  Но можно поддерживать указатель на конец списка при условии, что нас устраивают расходы по обновлению этого указателя при каждой вставке и удалении. В двусвязных списках этот указатель на конечный элемент можно обновлять за постоянное время: при вставке проверять, имеет ли `last->next` значение `NULL`, а при удалении переводить указатель `last` на предшественник `last` в списке, если удален последний элемент.

  В случае односвязных списков у нас нет эффективного способа найти этот предшественник. Как же мы получаем время исполнения операции *maximum* равным $\Theta(1)$ в таких списках? Трюк заключается в списывании затрат на каждое удаление, которое *уже* заняло линейное время. Дополнительный проход для обновления указателя, выполняемый за линейное время, не причиняет никакого вреда асимптотической сложности операции *delete*, но дает нам выигрыш в виде постоянного времени выполнения операции *maximum* как вознаграждение за четкое мышление. ■

3.4. Двоичные деревья поиска

Пока что мы ознакомились со структурами данных, позволяющими выполнять быстрый поиск *или* гибкое обновление, но не быстрый поиск *и* гибкое обновление одновременно. В неотсортированных двусвязных списках для операций вставки и удаления

[1] Abbott и Costello — знаменитый американский комический дуэт. — *Прим. перев.*

записей требуется время $O(1)$, а для операций поиска — линейное время в худшем случае. Отсортированные массивы обеспечивают логарифмическое время выполнения двоичного поиска и линейное время выполнения вставок и удалений.

Для двоичного поиска необходимо иметь быстрый доступ к двум элементам, а именно средним элементам верхней и нижней частей массива по отношению к данному элементу. Иными словами, нужен связный список, в котором каждый узел содержит два указателя. Так мы приходим к идее двоичных деревьев поиска.

Корневое двоичное дерево определяется рекурсивно либо как пустое, либо как состоящее из узла, называющегося корнем, из которого выходят два корневых двоичных дерева, называющиеся левым и правым поддеревом. В корневых деревьях порядок "родственных" узлов имеет значение, поэтому левое поддерево отличается от правого. На рис. 3.2 показаны пять разных двоичных деревьев, которые можно создать из трех узлов.

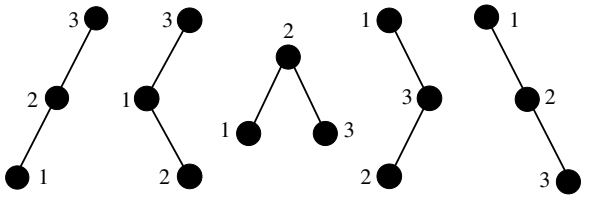


Рис. 3.2. Пять разных двоичных деревьев поиска с тремя узлами

Каждый узел двоичного *дерева поиска* помечается ключом x таким образом, что для любого такого узла ключи всех узлов в его левом поддереве меньше x, а ключи всех узлов в его правом поддереве больше x. Это достаточно необычный способ постановки меток на деревьях поиска. Для любого двоичного дерева с количеством узлов n и любого множества из n ключей существует *только один* способ постановки меток, который делает его двоичным деревом поиска. Допустимые способы постановки меток для двоичного дерева из трех узлов показаны на рис. 3.2.

3.4.1. Реализация двоичных деревьев

Кроме поля ключа каждый узел двоичного дерева имеет поля `left`, `right` и `parent`, которые содержат указатели на левый и правый дочерние узлы и на родительский узел соответственно. (Единственным узлом, указатель `parent` которого равен `NULL`, является корневой узел всего дерева.) Эти отношения показаны на рис. 3.3.

В листинге 3.6 приводится объявление типа для структуры дерева.

Листинг 3.6. Объявление типа для структуры дерева

```
typedef struct tree {
   item_type item;          /* Элемент данных */
   struct tree *parent;     /* Указатель на родительский узел */
   struct tree *left;       /* Указатель на левый дочерний узел */
   struct tree *right;      /* Указатель на правый дочерний узел */
} tree;
```

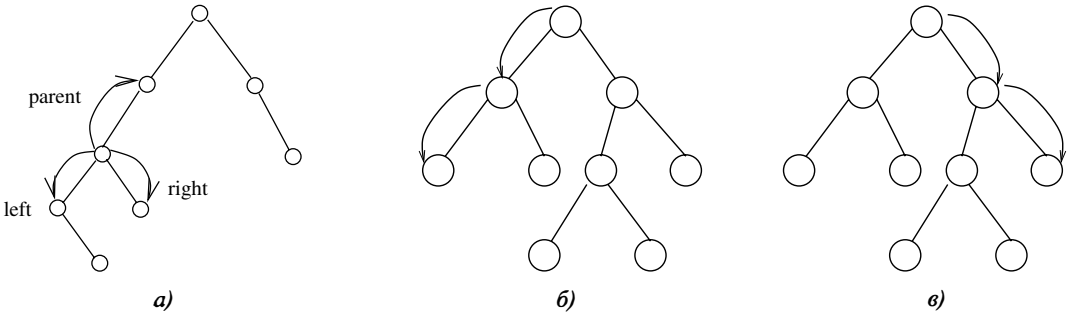


Рис. 3.3. Отношения в двоичном дереве *(а)*, поиск наименьшего *(б)* и наибольшего (в) элементов в двоичном дереве

Двоичные деревья поддерживают три основные операции: поиск, обход, вставку и удаление.

Поиск в дереве

Схема маркировки двоичного дерева поиска однозначно определяет расположение каждого ключа. Поиск начинается с корневого узла дерева. Если узел не содержит искомый ключ *x*, то в зависимости от того, меньше или больше значение искомого ключа значения ключа корневого узла, поиск продолжается в левом или правом дочернем узле соответственно. Этот алгоритм основан на том, что как левое, так и правое поддерево двоичного дерева сами являются двоичными деревьями. Реализация такого рекурсивного алгоритма поиска в двоичном дереве показана в листинге 3.7.

Листинг 3.7. Алгоритм рекурсивного поиска произвольного элемента в двоичном дереве

```
tree *search_tree(tree *l, item_type x)
{
   if (l == NULL) return(NULL);
   if (l->item == x) return(l);
   if (x < l->item)
      return( search_tree(l->left, x) );
   else
      return( search_tree(l->right, x) );
}
```

Время исполнения этого алгоритма равно $O(h)$, где h обозначает высоту дерева.

Поиск наименьшего и наибольшего элементов дерева

Реализация операции поиска наименьшего элемента требует понимания, каким образом этот элемент размещается в дереве. Согласно определению двоичного дерева, наименьший ключ должен находиться в левом поддереве корневого узла, т. к. значения всех ключей в этом дереве меньше, чем значение ключа корневого узла. Поэтому, как показано на рис. 3.3, *б*, наименьший элемент должен быть самым левым потомком корневого узла. Алгоритм поиска наименьшего элемента двоичного дерева показан в листинге 3.8.

Листинг 3.8. Поиск наименьшего элемента в двоичном дереве

```
tree *find_minimum(tree *t)
{
   tree *min; /* Указатель на наименьший элемент  */
   if (t == NULL) return(NULL);
   min = t;
   while (min->left != NULL)
      min = min->left;
   return(min);
}
```

Аналогично, наибольший элемент должен быть самым правым потомком корневого узла (см. рис. 3.3, *б*).

Обход дерева

Посещение всех узлов двоичного дерева является важной составляющей многих алгоритмов. Эта операция является частным случаем задачи обхода всех узлов и ребер графа, обсуждению которой посвящена *глава 5*.

Основным применением алгоритма обхода дерева является перечисление ключей всех узлов дерева. Природа двоичного дерева позволяет с легкостью перечислить ключи всех его узлов в отсортированном порядке. Согласно определению двоичного дерева, все ключи со значением меньшим, чем значение ключа корневого узла, должны находиться в левом поддереве корневого узла, а все ключи с большим значением — в правом. Таким образом, результатом рекурсивного посещения узлов согласно данному принципу является симметричный обход (in-order traversal) двоичного дерева. Реализация соответствующего алгоритма приведена в листинге 3.9.

Листинг 3.9. Рекурсивный алгоритм симметричного обхода двоичного дерева

```
void traverse_tree(tree *l)
{
   if (l != NULL) {
      traverse_tree(l->left);
      process_item(l->item);
      traverse_tree(l->right);
   }
}
```

Вставка элементов в дерево

В двоичном дереве T имеется только одно место, в которое можно вставить новый элемент x, где его потом можно будет найти в случае необходимости. Для этого нужно заменить указателем на вставляемый элемент указатель NULL, возвращенный неуспешным запросом поиска ключа k в дереве.

В реализации этой операции этапы поиска и вставки узла совмещаются с помощью рекурсии. Соответствующий алгоритм показан в листинге 3.10.

Листинг 3.10. Вставка узла в двоичном дереве

```
insert_tree(tree **l, item_type x, tree *parent)
{
   tree  *p;                    /* Временный указатель */
   if (*l == NULL) {
      p = malloc(sizeof(tree)); /* Выделение памяти для нового узла */
      p->item = x;
      p->left = p->right = NULL;
      p->parent = parent;
      *l = p;                   /* Указатель на родительский узел */
      return;
   }
   if (x < (*l)->item)
      insert_tree(&((*l)->left), x, *l);
   else
        insert_tree(&((*l)->right), x, *l);
}
```

Процедура `insert_tree` принимает три аргумента:

- указатель `l` на указатель, связывающий поддерево с остальной частью дерева;
- вставляемый ключ `x`;
- указатель `parent` на родительский узел, содержащий `l`.

Вставляемому узлу выделяется память и он интегрируется в структуру дерева, когда алгоритм находит указатель со значением `NULL`. Обратите внимание, что во время поиска соответствующему левому или правому указателю передается *указатель*, так что операция присваивания `*l = p;` интегрирует новый узел в структуру дерева.

После выполнения поиска за время $O(h)$ выделение памяти новому узлу и интегрирование его в структуру дерева выполняется за постоянное время.

Удаление элемента из дерева

Операция удаления элемента несколько сложнее, чем вставка, т. к. удаление узла означает, что нужно должным образом связать получившиеся поддеревья с общим деревом. Существуют три типа удаления, показанные на рис. 3.4.

Так как листовые узлы не имеют потомков, то их можно удалить, просто обнулив указатель на такой узел.

Удаление узла с одним потомком также не вызывает проблем. Такой узел имеет один родительский и один дочерний узел, и последний можно связать непосредственно с родительским узлом, не нарушая при этом симметричную схему размещения ключей дерева.

Но как удалить узел с двумя потомками? Нужно присвоить этому узлу ключ его непосредственного потомка в отсортированном порядке. Этот дочерний узел должен иметь наименьшее значение ключа в правом поддереве, а именно быть самым левым узлом в правом поддереве родительского дерева (`p`). Перемещение этого узла на место удаляемого узла дает в результате двоичное дерево с должным расположением ключей, а

также сводит нашу задачу к физическому удалению узла с, самое большее, одним потомком, а это задача уже была решена ранее.

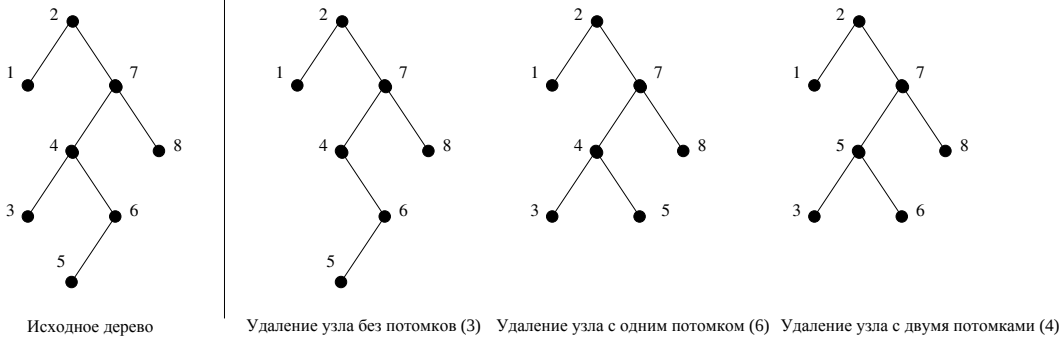


Рис. 3.4. Удаление узла дерева без потомков, с одним потомком и с двумя потомками

Полная реализация не приводится здесь по той причине, что она выглядит несколько устрашающе, хотя ее код логически следует из вышеизложенного описания.

Как определить сложность наихудшего случая? Для каждого удаления требуется, самое большее, две операции поиска, каждая из которых выполняется за время $O(h)$, где h — высота дерева, плюс постоянные затраты времени на манипуляцию указателями.

3.4.2. Эффективность двоичных деревьев поиска

Все три словарные операции, реализованные посредством двоичных деревьев поиска, исполняются за время $O(h)$, где h — высота дерева. Дерево имеет самую меньшую высоту, на которую мы можем надеяться, когда оно полностью сбалансированно; тогда высота равна $h = \lceil \log n \rceil$. Это очень хорошо, но дерево должно быть идеально сбалансированным.

Наш алгоритм вставки помещает каждый новый элемент в лист дерева, в котором он должен находиться. Вследствие этого форма (и, что более важно, высота) дерева зависит от порядка вставки ключей.

К сожалению, при построении дерева методом вставок могут происходить неприятные события, т. к. структура данных не контролирует порядок вставок. Например, посмотрим, что случится, если вставлять ключи в отсортированном порядке. Здесь операции *insert(a)*, *insert(b)*, *insert(c)*, *insert(d)* и т. д. дадут нам тонкое длинное дерево, в котором используются только правые узлы.

Таким образом, высота двоичных деревьев может лежать в диапазоне от $\lg n$ до n. Но какова средняя высота дерева? Что именно мы считаем средней высотой? Вопрос будет четко определен, если мы будем считать равновероятными каждое из $n!$ возможных упорядочиваний вставки и возьмем среднее их значение. В таком случае нам повезло, т. к. с высокой вероятностью высота получившегося дерева будет $O(\log n)$. Это будет показано в *разделе 4.6*.

Этот пример наглядно демонстрирует мощь *рандомизации*. Часто можно разработать простые алгоритмы, с высокой вероятностью имеющие хорошую производительность. Далее мы увидим, что подобная идея лежит в основе алгоритма быстрой сортировки quicksort.

3.4.3. Сбалансированные деревья поиска

Произвольные деревья поиска *обычно* дают хорошие результаты. Но если нам не повезет с порядком вставок, то в наихудшем случае мы может получить дерево линейной высоты. Мы не можем прямо контролировать этот наихудший случай, т. к. мы должны создавать дерево в ответ на запросы, исходящие от пользователя.

Но ситуацию можно улучшить с помощью процедуры вставки/удаления, которая после каждой вставки слегка корректирует дерево, удерживая его как можно более сбалансированным, так что максимальная высота дерева будет логарифмической. Существуют сложные структуры данных в виде сбалансированных двоичных деревьев поиска, которые гарантируют, что высота дерева всегда будет $O(\log n)$. Вследствие этого время исполнения любой из словарных операций (вставка, удаление, запрос) будет равняться $O(\log n)$. Реализация структур данных сбалансированных деревьев, таких как красно-черные и косые деревья, обсуждается в *разделе 12.1*.

С точки зрения разработки алгоритмов важно знать, что такие деревья существуют и что их можно использовать в качестве черного ящика, чтобы реализовать эффективный словарь. При расчете трудоемкости словарных операций для анализа алгоритма можно предположить, что сложность сбалансированного двоичного дерева в наихудшем случае будет подходящей мерой.

Подведение итогов

Выбор неправильной структуры данных для поставленной задачи может иметь катастрофические последствия для производительности. Определение самой лучшей структуры данных не настолько критично, т. к. возможно существование нескольких вариантов структур, имеющих примерно одинаковую производительность.

Остановка для размышлений. Использование сбалансированных деревьев поиска

ЗАДАЧА. Нужно прочитать n чисел и вывести их в отсортированном порядке. Допустим, что у нас имеется сбалансированный словарь, который поддерживает операции *search*, *insert*, *delete*, *minimum*, *maximum*, *successor* и *predecessor*, время исполнения каждой из которых равно $O(\log n)$.

1. Отсортируйте данные за время $O(n \log n)$, используя только операции вставки и симметричного обхода.
2. Отсортируйте данные за время $O(n \log n)$, используя только операции *minimum*, *successor* и *insert*.
3. Отсортируйте данные за время $O(n \log n)$, используя только операции *minimum*, *insert*, *delete* и *search*.

Решение. В первой задаче мы можем выполнять операции вставки и симметричного обхода. Это позволяет нам создать дерево поиска, вставив все n элементов, после чего выполнить обход дерева, чтобы отсортировать элементы.

```
Sort1()
  initialize-tree(t)
  while (not EOF)
    read(x);
    insert(x,t)
  traverse(t)
```

```
Sort2()
  initialize-tree(t)
  while (not EOF)
    read(x);
    insert(x,t);
  y = minimum(t)
  while (y≠NULL) do
    print(y→item)
    y = successor(y,t)
```

```
Sort3()
  initialize-tree(t)
  while (not EOF)
    read(x);
    insert(x,t);
  y = minimum(t)
  while (y≠NULL) do
    print(y→item)
    delete(y,t)
    y = minimum(t)
```

Во второй задаче, после создания дерева, мы можем использовать операции *minimum* и *maximum*. Для сортировки мы можем выполнить обход дерева, начав с наименьшего элемента и последовательно выполняя операцию поиска следующего элемента.

В третьей задаче мы не имеем операции поиска следующего элемента, но можем выполнять удаление. Для сортировки мы можем выполнить обход дерева, опять начав с наименьшего элемента и последовательно выполняя операцию удаления следующего наименьшего элемента.

Каждый из этих алгоритмов выполняет линейное количество операций с логарифмическим временем исполнения, что дает общее время его исполнения $O(n\log n)$. Основной подход к использованию сбалансированных двоичных деревьев заключается в том, чтобы рассматривать их как "черные ящики".

3.5. Очереди с приоритетами

Многие алгоритмы обрабатывают элементы в определенном порядке. Допустим, например, что нужно запланировать выполнение работ согласно их относительной важности. Для этого работы нужно сначала отсортировать по важности, после чего оценить их в этом отсортированном порядке.

Очереди с приоритетами предоставляют разработчику больше гибкости, чем обычная сортировка, т. к. они позволяют вводить новые элементы в систему в произвольном месте. Намного эффективнее вставить новый элемент в нужное место в очереди с приоритетами, чем сортировать заново все элементы при каждой вставке.

Базовая очередь с приоритетами поддерживает три основные операции:

- *insert(Q,x)* — вставляет в очередь с приоритетами Q элемент x с ключом k;
- *find-minimum(Q)* и *find-maximum(Q)* — возвращает указатель на элемент с наименьшим (наибольшим) значением ключа в очереди с приоритетами Q;
- *delete-minimum(Q)* и *delete-maximum(Q)* — удаляет из очереди Q элемент с наименьшим (наибольшим) значением ключа.

Очереди с приоритетами точно моделируют многие природные процессы. Например, люди с неустроенной личной жизнью мысленно (или открыто) поддерживают очередь

с приоритетами из возможных кандидатов на роль постоянного партнера. Впечатления о новом знакомстве отображаются непосредственно на шкале привлекательности. Привлекательность здесь играет роль ключа для создания нового элемента в очереди с приоритетами. Выбор партнера — это процесс, включающий в себя извлечение наиболее привлекательного элемента из очереди (выполняется операция *find-maximum*), свидание с ним для уточнения степени привлекательности и вставку обратно в очередь с приоритетами, возможно, в другое место с новым значением привлекательности.

Подведение итогов

Словари и очереди с приоритетами позволяют создавать алгоритмы с аккуратной структурой и хорошей производительностью.

Остановка для размышлений. Построение базовых очередей с приоритетами

ЗАДАЧА. Определить временную сложность в наихудшем случае трех основных операций очереди с приоритетами (вставка элемента, поиск наименьшего элемента и поиск наибольшего элемента) для следующих базовых структур данных:

- неотсортированного массива;
- отсортированного массива;
- сбалансированного двоичного дерева.

Решение. В реализации этих трех операций присутствуют некоторые тонкости, даже при использовании такой простой структуры данных, как неотсортированный массив. В словаре на основе неотсортированного массива (см. *раздел 3.3*) операции вставки и удаления выполняются за постоянное время, а операции поиска произвольного и наименьшего элементов — за линейное время. Операцию удаления наименьшего элемента *(delete-minimum)*, выполняемую за линейное время, можно составить из последовательности операций *find-minimum*, *search* и *delete*.

В отсортированном массиве операции вставки и удаления можно выполнить за линейное время, а найти наименьший элемент — за постоянное. Но любые удаления из очереди с приоритетами затрагивают только наименьший элемент. После сортировки массива в обратном порядке (наибольший элемент вверху) наименьший элемент будет самым последним элементом массива. Чтобы удалить последний элемент, не нужно перемещать никакие элементы, а только уменьшить на единицу количество оставшихся элементов n, вследствие чего операция удаления наименьшего элемента может выполняться за постоянное время.

Все это правильно, но, как можно видеть из следующей таблицы, операцию поиска наименьшего значения, выполняемую за постоянное время, можно реализовать для каждой структуры данных.

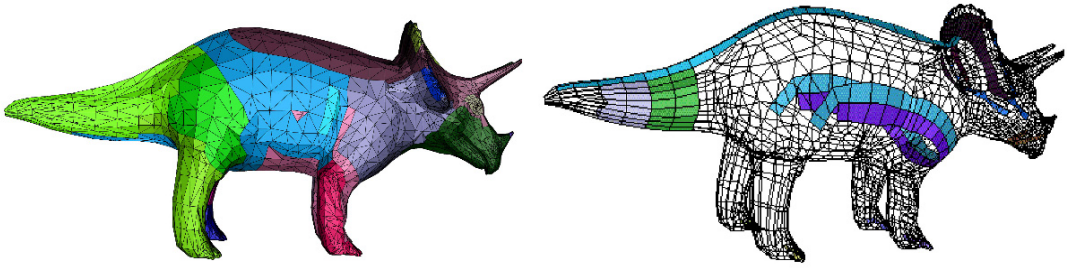
Операция	Неотсортированный массив	Отсортированный массив	Сбалансированное дерево
insert(Q,x)	$O(1)$	$O(n)$	$O(\log n)$
find-minimum(Q)	$O(1)$	$O(1)$	$O(1)$
delete-minimum(Q)	$O(n)$	$O(1)$	$O(\log n)$

Вся хитрость заключается в использовании дополнительной переменной для хранения указателя на наименьший элемент в каждой из этих структур, что позволяет просто возвратить это значение в любое время при получении запроса на поиск этого элемента. Обновление этого указателя при каждой вставке не составляет труда — он обновляется тогда и только тогда, когда вставленный элемент меньше, чем текущий наименьший элемент. Но что будет в случае удаления наименьшего элемента? Мы можем удалить текущий наименьший элемент, после чего найти новый наименьший элемент и зафиксировать его в этом качестве до тех пор, пока он тоже не будет удален. Настоящая операция поиска наименьшего элемента занимает линейное время для неотсортированных массивов и логарифмическое время для деревьев и поэтому затраты на ее выполнение можно включить в затраты на выполнение каждого удаления.

Очереди с приоритетами являются очень полезными структурами данных. Особенно изящная реализация очереди с приоритетами — пирамида — рассматривается в контексте задачи сортировки в *разделе 4.3*. Кроме этого, в *разделе 12.2* дается полный набор реализаций очереди с приоритетами.

3.6. История из жизни. Триангуляция

Применяемые в компьютерной графике геометрические модели обычно разбивают на множество небольших треугольников, как показано на рис. 3.5.



а) *б)*

Рис. 3.5. Триангуляционная модель динозавра (*а*), использование нескольких полос треугольников (*б*)

Для прорисовывания и закрашивания треугольников применяются высокопроизводительные движки визуализации (rendering engine), работающие на специализированном аппаратном обеспечении. Скорость работы этого аппаратного обеспечения такая высокая, что единственным узким местом в системе являются затраты на ввод в него триангулированной структуры.

Хотя каждый треугольник можно описать, указав три его вершины, альтернативное представление более эффективно. Вместо того чтобы определять каждый треугольник по отдельности, мы соединяем их в *полосы* смежных треугольников и обрабатываем их, перемещаясь вдоль этой полосы. Поскольку в этом случае каждый треугольник имеет две общие вершины со своим соседом, то мы экономим на расходах по передаче данных о двух вершинах и другой сопутствующей информации. Для однозначного описания треугольников в рендерере треугольной сетки OpenGL принимается соглашение, что все повороты чередуются влево-вправо, как показано на рис. 3.6.

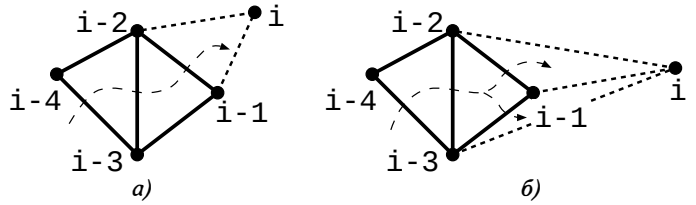


Рис. 3.6. Разбиение треугольной сетки на полосы: с чередующимися влево-вправо поворотами (*а*), с произвольным и поворотами (*б*)

Задачу определения наименьшего количества полос, которые покрывают все треугольники в сетке, можно рассматривать как задачу графов. В таком графе каждый треугольник сетки представляется вершиной, а смежные треугольники представляются ребром между соответствующими вершинами. Такое представление в виде *двойственного графа* содержит всю информацию, необходимую для разбиения треугольной сетки на полосы треугольников (см. *раздел 15.12*).

После получения двойственного графа можно было приступать к работе. Мы хотели разбить множество вершин на минимально возможное количество путей (или полос). Если бы мы смогли объединить их в один путь, то это бы означало, что мы нашли гамильтонов путь, т. е. путь, содержащий каждую вершину графа ровно один раз. Так как поиск гамильтонова пути является NP-полной задачей (см. *раздел 16.5*), мы знали, что нам не стоит искать алгоритм точного решения, а концентрироваться на эвристическом алгоритме приблизительного решения.

Самый простой эвристический алгоритм для полосового покрытия начинает работу с произвольного треугольника и двигается направо, пока не дойдет до границы объекта или до уже посещенного треугольника. Достоинством этого эвристического алгоритма являются его быстрота и простота, но при этом нет гарантии, что удастся найти наименьший набор ориентированных слева направо полос для данной триангуляции. А вот получение такого набора с помощью "жадного" эвристического алгоритма более вероятно. "Жадные" алгоритмы выбирают оптимальные возможные решения на каждом этапе в надежде, что и конечное решение также будет оптимальным. При выполнении триангуляции "жадный" алгоритм определяет начальный треугольник самой длинной полосы слева направо и отделяет ее.

Но сам факт использования "жадного" алгоритма не гарантирует наилучшее возможное решение, т. к. первая отделенная полоса может разбить много потенциальных полос, которые можно было бы использовать в дальнейшем. Тем не менее, хорошим практическим правилом для получения оптимального решения будет использование "жадного" алгоритма. Так как отделение самой длинной полосы оставляет самое меньшее количество треугольников для создания последующих полос, то "жадный" эвристический алгоритм является более производительным, чем простой эвристический алгоритм.

Но сколько времени понадобится этому алгоритму, чтобы найти следующую самую длинную полосу треугольников? Пусть k — длина прохода, начинающегося в средней вершине. Используя самую простую реализацию алгоритма, можно выполнять проход от каждой из n вершин, что позволит найти самую длинную оставшуюся полосу за время $O(kn)$. Повторение такого прохода для каждой из почти n/k извлекаемых полос

дает нам время исполнения $O(n)$, что неприемлемо медленно для типичной модели, состоящей из 20 тысяч треугольников.

Как можно улучшить это время? Интуитивно кажется, что неэкономично повторять проход от каждого треугольника после удаления лишь одной полосы. Можно просто хранить информацию о длине всех возможных будущих полос в какой-либо структуре данных. Но при удалении каждой полосы необходимо обновить информацию о длине всех других затронутых этим удалением полос. Эти полосы будут укорочены, т. к. они проходят через треугольник, которого больше нет. Такая структура данных будет иметь характеристики двух структур:

- *Очередь с приоритетами.* Так как мы многократно повторяем процесс определения самой длинной оставшейся полосы, то нам нужна очередь с приоритетами для хранения полос, упорядоченных по длине. Следующая удаляемая полоса всегда находится в начале очереди. Наша очередь с приоритетами должна была разрешать понижение приоритета произвольных элементов в очереди при каждом обновлении информации о длине полос, чтобы мы знали, какие треугольники были удалены. Так как длина всех полос ограничивалась довольно небольшим целым числом (аппаратные возможности не позволяли иметь в полосе больше 256 вершин), мы использовали ограниченную по высоте очередь с приоритетами (массив корзин, показанный на рис. 3.7 и рассматриваемый в *разделе 12.2*). Обычная куча также была бы вполне приемлемой.

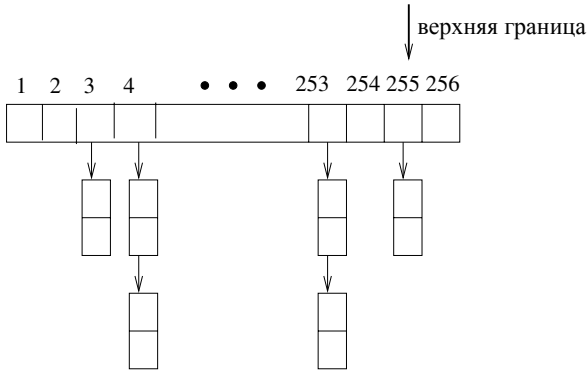


Рис. 3.7. Ограниченная по высоте очередь с приоритетами для полос треугольников

Для обновления элемента очереди, связанного с треугольником, требовалось быстро находить его. Это означало, что нам нужен словарь.

- *Словарь.* Для каждого треугольника в сетке нам нужно было знать его расположение в очереди. Это означало, что в словаре нужно было иметь указатель на каждый треугольник. Объединив этот словарь с очередью с приоритетами, мы создали структуру данных, способную поддерживать широкий диапазон операций.

Хотя возникали и другие трудности, например, необходимость найти способ быстрого вычисления новой длины у полос, затронутых удалением полосы, главный метод повышения производительности состоял в применении очереди с приоритетами. Использование этой структуры данных улучшило время исполнения на несколько порядков.

Из табл. 3.3 видно, насколько производительность "жадного" алгоритма выше, чем у простого.

Таблица 3.3. Сравнение производительности "жадного" и простого алгоритмов для нескольких типов сетки треугольников

Название модели	Количество треугольников	Затраты простого алгоритма	Затраты "жадного" алгоритма	Время "жадного" алгоритма
Ныряльщик	3 798	8 460	4 650	6,4 сек
Головы	4 157	10 588	4 749	9,9 сек
Каркас	5 602	9 274	7 210	9,7 сек
Барт Симпсон	9 654	24 934	11 676	20,5 сек
Космический корабль Enterprise	12 710	29 016	13 738	26,2 сек
Тор	20 000	40 000	20 200	272,7 сек
Челюсть	75 842	104 203	95 020	136,2 сек

Во всех случаях "жадный" алгоритм выдавал набор полос меньшей стоимости в смысле суммарной длины полос. Экономия составляла от 10 до 50%, что было просто замечательно, т. к. максимально возможное улучшение (уменьшение количества вершин каждого треугольника с трех до одной) позволяет сэкономить не более 66,6%.

Результатом реализации "жадного" алгоритма со структурой данных в виде очереди с приоритетами было время исполнения программы $O(n \cdot k)$, где n — количество треугольников, k — длина средней полосы. Вследствие этого обработка тора, состоящего из небольшого количества очень длинных полос, занимала больше времени, что обработка челюсти, хотя количество треугольников в последней фигуре было втрое больше.

Из этой истории можно извлечь несколько уроков. Во-первых, при работе с достаточно большим набором данных только алгоритмы с линейным или почти линейным временем исполнения (скажем, $O(n\log n)$) могут быть достаточно быстрыми. Во-вторых, выбор правильной структуры данных часто является ключевым фактором для уменьшения временной сложности алгоритма в такой степени. И в-третьих, применение "жадного" эвристического алгоритма может значительно улучшить производительность по сравнению с применением простого алгоритма. Насколько лучше будет производительность, можно определить только экспериментальным путем.

3.7. Хэширование и строки

Хэш-таблицы предоставляют *очень* практичный способ реализации словарей. В них используется то обстоятельство, что поиск элемента массива по индексу выполняется за постоянное время. Хэш-функция соотносит набор ключей с целочисленными значениями. Мы будем использовать значение нашей хэш-функции в качестве индекса массива и записывать элемент в этой позиции.

Сначала хэш-функция соотносит каждый ключ с большим целым числом. Пусть значение a представляет размер алфавита, используемого для создания строки S. Пусть char(c) будет функцией, которая однозначно отображает каждый символ алфавита в целое число в диапазоне от 0 до α – 1. Функция $H(S) = \sum_{i=0}^{|S|-1} \alpha^{|S|-(i+1)} \times char(S_i)$ однозначно отображает каждую строку в (большое) целое число, рассматривая символы строки как "цифры" системы счисления с основанием α.

В результате получатся уникальные идентификационные числа, но они будут настолько большими, что очень быстро превысят количество ячеек в нашей хэш-таблице (обозначаемое m). Это значение необходимо уменьшить до целого числа в диапазоне от 0 до m – 1, для чего выполняется операция получения остатка от деления $H(S) \bmod m$. Здесь применяется тот же самый принцип, что и в колесе рулетки. Шарик проходит долгий путь, обходя $\lfloor H(S)/m \rfloor$ раз колесо окружностью m, пока не остановится в произвольной ячейке, размер которой составляет незначительную часть пройденного пути. Если выбрать размер таблицы с достаточной тщательностью (в идеале, число m должно быть большим простым числом, не слишком близким к 2^i – 1), то распределение полученных хэш-значений будет довольно равномерным.

3.7.1. Коллизии

Независимо от того, насколько хороша наша хэш-функция, время от времени она будет отображать два разных ключа в одно хэш-значение, и нужно быть готовым к подобной ситуации. Самым легким способом разрешения таких коллизий является применение цепочек. Для этого хэш-таблица реализуется в виде массива из m связных списков, как показано на рис. 3.8.

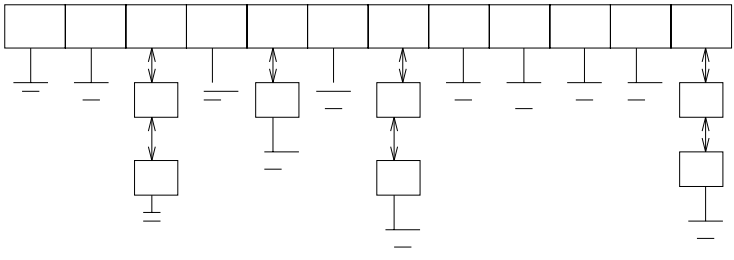


Рис. 3.8. Разрешение коллизий при помощи цепочек

Список с порядковым номером i содержит все элементы, хэшированные в одно и то же значение i. Таким образом, операции поиска, вставки и удаления сводятся к соответствующим операциям в связных списках. Если n ключей распределены в таблице равномерно, то каждый список будет содержать приблизительно n/m элементов, делая их размер постоянным при $m \approx n$.

Метод цепочек — самый естественный, но он требует значительных объемов памяти для хранения указателей. Эту память можно было использовать, чтобы сделать таблицу больше, а "списки", соответственно, меньше.

Другой способ разрешения коллизий хэш-значений называется *открытой адресацией*. При использовании этого метода хэш-таблица реализуется в виде массива ключей (а не корзин), каждый из которых инициализирован значением NULL, как показано на рис. 3.9.

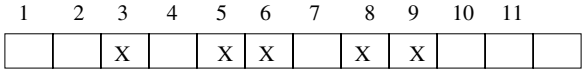


Рис. 3.9. Разрешение коллизий методом открытой адресации

При вставке ключа выполняется проверка, свободна ли требуемая ячейка. Если свободна, то вставка выполняется. В противном случае нам нужно найти другое место, куда вставить данный ключ. Самый простой подход к определению альтернативного места для вставки называется *последовательным исследованием* (sequential probing). Этот метод заключается в том, что последовательно исследуются следующие ячейки таблицы, пока не будет найдена свободная, в которую и выполняется вставка. Если таблица не слишком заполнена, то последовательность смежных занятых ячеек должна быть довольно небольшой, и свободная ячейка *должна* находиться лишь в нескольких позициях от требуемой.

Теперь для поиска определенного ключа нужно взять соответствующее хэш-значение и проверить, является ли его значение тем, какое нам нужно. Если да, то поиск заканчивается. В противном случае поиск продолжается в последовательности смежных ячеек, пока не будет найдено требуемое значение.

Удаление из хэш-таблицы с открытой адресацией довольно сложная процедура, т. к. удаление одного элемента может разорвать цепочку вставок, вследствие чего некоторые элементы последовательности больше не будут доступны. Не остается иного выхода, кроме как выполнить повторную вставку всех элементов, следующих за удаленным.

В обоих методах разрешения коллизий требуется время $O(m)$ для инициализации нулями хэш-таблицы из m элементов до того, как можно будет выполнять первую вставку. Обход всех элементов в таблице цепочек занимает время $O(n + m)$, т. к. нам нужно просмотреть все m корзин в процессе поиска, даже если в действительности таблица содержит небольшое количество вставленных элементов. Для таблицы с открытой адресацией это время равно $O(m)$, т. к. значение n может быть равным, самое большее, значению m.

При использовании метода цепочек для разрешения коллизий в хэш-таблице из m элементов словарные операции для n элементов можно реализовать со следующим и показателями времени исполнения в ожидаемом и наихудшем случае:

Операция	Ожидаемое время исполнения	Время исполнения в наихудшем случае
search(L,k)	$O(n/m)$	$O(n)$
insert(L,x)	$O(1)$	$O(1)$

(окончание)

Операция	Ожидаемое время исполнения	Время исполнения в наихудшем случае
delete(L,x)	$O(1)$	$O(1)$
successor(L,x)	$O(n + m)$	$O(n + m)$
predecessor(L,x)	$O(n + m)$	$O(n + m)$
minimum(L)	$O(n + m)$	$O(n + m)$
maximum(L)	$O(n + m)$	$O(n + m)$

С практической точки зрения, хэш-таблица часто является самым лучшим типом структуры данных для реализации словаря. Но область применения хэширования гораздо шире, чем просто создание словарей, в чем мы скоро убедимся.

3.7.2. Эффективный метод поиска строк посредством хэширования

Строки представляют собой последовательности символов, порядок размещения которых имеет значение, т. к. строка *АЛГОРИФМ* (устаревший вариант слова "алгоритм") отличается от строки *ЛОГАРИФМ*. Текстовые строки являются основным типом данных для многих компьютерных приложений, от синтаксического разбора и компилирования в языках программирования до поисковых систем Интернета и анализаторов биологических рядов.

Основной структурой данных для представления строк является массив символов, что обеспечивает постоянное время доступа к i-му символу строки. Для обозначения конца строки требуется хранить вместе с ней определенную вспомогательную информацию — специальный символ "конец строки" или (что, возможно, полезнее) количество символов в строке.

Основной строковой операцией является поиск подстроки в строке. В следующей задаче и ее решении приводится демонстрация такой операции.

ЗАДАЧА. Найти подстроку в строке.

Вход. Текстовая строка t и строка для поиска p.

Выход. Определить, содержит ли строка t подстроку p, и если содержит, то в каком месте?

Самый простой алгоритм поиска подстроки в строке циклически накладывает подстроку p на строку t, перемещаясь вправо на одну позицию в строке t, пока последний символ подстроки p не совмещается с последним символом строки t, и проверяет на совпадение каждого символа подстроки p с соответствующим символом строки t. Как было показано в *разделе 2.5.3*, время исполнения такого алгоритма равно $O(nm)$, где $n = |t|$ и $m = |p|$.

Это квадратичный предел времени исполнения в наихудшем случае. Но существуют алгоритмы и с линейным временем исполнения в наихудшем случае, но более слож-

ные. Такие алгоритмы подробно рассматриваются в *разделе 18.3*. Здесь же мы рассмотрим алгоритм поиска подстроки с ожидаемым линейным временем исполнения, называющийся алгоритмом Рабина-Карпа и основывающийся на использовании хэширования. Допустим, что мы вычислим определенную хэш-функцию от строки-образца p и от подстроки длиной m символов, начинающейся в позиции i текста t. Очевидно, что если эти две строки одинаковы, то и их хэш-значения должны быть одинаковыми. Если же строки разные, то и их хэш-значения *почти наверняка* будут разными. Одинаковые хэш-значения для разных строк должны встречаться настолько редко, что мы вполне можем позволить себе потратить время $O(m)$ на явную проверку идентичности строк при совпадении хэш-значений.

Это сводит сложность поиска подстроки к $n-m+2$ вычислениям хэш-значений ($n-m+1$ хэш-значений для окон-подстрок в тексте t, плюс одно хэш-значение для строки-образца p), плюс несколько операций проверки с временем исполнения $O(m)$, количество которых *должно быть* очень небольшим. Но проблема состоит в том, что вычисление хэш-функции для строки из m символов занимает время $O(m)$, а $O(n)$ таких вычислений, по-видимому, опять дают нам общее время исполнения алгоритма $O(mn)$ в качестве оценки сложности.

Давайте подробнее рассмотрим применение ранее определенной хэш-функции к m символам строки S, начиная с позиции j: $P(S,j)=\sum_{i=1}^{m-1}\alpha^{m-(i+1)}\times char(s_{i+j})$.

Что изменится, если сейчас мы попробуем вычислить значение $H(S,j+1)$, т. е. — хэш-значение следующей подстроки длиной в m символов? Обратите внимание, что $m-1$ символов одинаковы в обеих подстроках, хотя количество умножений на α различается у них на единицу. Выполнив некоторые алгебраические преобразования, видим, что:

$$H(S,j+1)=\alpha(P(S,j)-\alpha^{m-1}char(s_j))+char(s_{j+m})$$

Иными словами, когда нам известно хэш-значение подстроки, начинающейся в позиции j, то мы можем узнать хэш-значение подстроки с позиции $(j+1)$, выполнив две операции умножения, одну операцию сложения и одну операцию вычитания. Для этого требуется постоянное время (значение α^{m-1} можно вычислить один раз, а затем использовать его для вычисления всех хэш-значений). Такой подход годится даже для вычисления $H(S,j) \bmod M$, где M является достаточно большим простым числом, таким образом ограничивая величину наших хэш-значений (самое большее M) даже для длинных строк-образцов.

Алгоритм Рабина-Карпа является хорошим примером рандомизированного алгоритма (когда значение M выбирается каким-либо произвольным способом). Время исполнения алгоритма не обязательно будет равно $O(n+m)$, т. к. нам может не повезти и мы будем регулярно получать конфликтные хэш-значения для ложных совпадений. Тем не менее, у нас неплохие шансы на удачу. Если хэш-функция возвращает значения равномерно в диапазоне от 0 до $M-1$, то вероятность ложной коллизии хэш-значений должна быть $1/M$. Это вполне приемлемо: если $M \approx n$, то должна быть только одна ложная коллизия на каждую строку, а если $M \approx n^k$ при $k \geq 2$, то существует очень высокая вероятность, что мы никогда не получим ложных коллизий.

3.7.3. Выявление дубликатов с помощью хэширования

Основополагающей идеей хэширования является представление большого объекта (будь то ключ, строка или подстрока) посредством одного числа. Цель состоит в том, чтобы представлять большой объект другим объектом, которым можно манипулировать за постоянное время, при этом вероятность представления двух разных больших объектов одним и тем же числовым значением должна быть сравнительно невысокой.

Кроме ускорения поиска, хэширование находит многие другие разнообразные хитроумные применения. Я однажды слышал, как Уди Манбер (Udi Manber), в то время руководитель исследовательских работ в Yahoo, рассказывал об алгоритмах, используемых в его компании. По его словам, наиболее важными алгоритмами, используемыми в Yahoo, были хэширование, хэширование и снова хэширование.

Рассмотрим следующие задачи, имеющие красивые решения посредством хэширования.

- *Отличается ли данный документ от других в большом собраний документов?* Поисковый механизм, который уже накопил громаднейшую базу данных из n документов, обработал еще одну веб-страницу. Как он определит, добавит ли она нечто новое в базу данных или только продублирует уже имеющуюся информацию?

  Для большого собрания документов будет весьма неэффективно сравнивать новый документ D со всеми имеющимися. Но мы можем преобразовать документ D в целое число с помощью хэш-функции и сравнить его с хэш-кодами документов, уже имеющихся в базе данных. Только в случае коллизии хэш-значений документ D может быть возможным дубликатом. Так как вероятность ложных коллизий низка, то мы без больших затрат можем явно сравнить те несколько документов с одинаковыми хэш-кодами.

- *Не является ли данный документ плагиатом?* Ленивый студент копирует документ из Интернета как свою курсовую работу, слегка изменив его в некоторых местах. "Интернет большой, — ухмыляется он. — Никто ничего не узнает".

  Это гораздо более трудная задача, чем предыдущая. Если в документ добавлен, удален или изменен всего лишь один символ, то его хэш-код будет совершенно другим. Таким образом подход с использованием хэш-кода, применяемый для решения предыдущей задачи, не годится для решения этой более общей задачи.

  Но мы можем создать хэш-таблицу всех перекрывающихся окон (подстрок) длиной w символов для всех документов в базе данных. Любое совпадение хэш-кодов означает, что оба документа, вероятно, содержат одинаковую подстроку длиной w символов, что можно исследовать более подробно. Значение w нужно выбрать достаточно длинным, чтобы минимизировать вероятность случайного совпадения хэш-кодов.

  Самым большим недостатком этой схемы является то обстоятельство, что объем хэш-таблицы становится таким же большим, как и объем самих документов. Оставив для каждого документа небольшое, но удачно определенное подмножество хэш-кодов (состоящее, например, из чисел, кратных 100), мы с большой вероятностью сможем обнаруживать копии достаточно длинных строк.

- *Как убедиться в том, что файл не был изменен?* На закрытых торгах каждая сторона подает свое предложение цены, которое неизвестно никакой другой стороне. В назначенное время цены оглашаются и сторона, предложившая наивысшую цену, объявляется выигравшей торги. Если какой-либо недобросовестный участник торгов знает предложения других сторон, то он может предложить цену, немного превышающую максимальную цену, предложенную оппонентами, и выиграть торги. Одним из способов получить информацию о предложениях других сторон будет взлом компьютера, на котором хранится эта информация.

  Для предотвращения такого развития событий можно потребовать предоставление хэш-кода предложения до даты открытия предложений, с тем, чтобы само предложение было представлено после этой даты. После определения победителя его предложение хэшируется и полученное хэш-значение сравнивается с хэш-значением, предоставленным ранее. Такие методы *криптографического хэширования* позволяют удостовериться, что предоставленный сегодня файл такой же, как и первоначальный, т. к. любые изменения в файле отразятся в измененном хэш-коде.

Хотя наихудшие случаи любого алгоритма с хэшированием способны привести в смятение, при правильно подобранной хэш-функции мы можем с уверенностью ожидать хорошие результаты. Хэширование является основополагающей идеей рандомизированных алгоритмов, позволяющей получить линейное время исполнения алгоритма, по сравнению с $\Theta(n\log n)$ или даже $\Theta(n^2)$ в наихудшем случае.

3.8. Специализированные структуры данных

Все рассматриваемые до этого времени основные структуры данных представляют обобщенные наборы элементов, облегчающие доступ к данным. Эти структуры данных хорошо известны большинству программистов. Менее известны структуры данных для представления специализированных типов объектов, таких как точки в пространстве, строки и графы.

Принципы создания этих структур данных те же, что и для основных типов структур. Имеется набор основных многократно исполняемых операций. Нам нужна структура данных, поддерживающая достаточно эффективное исполнение этих операций. Эти специализированные структуры данных являются важными для создания высокопроизводительных алгоритмов для работы с графами и геометрическими объектами, поэтому нужно знать об их существовании. Далее приводится краткое описание нескольких таких специализированных структур данных. Подробно они обсуждаются при использовании в решениях в каталоге задач.

- *Строки.* Строки обычно представляются в виде массивов символов, возможно, с использованием специального символа для обозначения конца строки. Примером таких структур данных является суффиксное дерево/массив, в котором выполняется предварительная обработка строк для ускорения операции поиска подстрок. Подробности см. в *разделе 12.3*.
- *Геометрические структуры данных.* Геометрические данные обычно представляют собой коллекцию точек и областей данных. Плоскость можно представить в виде многоугольника с границей, состоящей из цепочки отрезков. Многоугольник можно

представить с помощью массива точек (v_1, ..., v_m, v_1), где (v_i, v_{i+1}) является сегментом границы многоугольника. В пространственных структурах данных, таких как kd-деревья, точки и области организованы по их геометрическому расположению для ускорения поиска. Подробности см. в *разделе 12.6*.

- *Графы.* Графы обычно представляются посредством матриц или списков смежных вершин графа. Выбор конкретного типа представления может существенно повлиять на структуру конечных алгоритмов для работы с графом, как рассказывается в *главе 6* и в *разделе 12.4*.
- *Множества.* Подмножества элементов обычно представляются посредством словаря для обеспечения быстроты поиска. Кроме того, могут использоваться *двоичные векторы*, которые представляют собой булевы массивы, в которых установленный i-й бит означает, что i является членом подмножества. Структуры данных для манипулирования множествами представлены в *разделе 12.5*. Структура данных, используемая при работе с разбиениями множества, рассматривается в *разделе 6.1.3*.

3.9. История из жизни. Геном человека

В геноме человека закодирована вся необходимая для создания человека информация. Проект по его расшифровке, называемый *геномом человека*, уже оказал огромное влияние на развитие современной медицины и молекулярной биологии. Алгористы также заинтересовались проектом генома человека, и на то были свои причины.

- Последовательности ДНК можно точно представить в виде строк алфавита из четырех символов — A, C, T и G. Нужды биологов возродили интерес к старым алгоритмическим задачам, таким как поиск подстрок (см. *раздел 18.3*), а также создали новые задачи, такие как поиск самой короткой общей подстроки (см. *раздел 18.9*).
- Последовательности ДНК являются *очень* длинными строками. Длина генома человека составляет приблизительно три миллиарда базовых пар (или символов). Такой большой размер входного экземпляра задачи означает, что асимптотический анализ сложности алгоритма совершенно необходим в биологических приложениях.
- В геномику вливается достаточно большой объем денег, что вызывает у специалистов в компьютерной области желание получить свою долю.

Лично меня в вычислительной биологии заинтересовал метод, предложенный для секвенирования ДНК и названный *секвенированием путем гибридизации* (sequencing by hybridization, SBH). Для его реализации набор проб (коротких нуклеотидных последовательностей) организуется в массив, создавая *секвенирующий чип*. Каждая из этих проб определяет наличие строки пробы в виде подстроки в целевой ДНК. Теперь можно выполнить секвенирование целевой ДНК на основе ограничений, определяющих, какие строки являются подстроками целевой ДНК.

По данному набору всех подстрок длиной k строки S мы должны были идентифицировать все строки длиной $2k$, возможные подстроки S. Предположим, нам известно, что строки AC, CA и CC являются двухсимвольными подстроками строки S. Возможно, что строка $ACCA$ является подстрокой строки S, т. к. средняя подстрока является одним из вариантов. Но строка $CAAC$ не может быть подстрокой S, т. к. ее часть AA не является

подстрокой S. Так как строка S могла быть очень длинной, нам нужно было найти быстрый алгоритм для построения всех допустимых строк длиной $2k$.

Самым простым решением была бы конкатенация всех $O(n^2)$ пар строк длиной k с последующей проверкой, что все $k-1$ строк длиной k, переходящих границу конкатенации, действительно являются подстроками (рис. 3.10).

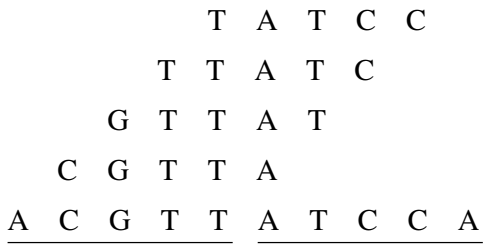


Рис. 3.10. Конкатенация двух строк может входить в строку S, только если в нее входят все объединяемые строки

Например, для подстрок AC, CA и CC возможны девять вариантов объединения — $ACAC$, $ACCA$, $ACCC$, $CAAC$, $CACA$, $CACC$, $CCAC$, $CCCA$ и $CCCC$. Из них можно исключить только последовательность $CAAC$, т. к. AA не является первоначальной подстрокой.

Нам нужно было найти быстрый способ проверить, что $k-1$ подстрок, переходящих границу конкатенации, были членами нашего словаря разрешенных строк длиной k. Время выполнения такого поиска зависит от структуры используемого словаря. Двоичное дерево поиска позволило бы найти правильную строку за $O(\log n)$ сравнений, состоящих в проверке, какая из двух строк длиной k символов встречается первой (по алфавиту). Общее время исполнения с применением такого двоичного дерева было бы равно $O(k\log n)$.

Все это выглядело довольно хорошо, и мой аспирант, Димитрис Маргаритис (Dimitris Margaritis), создал алгоритм поиска с использованием двоичного дерева. Алгоритм выглядел прекрасно, пока мы не испытали его в действии.

— Я выполнил программу на самом мощном компьютере в нашем отделе, но она слишком медленная, — пожаловался мне Димитрис. — Работает бесконечно долго на строках длиной всего лишь в 2 000 символов. Мы никогда не сможем дойти до 50 000.

Мы исследовали нашу программу на профилировщике и обнаружили, что почти все время уходило на поиск в структуре данных. Это нас не удивило, т. к. эта операция выполнялась $k-1$ раз для каждой из $O(n^2)$ возможных последовательностей. Нам была нужна более быстрая структура данных, т. к. поиск был самой внутренней операцией из множества вложенных циклов.

— А если попробовать хэш-таблицу? — предложил я. — Хэширование строки длиной в k символов и поиск ее в нашей таблице должны занять время $O(k)$. Это сократит время исполнения до $O(\log n)$, что будет довольно важно при $n \approx 2\,000$.

Димитрис принялся опять за работу и реализовал хэш-таблицу для нашего словаря. Как и в первый раз, программа выглядела прекрасно, пока мы не испытали ее в действии.

— Программа все еще работает слишком медленно, — опять пожаловался Димитрис. — Конечно же, сейчас на строках длиной в 2 000 символов она выполняется раз в десять быстрее, так что мы можем дойти до строк длиной около 4 000 символов. Но мы все равно никогда не сможем дойти до 50 000.

— Нам следовало ожидать этого, — размышлял я. — В конце концов, $\lg_2(2000) \approx 11$. Нам нужна более быстрая структура данных для поиска строк в словаре.

— Но что может быть быстрее, чем хэш-таблица? — возразил Димитрис. — Чтобы найти строку длиной в k символов, нам нужно считать все эти k символов. Наша хэш-таблица уже дает нам время поиска $O(k)$.

— Конечно же, чтобы проверить первую подстроку, нужно выполнить k сравнений. Но, возможно, мы можем улучшить производительность на последующих проверках. Вспомни, как получаются наши запросы. Для конкатенации подстрок *ABCD* и *EFGH* мы сначала ищем в словаре части *BCDE*, а потом *CDEF*. Эти две подстроки отличаются всего лишь одним символом. Мы должны использовать это обстоятельство, чтобы каждая последующая проверка выполнялась за постоянное время.

— С хэш-таблицей этого нельзя сделать, — заметил Димитрис. — Второй ключ не будет находиться рядом с первым в таблице. Двоичное дерево поиска тоже не поможет. Так как ключи *ABCD* и *BCDE* различаются в первом символе, то они будут в разных частях дерева.

— Но мы можем использовать для этого суффиксное дерево, — возразил я. — Суффиксное дерево содержит все суффиксы данного набора строк. Например, суффиксами *ACAC* строки будут {*ACAC*, *CAC*, *AC*, *C*}. В месте с суффиксами строки *CACT* это даст нам суффиксное дерево, показанное на рис. 3.11.

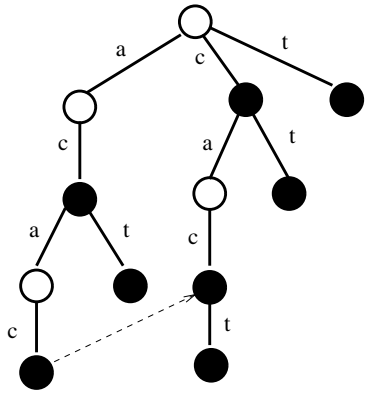


Рис. 3.11. Суффиксное дерево строк *ACAC* и *CACT* с указателем на суффикс строки *ACAC*

Следуя по указателю из строки *ACAC* на ее самый длинный суффикс *CAC*, мы попадем в правильное место для выполнения проверки строки *CACT* на вхождение в наше множество строк. Значит, нам нужно выполнить только одно сравнение символов.

Суффиксные деревья являются удивительными структурами данных; более подробно они рассматриваются в *разделе 12.3*. Димитрис некоторое время изучал литературу

о них, после чего создал элегантный словарь на их основе. Как и прежде, программа выглядела прекрасно, пока мы не испытали ее в действии.

— Теперь скорость работы программы нормальная, но ей не хватает памяти, — пожаловался Димитрис. — Программа создает путь длиной k для каждого суффикса длиной k, так что в дереве получается $\Theta(n^2)$ узлов. Когда количество символов превышает 2 000, происходит аварийное завершение программы. Мы никогда не сможем дойти до строки из 50 000 символов.

Но я еще не был готов сдаваться.

— Проблему с памятью можно решить, используя сжатые суффиксные деревья, — вспомнил я. — Вместо явного представления длинных путей мы можем ссылаться назад на исходную строку. — Сжатые суффиксные деревья всегда занимают линейный объем памяти (см. *раздел 12.3*).

Димитрис пошел снова переделывать программу и реализовал структуру данных сжатого суффиксного дерева. Вот *теперь* программа работала отличнейшим образом! Как видно из табл. 3.4, мы смогли без проблем выполнить нашу эмуляцию для строк длиной $n = 65\,536$.

Таблица 3.4. Время исполнения эмуляции секвенирования SBH с применением разных структур данных

Длина строки	Двоичное дерево	Хэш-таблица	Суффиксное дерево	Сжатое суффиксное дерево
8	0,0	0,0	0,0	0,0
16	0,0	0,0	0,0	0,0
32	0,1	0,0	0,0	0,0
64	0,3	0,4	0,3	0,0
128	2,4	1,1	0,5	0,0
256	17,1	9,4	3,8	0,2
512	31,6	67,0	6,9	1,3
1,024	1 828,9	96,6	31,5	2,7
2,048	11 441,7	941,7	553,6	39,0
4,096	> 2 дней	5 246,7	нехватка	45,4
8,192		> 2 дней	памяти	642,0
16,384				1 614,0
32,768				13 657,8
65,536				39 776,9

Полученные результаты показали, что интерактивное секвенирование SBH может быть очень эффективным. Это позволило нам заинтересовать биологов нашим методом. Но обеспечение реальных экспериментов в лаборатории стало очередной проверкой наше-

го знания вычислительных алгоритмов. Описание нашей работы по решению этой задачи приводится в *разделе 7.7*.

Выводы, которые можно сделать на основе информации из табл. 3.4, очевидны.

Мы выделили одну многократно выполняемую операцию (поиск строки в словаре) и оптимизировали используемую в ней структуру данных. Мы начали с простой структуры (двоичное дерево поиска) в надежде, что этого будет достаточно, а когда оказалось, что нет, то выполнили профилирование программы, чтобы локализовать проблему. Когда и улучшенная структура словаря оказалась недостаточно эффективной, мы провели более глубокий анализ выполняемых запросов, чтобы можно было определить, какие дополнительные улучшения можно было сделать. Наконец, мы не сдались после нескольких неудачных попыток, а продолжали поиски подходящей структуры данных, пока не нашли такую, которая обеспечивала бы требуемый уровень производительности. В разработке алгоритмов, как и в жизни, настойчивость обычно приносит результаты.

Замечания к главе

Оптимизация производительности хэш-таблицы оказывается неожиданно сложной задачей для такой концептуально простой структуры данных. Подробности см. в книге [Knu98].

Программа для оптимизации полосы треугольников, stripe, описывается в книге [ESV96]. Использование методов хэширования для выявления плагиата рассматривается в книге [SWA03].

Обзор алгоритмических вопросов в секвенировании ДНК методом гибридизации приводится в книгах [CK94] и [PL94]. Доклад о нашей работе над интерактивным методом SBH, описанной в *разделе 3.9*, представлен в [MS95a].

3.10. Упражнения

Стеки, очереди и списки

1. [3] Распространенной задачей в компиляторах и текстовых редакторах является обеспечение правильного соотношения и вложенности открывающих и закрывающих скобок в строке. Например, в строке (((())())() скобки вложены правильно, а в строках)()(и ()) — нет. Разработайте алгоритм для проверки размещения скобок, который возвращает значение ИСТИНА при правильном вложении скобок и ЛОЖЬ в противном случае. Чтобы решение было полностью засчитано, алгоритм должен определять положение первой неправильной скобки в случае непарных или неправильно вложенных скобок.
2. [3] Напишите программу для изменения направления односвязного списка на противоположное. Иными словами, после обработки программой все указатели должны быть перевернуты в обратном направлении. Алгоритм должен исполняться за линейное время.
3. [5] Мы видели, как использование динамических массивов позволяет увеличивать размер массива, сохранив при этом постоянное амортизированное время исполнения. В этой задаче рассматривается вопрос произвольного расширения и уменьшения размеров динамических массивов.

а) Разработайте стратегию экономии памяти, при которой размер массива, заполненного меньше чем на половину, уменьшается вдвое. Приведите пример последовательности вставок и удалений, при которой эта стратегия дает неприемлемое амортизированное время исполнения.

б) Разработайте лучшую стратегию экономии памяти, чем предложенная выше, в которой достигается постоянное амортизированное время исполнения для каждой операции удаления.

Деревья и другие словарные структуры

4. [3] Разработайте словарную структуру данных с временем исполнения $O(1)$ в наихудшем случае для операций поиска, вставки и удаления. В данной задаче допускается, что элементами множества являются целые числа из конечного множества 1, 2, ..., n, а инициализация выполняется за время $O(n)$.

5. [3] Определите долю накладных расходов (отношение объема памяти, занимаемого данными, к общему объему памяти, отведенному под структуру) для каждой из следующих реализаций двоичного дерева с n узлами:

   а) Все узлы содержат данные, два указателя на дочерние узлы и указатель на родительский узел. Как поле данных, так и каждый указатель занимает четыре байта.

   б) Только листья содержат данные; внутренние узлы содержат два указателя на дочерние узлы. Поле данных занимает четыре байта, а каждый указатель — два байта.

6. [5] Опишите, как можно модифицировать любую сбалансированную структуру данных таким образом, чтобы время исполнения операций поиска, вставки, удаления, определения минимума и максимума оставалось равным $O(\log n)$, а операции поиска предшествующего и следующего узла выполнялись за время $O(1)$. Какие операции нужно модифицировать для этого?

7. [5] Допустим, имеется сбалансированный словарь, который поддерживает операции *search*, *insert*, *delete*, *minimum*, *maximum*, *successor* и *predecessor*, время исполнения каждой из которых равно $O(\log n)$. Как можно модифицировать операции вставки и удаления, чтобы их время исполнения оставалось равным $O(\log n)$, но время исполнения операций определения максимума и минимума было $O(1)$. (Подсказка: думайте в терминах абстрактных словарных операций и не теряйте время на указатели и т. п.)

8. [6] Разработайте структуру данных, поддерживающую следующие операции:

   - *insert(x,T)* — вставляет элемент x в множество T;
   - *delete(k,T)* — удаляет наименьший k-й элемент из множества T;
   - *member(x,T)* — возвращает значение ИСТИНА тогда и только тогда, когда x является членом T.

   Время исполнения всех операций для набора входных данных из n элементов должно быть равным $O(\log n)$.

9. [8] Операция конкатенации получает на входе два множества S_1 и S_2, где каждый элемент из S_1 меньше, чем любой элемент из S_2, и соединяет их в одно множество. Разработайте алгоритм для конкатенации двух двоичных деревьев в одно. Время исполнения в наихудшем случае должно быть равно $O(h)$, где h — максимальная высота обоих деревьев.

Применение древовидных структур

10. [5] В задаче разложения по контейнерам нам нужно разложить n объектов, каждый весом от нуля до одного килограмма, в наименьшее количество контейнеров, максимальная емкость каждого из которых не больше одного килограмма.

    - Эвристический алгоритм типа "первый лучший" выглядит таким образом. Объекты рассматриваются в порядке, в котором они представлены. Каждый рассматриваемый объект помещается в частично заполненный контейнер, в котором *после* помещения данного объекта останется наименьший свободный объем. Если такого контейнера нет, объект помещается в новый (пустой) контейнер. Реализуйте алгоритм "первый лучший" с временем исполнения $O(n\log n)$. Алгоритм принимает в качестве входа множество из n объектов $w_1, w_2, ..., w_n$ и возвращает количество требуемых контейнеров.
    - Разработайте и реализуйте алгоритм типа "первый худший", в котором следующий объект помещается в такой частично заполненный контейнер, в котором *после* его помещения останется наибольший свободный объем.

11. [5] Допустим, что для последовательности из n значений $x_1, x_2,..., x_n$ нам нужно быстро выполнять запросы, в которых, зная i и j, нужно найти наименьшее значение в подмножестве $x_i, ..., x_j$.

    а) Разработайте структуру данных объемом $O(n^2)$ и временем исполнения запросов $O(1)$.

    б) Разработайте структуру данных объемом $O(n)$ и временем исполнения запросов $O(\log n)$. Чтобы решение было зачтено частично, структура данных может иметь объем $O(n\log n)$ и обеспечивать выполнение запросов за время $O(\log n)$.

12. [5] Допустим, что есть множество S из n чисел и черный ящик, который при вводе в него любой последовательности действительных чисел и целого числа k немедленно выдает ответ о наличии (или отсутствии) в данной последовательности подмножества, сумма которого равна точно k. Покажите, как можно использовать черный ящик $O(n)$ раз, чтобы найти подмножество множества S, сумма членов которого равна точно k.

13. [5] Пусть $A[1..n]$ — массив действительных чисел. Разработайте алгоритм для выполнения последовательности следующих операций:

    - *add(i,y)* — складывает значение y и i-й элемент массива;
    - *partial-sum(i)* — возвращает сумму первых i элементов массива, т. е. $\sum_{j=1}^{i} A[j]$.

    Нет никаких вставок или удалений, только изменяются значения чисел. Каждая операция должна выполняться за $O(\log n)$ шагов. Можно использовать дополнительный массив размером n в качестве буфера.

14. [8] Расширьте структуру данных в предыдущей задаче, чтобы обеспечить поддержку операций вставки и удаления. Каждый элемент теперь имеет *ключ* и *значение*. Доступ к элементу осуществляется по его ключу. Операция сложения выполняется над значениями элементов. Операция *partial-sum* отличается от аналогичной операции из предыдущей задачи. Используемые операции таковы:

    - *add(k,y)* — складывает значение y со значением элемента с ключом k;
    - *insert(k,y)* — вставляет новый элемент с ключом k и значением y;

- *delete(k)* — удаляет элемент с ключом *k*;
- *partial-sum(k)* — возвращает сумму всех текущих элементов множества, у которых значение ключа меньше чем *y*, т. е. $\sum_{x_j<y} x_i$.

Время исполнения в наихудшем случае для любой последовательности $O(n)$ операций должно оставаться таким же — $O(n\log n)$.

15. [8] Разработайте структуру данных, поддерживающую операции поиска, вставки и удаления целого числа X за время $O(1)$ (т. е. за постоянное время, независимо от общего количества хранящихся в структуре целых чисел). Допустим, что $1 < X < n$, и что существует $m + n$ ячеек для хранения целых чисел, где m — наибольшее количество целых чисел, которые могут одновременно храниться в структуре. (Подсказка: используйте два массива $A[1..n]$ и $B[1..m]$.) Массивы нельзя инициализировать, т. к. для этого понадобится $O(m)$ или $O(n)$ операций. Отсюда следует, что массивы изначально содержат непредсказуемые значения, поэтому вам нужно проявлять особую аккуратность.

Проекты по реализации

16. [5] Реализуйте разные варианты словарных структур данных, такие как связные списки, двоичные деревья, сбалансированные двоичные деревья поиска и хэш-таблицы. Экспериментальным путем сравните производительность этих структур данных в простом приложении, считывающим текстовый файл большого объема и отмечающим только один раз каждое встречающееся в нем слово. Это приложение можно эффективно реализовать, создав словарь и вставляя в него каждое новое обнаруженное в тексте слово. Напишите краткий доклад с вашими выводами.
17. [5] Шифр Цезаря (см. *раздел 18.6*) относится к классу простейших шифров. К сожалению, зашифрованные таким способом сообщения можно расшифровать, используя статистические данные о частотном распределении букв латинского алфавита. Разработайте программу для расшифровки достаточно длинных текстов, зашифрованных шифром Цезаря.

Задачи, предлагаемые на собеседовании

18. [3] Каким методом вы бы воспользовались, чтобы найти слово в словаре?
19. [3] Представьте, что у вас полный шкаф рубашек. Как можно организовать рубашки, чтобы упростить их извлечение из шкафа?
20. [4] Напишите функцию поиска среднего узла односвязного списка.
21. [4] Напишите функцию сравнения двух двоичных деревьев. Идентичные двоичные деревья имеют одинаковую структуру и одинаковые значения в соответствующих узлах.
22. [4] Напишите программу для преобразования двоичного дерева поиска в связный список.
23. [4] Реализуйте алгоритм для изменения направления связного списка на обратное. Разработайте такой же алгоритм, но без использования рекурсии.
24. [5] Какой тип структуры данных будет наилучшим для хранения URL-адресов, посещенных поисковым механизмом? Создайте алгоритм для проверки, был ли ранее посещен данный URL-адрес; оптимизируйте алгоритм по времени и памяти.

25. [4] У вас имеется строка поиска и журнал. Вам нужно сгенерировать все символы в строке поиска, выбрав их из журнала. Создайте эффективный алгоритм для определения, содержит ли журнал все символы в строке поиска.
26. [4] Создайте алгоритм для изменения порядка слов в предложении на обратный, т. е. фраза "Меня зовут Крис" должна превратиться в "Крис зовут меня". Оптимизируйте алгоритм по времени и памяти.
27. [5] Разработайте как можно быстрый алгоритм, использующий минимальный объем памяти, для определения, содержит ли связный список петлю. В положительном случае определите местонахождение петли.
28. [5] Имеется неотсортированный массив X, содержащий n целых чисел. Определите множество M, содержащее n элементов, где M_i является произведением всех целых чисел из X, за исключением X_i. Операцию деления применять нельзя, но можно использовать дополнительную память. (Подсказка: существуют решения с временем исполнения быстрее, чем $O(n^2)$.)
29. [6] Разработайте алгоритм, позволяющий найти на веб-странице наиболее часто встречающуюся упорядоченную пару слов (например, "New York"). Какой тип структуры данных вы бы использовали? Оптимизируйте алгоритм по времени и памяти.

Задачи по программированию

Эти задачи доступны на сайтах **http://www.programming-challenges.com** и **http://uva.onlinejudge.org**.

1. Jolly Jumpers. 110201/10038.
2. Crypt Kicker. 110204/843.
3. Where's Waldorf? 110302/10010.
4. Crypt Kicker II. 110304/850.

ГЛАВА 4

Сортировка и поиск

Студенты, получающие специальность, связанную с вычислительными системами, изучают основные алгоритмы сортировки по крайней мере три раза: сначала во введении в программирование, затем в курсе по структурам данных и, наконец, в курсе разработки и анализа алгоритмов. Почему сортировке уделяется так много внимания? На то есть ряд причин.

- Сортировка является базовым строительным блоком, на котором основаны многие алгоритмы. Понимание сортировки расширяет наши возможности при решении других задач.
- Большинство интересных идей, которые используются в разработке алгоритмов (в частности, метод "разделяй и властвуй", структуры данных и рандомизированные алгоритмы), возникло в контексте сортировки.
- Исторически сложилось так, что на сортировку уходит больше компьютерного времени, чем на остальные задачи. Четверть циклов всех мэйнфреймов была использованы для сортировки данных [Knu98]. Сортировка по-прежнему остается самой распространенной практической алгоритмической задачей.
- Сортировка — самая изученная задача в теории вычислительных систем. Известны буквально десятки алгоритмов сортировки, большинство из которых имеют определенное преимущество над другими алгоритмами в определенных ситуациях.

В этой главе мы обсудим сортировку, уделяя особое внимание тому, как ее можно применить для решения других задач. Мы рассмотрим подробное описание нескольких фундаментальных алгоритмов, — *пирамидальной сортировки*[1] (heapsort), *сортировки слиянием* (mergesort), *быстрой сортировки* (quicksort) и *сортировки распределением* (distribution sort), — представляющих собой примеры важных парадигм разработки алгоритмов. Задача сортировки также представлена в *разделе 14.1*.

4.1. Применение сортировки

В этой главе мы рассмотрим несколько алгоритмов сортировки и оценим их временную сложность. Важнейшая идея, которую я хочу донести до читателя, заключается в том, что существуют алгоритмы сортировки со временем исполнения $O(n\log n)$. Это намного лучше, чем производительность $O(n^2)$, которую демонстрируют простые алгоритмы сортировки на больших значениях n.

[1] Другое название "сортировка кучей". — *Прим. перев.*

Рассмотрим следующую таблицу:

n	$n^2/4$	$n\lg n$
10	25	33
100	2 500	664
1 000	250 000	9 965
10 000	25 000 000	132 877
100 000	2 500 000 000	1 660 960

Алгоритмы квадратичной сложности могут быть приемлемыми еще при $n = 10\,000$, но когда $n > 100\,000$, сортировка за квадратичное время становится неприемлемой.

Большинство важных задач можно свести к сортировке; в результате задачу, на первый взгляд требующую квадратичного алгоритма, можно решить с помощью интеллектуальных алгоритмов с временной сложностью $O(n\log n)$. Одним из важных методов разработки алгоритмов является использование сортировки в качестве базового конструктивного блока, т. к. после сортировки набора входных данных многие другие задачи становятся легко решаемыми.

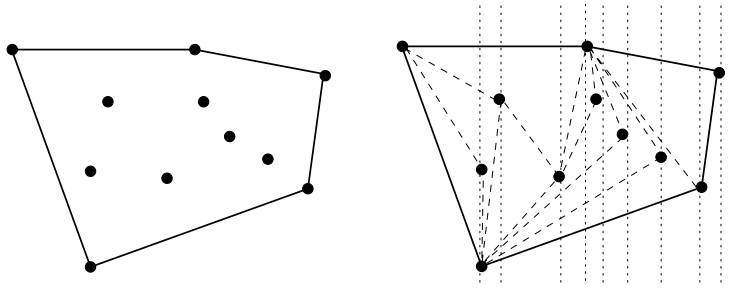
Рассмотрим несколько задач.

- *Поиск.* При условии, что входные данные отсортированы, двоичный поиск элемента в словаре занимает время $O(\log n)$. Предварительная обработка данных для поиска, пожалуй, является наиболее важным применением сортировки.
- *Поиск ближайшей пары.* Как найти в множестве из n чисел два числа с наименьшей разностью между ними? После сортировки входного набора такие числа должны находиться рядом в упорядоченной последовательности. Поиск этих чисел занимает линейное время, а общее время, включая сортировку, составляет $O(n\log n)$.
- *Определение уникальности элементов.* Как определить, имеются ли дубликаты в множестве из n элементов? Это частный случай общей задачи поиска ближайшей пары элементов, в котором нам нужно найти два элемента, разность между которыми равна нулю. Данная задача решается так же, как и предыдущая — сначала выполняется сортировка входного множества, после чего отсортированная последовательность перебирается за линейное время, которое требуется для проверки всех смежных пар.
- *Частотное распределение.* Задача состоит в определении самого часто встречающегося элемента в множестве из n элементов. Если элементы множества отсортированы, то их можно просто перебрать слева направо, подсчитывая, сколько раз встречается каждый элемент, т. к. при сортировке все одинаковые элементы будут размещены рядом друг с другом.

  Для подсчета количества вхождений произвольного элемента k в некоторое множество выполняется двоичный поиск этого элемента в отсортированном массиве ключей. После нахождения требуемого элемента в массиве выполняется сканирование влево от этого элемента, пока не будет найден первый элемент, отличный от k. Потом эта же процедура повторяется вправо от первоначальной точки. Общее время

определения вхождений этим методом равно $O(\log n + c)$, где c — количество вхождений элемента k. Еще лучшее время — $O(\log n)$ — можно получить, определив посредством двоичного поиска местонахождение как элемента $k - \varepsilon$, так и элемента $k + \varepsilon$ (где ε — сколь угодно малое) и затем вычислив разницу между этими двумя позициями.

- *Выбор элемента.* Как найти k-й по величине элемент массива? Если элементы массива отсортированы, то k-й по величине элемент можно найти за линейное время, просто прочитав k-ю ячейку массива. В частности, средний элемент находится в отсортированном массиве в $(n/2)$-й позиции.
- *Выпуклая оболочка.* Как определить многоугольник с наименьшей поверхностью, содержащий множество точек n в двух измерениях? Выпуклую оболочку можно сравнить с резиновой нитью, натянутой вокруг точек в плоскости. Резиновая нить сжимается вокруг самых выступающих точек, образуя многоугольник (рис. 4.1, *а*).



а) *б)*

Рис. 4.1. Создание выпуклой оболочки: резиновой нитью *(а)*, вставкой точек слева направо *(б)*

Выпуклая оболочка дает хорошее представление о размещении точек и является одним из важных конструктивных блоков для создания более сложных геометрических алгоритмов, рассматриваемых в *разделе 17.2*.

Но каким образом можно использовать сортировку для создания выпуклой оболочки? После того как точки отсортированы по x-координате, их можно вставлять в оболочку слева направо. Так как самая правая точка всегда расположена на периметре, то мы знаем, что она войдет в оболочку. После добавления этой новой самой правой точки остальные могут оказаться удаленными, но мы легко идентифицируем эти точки, т. к. они находятся внутри многоугольника, полученного в результате добавления новой точки (рис. 4.1, *б*). Эти точки будут соседями предыдущей вставленной точки, так что их можно будет с легкостью найти и удалить. После сортировки общее время исполнения — линейное.

Хотя некоторые из этих задач (например, выбор элемента) можно решить за линейное время с помощью более сложных алгоритмов, сортировка предоставляет самые простые решения. Время исполнения сортировки оказывается узким местом лишь в немногих приложениях, к тому же в этом случае ее почти всегда можно заменить на более интеллектуальные алгоритмы. Поэтому никогда не бойтесь потратить время на сортировку, при условии, что используется эффективная процедура сортировки.

Подведение итогов

Сортировка является центральной частью многих алгоритмов. Сортировка данных должна быть одним из первых шагов, предпринимаемых любым разработчиком алгоритмов с целью повышения эффективности разрабатываемого решения.

Остановка для размышлений. Поиск пересечения множеств

ЗАДАЧА. Предоставить эффективный алгоритм для определения, являются ли два множества (мощностью m и n соответственно) непересекающимися. Проанализировать сложность алгоритма в наихудшем случае относительно m и n, рассматривая случай, когда m значительно меньше, чем n.

Решение. В голову приходят, по крайней мере, три решения, каждое из которых является разновидностью сортировки и поиска.

- *Сортируется только большое множество.* Большое множество можно отсортировать за время $O(n\log n)$. Теперь для каждого из m элементов меньшего множества выполняется поиск на его наличие в большом множестве. Общее время исполнения будет равно $O((n + m)\log n)$.
- *Сортируется только малое множество.* Меньшее множество сортируется за время $O(m\log m)$. Теперь для каждого из n элементов большего множества выполняется поиск на его наличие в меньшем множестве. Общее время исполнения будет равно $O((n + m)\log n)$.
- *Сортируются оба множества.* Когда оба множества отсортированы, то для определения общего элемента больше не требуется выполнять двоичный поиск. Вместо этого мы сравниваем наименьшие элементы в обеих отсортированных последовательностях и, если они различаются, удаляем меньший элемент. Операция повторяется рекурсивно на все уменьшающихся последовательностях, занимая линейное время, а общее время исполнения (включая сортировку) равно $O(n\log n + m\log m + n + m)$.

Какой же из этих методов самый быстрый? Ясно, что сортировка меньшего множества выполняется быстрее, чем сортировка большего множества, т. к. $\log m < \log n$ при $m < n$. Аналогично, $(n + m)\log m$ должно быть асимптотически меньше, чем $n\log n$, т. к. $n + m < 2n$ при $m < n$. Следовательно, метод сортировки меньшего множества является самым лучшим из этих трех. Обратите внимание, что при константном значении m время исполнения будет линейным.

Кроме этого, *ожидаемое* линейное время можно получить посредством хэширования — создается хэш-таблица, содержащая элементы обоих множеств, после чего выполняется проверка, что конфликты в корзине являются результатом хэширования идентичных элементов. На практике это решение может быть самым лучшим. ■

4.2. Практические аспекты сортировки

Мы уже видели, что сортировка часто применяется в различных алгоритмах. Теперь обсудим несколько эффективных алгоритмов сортировки. Это делает актуальным вопрос — в каком порядке нужно сортировать элементы?

Ответ на этот вопрос зависит от конкретной задачи. Более того, во внимание должен приниматься ряд соображений.

- *Сортировать в возрастающем или убывающем порядке?* Набор ключей S отсортирован в возрастающем порядке, если $S_i \le S_{i+1}$ для всех $1 \le i < n$, и в убывающем по рядке, если $S_i \ge S_{i+1}$ для всех $1 \le i < n$. Для разных приложений требуется разный порядок сортировки.

- *Сортировать только ключ или все поля записи?* При сортировке набора данных необходимо сохранять целостность сложных записей. Например, элементы списка рассылки, содержащего имена, адреса и телефонные номера, можно отсортировать по полю имен, но при этом необходимо сохранять связь между полем имен и другими полями. Таким образом, для сложной записи нам нужно указать, какое поле является ключевым, а также понимать полную структуру записей.

- *Что делать в случае совпадения ключей?* Элементы с одинаковыми значениями ключей будут сгруппированы вместе при любом порядке сортировки, но иногда имеет значение порядок размещения этих элементов относительно друг друга. Допустим, что энциклопедия содержит записи, как для Майкла Джордана баскетболиста, так и для Майкла Джордана специалиста в области статистики. Какая из этих записей должна быть первой? Для решения вопроса одинаковых первичных ключей придется прибегнуть к использованию вторичного ключа, которым может оказаться, например, размер статьи.

  Иногда при сортировке необходимо сохранить первоначальный порядок элементов с одинаковыми ключами. Алгоритмы сортировки, которые автоматически выполняют это требование, называются *устойчивыми* (stable). К сожалению, очень немногие быстрые алгоритмы являются устойчивыми. Стабильность в любых алгоритмах сортировки можно обеспечить, указав первоначальное положение записи в качестве вторичного ключа.

  Конечно же, в случае совпадения ключей можно ничего не делать, а просто позволить алгоритму разместить их там, где он сочтет нужным. Но имейте в виду, что производительность некоторых эффективных алгоритмов сортировки (например, быстрой сортировки) может ухудшиться до квадратичной, если в них явно не предусмотреть возможность обработки большого количества одинаковых значений ключей.

- Как поступать с нечисловыми данными? Сортировка текстовых строк называется упорядочиванием по алфавиту. В библиотеках применяются очень подробные и сложные правила касательно последовательности упорядочения букв и знаков пунктуации. Это вызвано необходимостью принимать такие решения, как, например, являются ли ключи *Skiena* и *skiena* одинаковыми, или как упорядочить записи *Brown-Williams, Brown America* и *Brown, John*?

Правильным решением таких вопросов в алгоритме сортировки будет использование *функции попарного сравнения* элементов, специфической для конкретного приложения. Такая функция принимает в качестве входа указатели на элементы a и b и возвращает "<", если $a < b$, ">", если $a > b$, и " = ", если $a = b$.

Сводя попарное упорядочивание к подобной функции, мы можем реализовывать алгоритмы сортировки, не обращая внимания на такие детали. Мы просто передаем функ-

цию сравнения процедуре сортировки в качестве аргумента. Любой серьезный язык программирования содержит встроенную в виде библиотечной функции процедуру сортировки. Почти во всех случаях использование этой функции предпочтительнее написания своей собственной процедуры сортировки. Например, стандартная библиотека языка C содержит функцию сортировки `qsort`:

```
#include <stdlib.h>
void qsort(void *base, size_t nel, size_t width,
           int (*compare) (const void *, const void *));
```

При использовании функции `qsort` важно понимать назначение ее аргументов. Функция сортирует первые `nel` элементов массива, длина которых составляет `width` байт (на сам массив указывает переменная `base`). Таким образом, мы можем сортировать массивы однобайтовых символов, четырехбайтовых целых чисел или 100-байтовых записей, изменяя лишь значение переменной `width`.

Конечный требуемый порядок отсортированной последовательности определяется функцией `compare`. Эта функция принимает в качестве аргументов указатели на два элемента размером `width` и возвращает отрицательное число, если в отсортированной последовательности первый элемент должен быть перед вторым, положительное число, если наоборот, и ноль, если элементы одинаковые. Код функции для сравнения целых чисел в возрастающей последовательности показан в листинге 4.1.

Листинг 4.1. Реализация функции сравнения

```
int intcompare(int *i, int *j)
{
   if (*i > *j) return (1);
   if (*i < *j) return (-1);
   return (0);
}
```

Эту функцию сравнения можно использовать при сортировке массива `a`, в котором заняты первые `n` ячеек. Функция сортировки вызывается таким образом:

```
qsort(a, n, sizeof(int), intcompare);
```

Название функции сортировки `qsort` дает основания полагать, что в ней реализован алгоритм быстрой сортировки `quicksort`, но это обстоятельство обычно не имеет никакого значения для пользователя.

4.3. Пирамидальная сортировка

Тема сортировки представляет собой естественную лабораторию для изучения принципов разработки алгоритмов, т. к. использование различных методов влечет за собой создание интересных алгоритмов сортировки. В следующих нескольких разделах представляются методы разработки алгоритмов, порожденные определенными алгоритмами сортировки.

Внимательный читатель должен здесь спросить, почему мы рассматриваем стандартные методы сортировки, когда в конце предыдущего раздела была дана рекомендация

не реализовывать их, а вместо этого использовать встроенные библиотечные функции сортировки конкретного языка программирования. Ответ на этот вопрос состоит в том, что применяемые для разработки этих алгоритмов методы также являются очень важными для разработки алгоритмов решения других задач, с которыми вам, скорее всего, придется столкнуться.

Мы начнем с разработки структур данных, т. к. одним из методов существенного улучшения производительности алгоритмов сортировки является использование соответствующих задаче структур данных. Сортировка методом выбора представляет собой простой алгоритм, который в цикле извлекает наименьший оставшийся элемент из неотсортированной части набора входных данных. На псевдокоде этот алгоритм можно выразить таким образом:

```
SelectionSort(A)
     For i = 1 to n do
          Sort[i] = Find-Minimum from A
          Delete-Minimum from A
     Return(Sort)
```

Реализация алгоритма на языке С приводится в листинге 2.1. Здесь массив входных данных разделяется на отсортированную и неотсортированную части. Поиск наименьшего элемента выполняется сканированием элементов в неотсортированной части массива, что занимает линейное время. Найденный наименьший элемент меняется местами с элементом i массива, после чего цикл повторяется. Всего выполняется n итераций, в каждой из которых, в среднем, $n/2$ шагов, а общее время исполнения равно $O(n^2)$.

Может быть, существует лучшая структура данных? После того как местонахождение элемента в неотсортированном массиве определено, его удаление занимает время $O(1)$, но чтобы найти наименьший элемент, нужно время $O(n)$. Эти операции могут использовать очереди с приоритетами. Что будет, если заменить текущую структуру данных структурой с реализацией очереди с приоритетами, такой как пирамида или сбалансированное двоичное дерево? Теперь, вместо $O(n)$, операции внутри цикла исполняются за время $O(\log n)$. Использование такой реализации очереди с приоритетами ускоряет сортировку методом выбора с $O(n^2)$ до $O(n\log n)$.

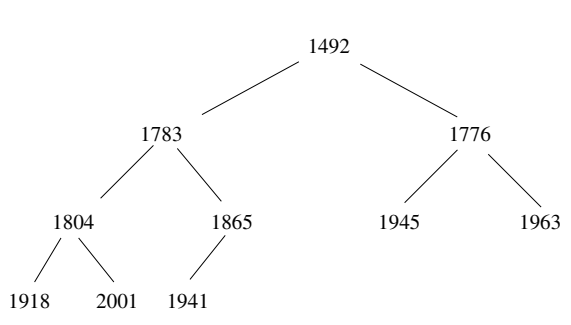
Распространенное название этого алгоритма — *пирамидальная сортировка* — не отражает его механизма, но пирамидальная сортировка в действительности есть нечто иное, как реализация сортировки методом выбора с применением удачной структуры данных.

4.3.1. Пирамиды

Пирамида представляет собой простую и элегантную структуру данных, поддерживающую такие операции очередей с приоритетами, как вставка и поиск наименьшего элемента. Принцип работы пирамиды основан на поддержании порядка "частичной отсортированности" в заданном наборе элементов. Такой порядок слабее, чем состояние полностью отсортированного набора (что обеспечивает эффективность сопровождения), но сильнее, чем произвольный порядок (что обеспечивает быстрый поиск наименьшего элемента).

Властные отношения в любой организации с иерархической структурой отображаются деревом, каждый узел которого представляет сотрудника, а ребро (x, y) означает, что x непосредственно управляет y (доминирует над ним). Сотрудник, отображаемый корневым узлом, находится наверху организационной пирамиды.

Подобным образом *пирамида* определяется как двоичное дерево, в котором значение ключа каждого узла *доминирует* над значением ключа каждого из его потомков. В *неубывающей* бинарной пирамиде (min-heap) доминирующим над своими потомками является узел с меньшим значением ключа, чем значения ключей его потомков; а в *невозрастающей* бинарной пирамиде (max-heap) доминирующим является узел со значением ключа большим, чем значения ключей его потомков. На рис. 4.2, *а* показана неубывающая пирамида дат знаменательных событий в истории Соединенных Штатов.



1	1492
2	1783
3	1776
4	1804
5	1865
6	1945
7	1963
8	1918
9	2001
10	1941

Рис. 4.2. Пирамидальное дерево дат важных событий в американской истории (*а*) и соответствующий массив (*б*)

В наиболее естественной реализации этого двоичного дерева каждый ключ сохранялся бы в узле с указателями на двух его потомков. Как и в случае с двоичными деревьями поиска, объем памяти, занимаемый указателями, может быстро превысить объем памяти, занимаемый ключами, которые интересуют нас в первую очередь.

Однако пирамида является настолько удачной структурой данных, что позволяет нам реализовать двоичные деревья без помощи указателей. Данные в ней сохраняются в виде массива ключей, а позиции ключей используются в роли *неявных* указателей.

Корневой узел дерева сохраняется в первой ячейке массива, а его левый и правый потомки — во второй и третьей соответственно. В общем, мы сохраняем 2^l ключей уровня l полного двоичного дерева в направлении слева направо в позициях 2^{l-1} до 2^l-1, как показано на рис. 4.2, *б*. Для простоты мы полагаем, что нумерация ячеек массива начинается с единицы.

```
typedef struct {
     item_type q[PQ_SIZE+1];    /* Тело очереди */
     int n;                     /* Количество элементов очереди */
} priority_queue;
```

Это представление весьма удачно благодаря легкости, с которой определяется местонахождение родителя и потомка ключа, расположенного в позиции k. Левый потомок

ключа k находится в позиции $2k$, правый — в позиции $2k + 1$, а родительский ключ находится в позиции $\lfloor n/2 \rfloor$. Таким образом, по дереву можно перемещаться без использования указателей (листинг 4.2).

Листинг 4.2. Листинг для работы с пирамидой

```
pq_parent(int n)
{
   if (n == 1) return(-1);
   else return((int) n/2); /* Явно вычисляем floor(n/2) */
}
pq_young_child(int n)
{
   return(2 * n);
}
```

Итак, мы можем сохранить любое двоичное дерево в массиве, не используя указатели. Но нет ли здесь какого-то подвоха? Да, есть. Допустим, что наше дерево высотой h разреженное, т. е. количество его узлов $n < 2h$. Тем не менее, нам нужно выделить место в структуре для всех отсутствующих узлов, т. к. необходимо представить полное двоичное дерево, чтобы сохранить позиционные отношения между узлами.

Таким образом, чтобы не расходовать понапрасну память, мы не можем допустить наличие пробелов в нашем дереве, иными словами, каждый уровень должен быть заполнен до предела. Только последний уровень может быть заполнен частично. Упаковывая элементы последнего уровня как можно плотнее влево, мы можем представить дерево из n ключей, используя точно n элементов массива. Если не придерживаться этих структурных ограничений, то для представления этого же количества элементов нам может потребоваться массив размером $2n$. Так как в пирамидальном дереве всегда заполняются все уровни, кроме последнего, высота пирамиды из n элементов логарифмическая: $h = \lfloor \lg n \rfloor$, т. к. $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1 \geq n$.

Хотя такое неявное представление двоичных деревьев позволяет сэкономить память, оно менее гибкое, чем представление с использованием указателей. Сохранение деревьев с произвольной топологией влечет за собой большой расход памяти. Кроме этого, размещение поддеревьев можно изменять, только явно перемещая все элементы поддерева, а не обновляя один-единственный указатель, как делается в случае деревьев с указателями. Такое отсутствие гибкости объясняет, почему эту идею нельзя использовать для представления двоичных деревьев поиска. Впрочем, для пирамид она вполне подходит.

Остановка для размышлений. Поиск в пирамиде

ЗАДАЧА. Возможен ли эффективный поиск определенного ключа в пирамиде?

Решение. Невозможен. В пирамиде нельзя использовать двоичный поиск, т. к. пирамида не является двоичным деревом. Нам почти ничего не известно об относительном

размещении $n/2$ листьев пирамиды, во всяком случае, ничего, что помогло бы избежать линейного поиска в этих листьях. ■

4.3.2. Создание пирамиды

Пирамиду можно создать пошагово, вставляя каждый новый элемент в самую левую свободную ячейку массива, а именно в $(n + 1)$-ю ячейку предыдущей пирамиды из n элементов. Таким образом обеспечивается сбалансированная форма пирамидального дерева, но необязательно соблюдается порядок доминирования ключей. Новый ключ может быть меньшим, чем его предшественник в неубывающей бинарной пирамиде, или большим, чем его предшественник в невозрастающей бинарной пирамиде.

Эта задача решается за счет обмена местами такого элемента с его родителем. Таким образом, прежний родитель теперь занимает должное место в иерархии доминирования, а порядок доминирования другого дочернего узла прежнего родителя продолжает оставаться правильным, т. к. теперь над ним находится элемент с большим уровнем доминирования, чем у его предыдущего родителя. Новый элемент теперь в несколько лучшем положении, но может продолжать доминировать над своим новым родителем. Тогда мы повторяем процедуру необходимое количество раз на более высоком уровне, перемещая новый ключ *пузырьковым методом* на должное место в иерархии. Так как на каждом шаге корень поддерева заменяется элементом с большим уровнем доминирования, то порядок доминирования сохраняется во всех других частях пирамиды.

Код для вставки нового элемента в пирамиду показан в листинге 4.3.

Листинг 4.3. Вставка элемента в пирамиду

```
pq_insert(priority_queue *q, item_type x)
{
   if (q->n >= PQ_SIZE)
      printf("Warning: priority queue overflow insert x=%d\n",x);
   else {
      q->n = (q->n) + 1;
      q->q[ q->n ] = x;
      bubble_up(q, q->n);
   }
}
bubble_up(priority_queue *q, int p)
{
if (pq_parent(p) == -1) return; /* Корень */
   if (q->q[pq_parent(p)] > q->q[p])  {
      pq_swap(q,p,pq_parent(p));
      bubble_up(q, pq_parent(p));
   }
}
```

На каждом уровне процесс обмена элементов местами занимает постоянное время. Так как высота пирамиды из n элементов равна $\lfloor \lg n \rfloor$, то каждая вставка занимает, самое большее, время $O(\log n)$. Таким образом, первоначальную пирамиду из n элементов

можно создать посредством n таких вставок за время $O(n\log n)$. Соответствующий код приведен в листинге 4.4.

Листинг 4.4. Создание пирамиды повторяющимися вставками

```
pq_init(priority_queue *q)
{
   q->n = 0;
}
make_heap(priority_queue *q, item_type s[], int n)
{
   int i; /* Счетчик */
   pq_init(q);
   for (i=0; i<n;i++)
      pq_insert(q, s[i]);
}
```

4.3.3. Наименьший элемент пирамиды

Осталось рассмотреть такие операции, как поиск и удаление корневого элемента пирамиды. Поиск не составляет никакого труда, т. к. верхний элемент пирамиды находится в первой ячейке массива.

После удаления элемента остается пустая ячейка в массиве, которую можно заполнить, перемещая в нее элемент из самого правого листа (который размещен в n-й ячейке массива).

Таким образом восстанавливается форма дерева, но (как и в случае со вставкой) значение корневого узла может больше не удовлетворять иерархическим требованиям пирамиды. Более того, над этим новым корнем могут доминировать оба его потомка. Корень данной неубывающей бинарной пирамиды должен быть наименьшим из этих трех элементов, т. е. данного корня и его двух потомков. Если текущий корень доминирует над своими потомками, значит, порядок пирамиды восстановлен. В противном случае доминирующий потомок меняется местами с корнем и проблема передается на один уровень вниз.

Проблемный элемент продвигается вниз *пузырьковым методом* до тех пор, пока он не начнет доминировать над всеми своими потомками, возможно, став листом. Данная операция "просачивания" элемента вниз также называется *восстановлением пирамиды* (heapify), т. к. она сливает вместе две пирамиды (поддеревья под первоначальным корнем) с новым ключом. Код для удаления наименьшего элемента пирамиды показан в листинге 4.5.

Листинг 4.5. Удаление наименьшего элемента пирамиды

```
item_type extract_min(priority_queue *q)
{
   int min = -1;      /* Минимальное значение */
   if (q->n <= 0) printf("Warning: empty priority queue.\n");
```

```
   else {
      min = q->q[1] ;
      q->q[1] = q->q[q->n] ;
      q->n = q->n - 1;
      bubble_down(q,1);
   }
   return(min);
}
bubble_down(priority_queue *q, int p)
{
   int c;   /* Индекс потомка */
   int i;   /* Счетчик */
   int min index;   /* Индекс наименьшего потомка */
   c = pq_young_child(p);
   min_index = p;
   for (i=0; i<=1; i++)
      if ((c+i) <= q->n)
         if (q->q[min_index] > q->q[c+i]) min_index = c+i;
      }
   if (min_index != p) {
      pq_swap(q,p,min_index);
      bubble_down(q, min_index);
   }
}
```

Для достижения позиции листа требуется $\lfloor \lg n \rfloor$ исполнений процедуры bubble_down, каждое из которых исполняется за линейное время. Таким образом, удаление корня занимает время $O(\log n)$.

Обмен местами наибольшего элемента с последним элементом и многократный вызов процедуры восстановления пирамиды дает нам алгоритм пирамидальной сортировки с временем исполнения $O(n\log n)$ (листинг 4.6).

Листинг 4.6. Алгоритм пирамидальной сортировки

```
heapsort(item_type s[],int n)
{
   int i;              /* Счетик */
   priority_queue q;   /* Память для пирамидальной сортировки */
   make_heap(&q,s,n);
   for (i=0; i<n; i++)
     s[i] = extract_min(&q);
}
```

Пирамидальная сортировка является замечательным алгоритмом. Его легко реализовать, что подтверждает полный код, представленный в листингах 4.4–4.6. Время исполнения этого алгоритма в наихудшем случае равно $O(n\log n)$, что является наилучшим временем исполнения, которое можно ожидать от любого алгоритма сортировки. Сортировка выполняется "на месте", что означает, что используется только память, содержащая массив с сортируемыми элементами. Хотя на практике другие алгоритмы

сортировки могут оказаться немного быстрее, вы не ошибетесь, если предпочтете этот для сортировки данных в оперативной памяти компьютера.

Очереди с приоритетами являются очень полезными структурами данных. Вспомните, как мы их использовали на практике (см. *раздел 3.6*). Кроме этого, в *разделе 12.2* дается полный набор реализаций очереди с приоритетами.

4.3.4. Быстрый способ создания пирамиды (*)

Как мы видели, пирамиду из n элементов можно создать за время $O(n\log n)$ методом пошаговой вставки элементов. Удивительно, но пирамиду можно создать еще быстрее, используя нашу процедуру `bubble_down` (см. листинг 4.5) и выполнив некоторый анализ.

Допустим, что мы упакуем n ключей, предназначенных для построения пирамиды, в первые n элементов массива очереди с приоритетами. Форма пирамиды будет правильной, но иерархия доминирования будет полностью нарушена. Как можно исправить это положение?

Рассмотрим массив в обратном порядке, начиная с последней (n-й) ячейки. Эта ячейка является листом дерева и поэтому доминирует над своими несуществующими потомками. То же самое верно и для последних $n/2$ ячеек массива, т. к. все они являются листьями. Продолжая двигаться по массиву в обратном направлении, мы, наконец, дойдем до внутреннего узла, имеющего потомков. Этот элемент может не доминировать над своими потомками, но эти потомки представляют правильно построенные (хоть и небольшого размера) пирамиды.

Это как раз та ситуация, для исправления которой предназначена процедура `bubble_down`, — восстановление правильности иерархии произвольного элемента пирамиды, находящегося сверху двух меньших пирамид. Таким образом мы можем создать пирамиду, выполнив $n/2$ вызовов процедуры `bubble_down`, как показано в листинге 4.7.

Листинг 4.7. Алгоритм быстрого создания пирамиды

```
make_heap(priority_queue *q, item_type s[], int n)
{
   int i;          /* Счетчик */
   q->n = n;
   for (i=0; i<n; i++) q->q[i+1] = s[i];
   for (i=q->n; i>=1; i--) bubble_down(q,i);
}
```

Умножив количество вызовов (n) процедуры `bubble_down` на верхний предел сложности каждой операции ($O(\log n)$), мы получим время исполнения $O(n\log n)$. Но это ничуть не быстрее, чем время исполнения алгоритма пошаговой вставки, описанного ранее.

Но обратите внимание, что это *действительно* верхний предел, т. к. фактически только последняя вставка требует $\lfloor \lg n \rfloor$ шагов. Вспомните, что время исполнения процедуры `bubble_down` пропорционально высоте пирамид, слияние которых она осуществляет. Большинство из этих пирамид очень небольшого размера. В полном двоичном дереве из n узлов $n/2$ узлов являются листьями (т. е. имеют высоту 0), $n/4$ узлов имеют высо-

ту 1, $n/8$ узлов имеют высоту 2, и т. д.). В общем, имеется самое большее $\lceil n/2^{h+1} \rceil$ узлов высотой h, соответственно, затраты на создание пирамиды составляют:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil h \le n \sum_{h=0}^{\lfloor \lg n \rfloor} h/2^h \le 2n$$

Так как эта сумма, строго говоря, не является геометрической прогрессией, мы не можем выполнить обычную операцию тождества. Но можно быть уверенным, что несущественный вклад числителя (h) полностью перекроется значением знаменателя (2). Таким образом, функция быстро приближается к линейной.

Имеет ли значение тот факт, что мы можем создать пирамиду за линейное время вместо времени $O(n\log n)$? Как правило, нет. Время создания не доминирует над сложностью пирамидальной сортировки, поэтому улучшение времени создания не улучшает его производительность в наихудшем случае. Тем не менее, перед нами убедительная демонстрация важности внимательного анализа и возможности получения дополнительных бонусов от сходимости геометрической прогрессии.

Остановка для размышлений. Расположение элемента в пирамиде

ЗАДАЧА. Для данной пирамиды на основе массива из n элементов и действительного числа x разработайте эффективный метод определения, является ли k-й элемент пирамиды большим или равным x. Независимо от размера пирамиды время исполнения алгоритма в наихудшем случае должно быть равным $O(k)$. Подсказка: находить k-й наименьший элемент не нужно; требуется только определить его взаимосвязь с x.

Решение. Существуют, по крайней мере, два разных подхода, дающих правильные, но неэффективные алгоритмы для решения этой задачи.

1. Процедура выборки наименьшего значения вызывается k раз, и каждое. из выбранных значений проверяется, меньше ли оно, чем значение x. Таким образом явно сортируются первые k элементов, что предоставляет нам больше информации, чем требует постановка задачи, но для этого требуется время $O(k\log n)$.
2. Наименьший k-й элемент не может находиться глубже, чем на k-м уровне пирамиды, т. к. путь от него до корня должен проходить через элементы с убывающими значениями. Таким образом, мы можем просмотреть все элементы первых k уровней пирамиды и сосчитать, сколько из них меньше, чем x; просмотр прекращается, когда мы либо нашли k таких элементов, либо исчерпали все элементы. В то время как этот алгоритм дает правильное решение задачи, он исполняется за время $O(\min(n, 2^k))$, поскольку k верхних уровней насчитывают 2^k элементов.

Решение с временем исполнения $O(k)$ должно проверить только k элементов меньших, чем x, плюс не более $O(k)$ элементов больших, чем x. Такое решение в виде рекурсивной процедуры, вызывающейся для корневого элемента с параметрами $i = 1$ и $count = k$, приводится в листинге 4.8.

Листинг 4.8. Сравнение *k*-го элемента с числом *x*

```
int heap_compare(priority_queue *q, int i, int count, int x)
   {
   if ((count <= 0) || (i > q->n) return(count);
```

```
   if (q->q[i] < x) {
      count = heap_compare(q, pq_young_child(i), count-1, x);
      count = heap_compare(q, pq_young_child(i)+1, count, x);
   }
    return(count);
}
```

Если корневой элемент неубывающей бинарной пирамиды меньше чем x, тогда в оставшейся пирамиде не может быть элементов меньше чем x, т. к. по определению этого типа пирамиды корневой элемент должен быть наименьшим. В противном случае процедура просматривает потомки всех узлов со значением меньшим, чем x до тех пор, пока либо не найдет k таких узлов (и в этом случае возвращается 0), либо не исчерпает все узлы (и тогда возвращается положительное значение). Таким образом, процедура найдет достаточное количество подходящих элементов, при условии, что они имеются в пирамиде.

Но сколько времени потребуется? Процедура проверяет потомков только тех узлов, чье значение меньше, чем x, и общее количество таких узлов равно, самое большее, k. Для каждого из этих узлов проверяются, самое большее, два потомка, следовательно, количество проверяемых узлов равно, самое большее, $2k$, а общее время исполнения равно $O(k)$. ■

4.3.5. Сортировка вставками

Теперь рассмотрим другой подход к сортировке, опирающийся на использование эффективных структур данных. Выбираем произвольный элемент из неотсортированного списка и вставляем его в должную позицию в отсортированном списке. Псевдокод этого метода сортировки показан в листинге 4.9.

Листинг 4.9. Сортировка вставками

```
InsertionSort(A)
   A[0] = -∞
   for i = 2 to n do
      j = i
      while (A[j] < A[j - 1]) do
         swap(A[j],A[j - 1])
         j = j - 1
```

Реализация алгоритма сортировки вставками на языке C приводится в листинге 2.2. Хотя время исполнения алгоритма сортировки вставками в наихудшем случае равно $O(n)$, его производительность значительно выше, если данные почти отсортированные, т. к. для помещения элемента в должное место достаточно лишь небольшого числа итераций внутреннего цикла.

Сортировка вставками является, возможно, самым простым примером метода сортировки *поэтапными вставками*, где мы создаем сложную структуру из n элементов, сначала создав ее из $n - 1$ элементов, после чего осуществляя необходимые изменения,

чтобы добавить последний элемент. Метод поэтапной вставки оказывается особенно полезным в геометрических алгоритмах. Обратите внимание, что быстрые алгоритмы сортировки на основе поэтапных вставок обеспечены эффективными структурами данных. Операция вставки в сбалансированное дерево поиска занимает время $O(\log n)$, а общее время создания дерева равно $O(n\log n)$, При симметричном обходе такого дерева элементы считываются в отсортированном порядке, что занимает линейное время.

4.4. История из жизни. Билет на самолет

Я взялся за эту работу в поисках справедливости. Меня наняла одна туристическая фирма, чтобы помочь им разработать алгоритм поиска самого дешевого маршрута для перелета из города *x* в город *y*. Сумасшедшие колебания цены на авиабилеты, устанавливаемые согласно современным методам "управления доходами", повергали меня в полное недоумение. Возникало впечатление, что цены на авиарейсы взлетают намного эффективнее, чем сами самолеты. Мне казалось, проблема в том, что авиакомпании скрывают действительно низкие цены на их рейсы. Я рассчитывал, что если я справлюсь с поставленной задачей, это позволит мне в будущем покупать билеты подешевле.

— Смотрите, — начал я свою речь на первом совещании. — Все не так уж сложно. Создаем граф, в котором вершины представляют аэропорты, и соединяем ребром каждую пару аэропортов *u* и *v*, между которыми есть прямой рейс. Устанавливаем вес этого ребра равным стоимости самого дешевого имеющегося в наличии билета из *u* в *v*. Теперь самая низкая цена перелета из города *x* в город *y* будет соответствовать самому короткому пути между соответствующими точками графа. Этот путь можно найти с помощью алгоритма Дейкстры для поиска кратчайшего пути. Проблема решена! — провозгласил я, эффектно взмахнув руками.

Члены собрания задумчиво покивали головами, потом разразились смехом. Мне предстояло кое-что узнать о чрезвычайно сложном процессе формирования цен на билеты пассажирских авиарейсов. В любой момент времени существует буквально миллион разных цен, при этом в течение дня они меняются несколько раз. Цена определяется сложным набором правил, которые являются общими для всей отрасли пассажирских авиаперевозок и представляют собой запутанную систему, не основанную на каких-либо логических принципах. Именно поэтому нам и требовался эффективный способ поиска минимальной цены перелета. Исключением из общего правила являлась только восточноафриканская страна Малави. Обладая населением в 12 миллионов и доходом на душу населения в 596 долларов (179 место в мире), она неожиданно оказалась серьезным фактором, формирующим политику ценообразования мировых пассажирских авиаперевозок. Чтобы определить точную стоимость любого авиамаршрута, требовалось убедиться, что он не проходит через Малави.

Задачу усложнял тот факт, что для первого отрезка маршрута, скажем, от Лос-Анджелеса (аэропорт LAX) до Чикаго (аэропорт ORD) существует сотня разных тарифов, и не меньшее количество тарифов может существовать для каждого последующего отрезка маршрута, например, от Чикаго до Нью-Йорка (аэропорт JFK). Самый деше-

вый билет для отрезка LAX-ORD (скажем, для детей членов Американской ассоциации пенсионеров) может быть несовместим с самым дешевым билетом для отрезка ORD-JFK (если, например, он является специальным предложением для мусульман, которое можно использовать только с последующей пересадкой на рейс в Мекку).

Признав справедливость критики за чрезмерное упрощение задачи, я принялся за работу. Я начал со сведения задачи к наиболее простому случаю.

— Хорошо. Скажем, вам нужно найти самый дешевый билет для рейса с одной пересадкой (т. е. двухэтапного), который проходит вашу проверку соответствия правилам. Есть ли какой-либо способ заранее определить, какие пары этапов маршрута пройдут проверку, и не выполнять при этом саму проверку?

— Нет никакой возможности знать это наперед, — заверили меня. — Мы только можем выполнить процедуру "черного ящика", чтобы решить, существует ли конкретная цена на билет для данного маршрута и для данного пассажира.

— Значит, наша цель заключается в том, чтобы вызывать эту процедуру для минимального количества комбинаций билетов. Это означает, что нужно оценивать все возможные комбинации билетов, начиная с самой дешевой до самой дорогой, до тех пор, пока мы не найдем первую удовлетворяющую правилам комбинацию.

— Совершенно верно.

— Тогда почему бы нам не составить набор всех возможных $m \times n$ пар билетов, отсортировать их по стоимости, после чего оценить их в отсортированной последовательности? Безусловно, это можно сделать за время $O(nm\log(nm))$[1].

— Это мы сейчас и делаем, но составление полного набора $m \times n$ пар обходится довольно дорого, притом, что подходящей может оказаться первая пара.

Я понял, что задача действительно интересная. — Что вам действительно нужно, так это эффективная структура данных, которая многократно возвращает *следующую* наиболее дорогую пару билетов, при этом не составляя все пары наперед.

Это в самом деле было интересно. Поиск наибольшего элемента в наборе значений, подвергающемся вставкам и удалениям, является как раз той задачей, для решения которой замечательно подходят очереди с приоритетами. Но проблема в данном случае заключалась в том, что мы не могли заранее заполнить ключами очередь с приоритетами. Новые пары нужно было вставлять в очередь после каждой оценки.

Я составил несколько примеров, как показано на рис. 4.3.

Каждую возможную цену билета двухэтапного рейса можно было представлять списком индексов ее компонентов (т. е. цен билетов каждого отрезка маршрута). Безусловно, самый дешевый билет для всего маршрута будет получен сложением самых дешевых билетов для каждого отрезка маршрута. В нашем представлении это будет билет (1, 1). Следующий самый дешевый билет получается сложением первого билета

---

[1] Вопрос, можно ли отсортировать все такие суммы быстрее, чем nm произвольных целых чисел, представляет собой нерешенную задачу теории алгоритмов. Дополнительную информацию по сортировке $X + Y$ (так называется эта задача) можно найти в книгах [Fre76] и [Lam92].

X	Y
$100	$50
$110	$75
$130	$125

X+Y	
$150	(1,1)
$160	(2,1)
$175	(1,2)
$180	(3,1)
$185	(2,2)
$205	(2,3)
$225	(1,3)
$235	(2,3)
$255	(3,3)

Рис. 4.3. Сортировка возрастающем порядке сумм *X* и *Y*

одного отрезка маршрута и второго билета другого маршрута, т. е. это будет или (1, 2) или (2, 1). Дальше становится сложнее. Третьим самым дешевым билетом могла бы быть неиспользованная пара из двух, приведенных выше, или же пара (1, 3) или (3, 1). На самом деле, это могла бы быть пара (3, 1), если бы третья по величине цена билета на отрезок *X* маршрута была 120 долларов.

— Скажите, — спросил я, — у нас есть время, чтобы отсортировать эти оба списка цен билетов в возрастающем порядке?

— В этом нет надобности, — ответил руководитель группы. — Они подаются из базы данных в отсортированном порядке.

Это была хорошая новость. Нам не нужно было исследовать пары цен $(i+1, j)$ и $(i, j+1)$ до рассмотрения пары (i, j), т. к. они очевидно были более высокими.

— Есть, — сказал я. — Мы будем отслеживать пары индексов в очереди с приоритетами, а ключом пары будет сумма цен билетов. Вначале в очередь вставляется только одна пара цен — (1, 1). Если эта пара оказывается неподходящей, мы вставляем в очередь две следующие пары — (1, 2) и (2, 1). В общем, после исследования и выбраковки пары (i, j) мы последовательно вставляем в очередь пары $(i+1, j)$ и $(i, j+1)$. Таким образом, мы исследуем все пары в правильном порядке.

Компания быстро уловила суть решения.

— Конечно. Но как насчет дубликатов? Мы будем создавать пару (x, y) двумя разными способами — при расширении пар $(x-1, y)$ и $(x, y-1)$.

— Вы правы. Нам нужна дополнительная структура данных, чтобы предотвратить появление дубликатов. Самым простым решением будет использование хэш-таблицы для проверки существования данной пары в очереди с приоритетами до ее вставки. В действительности, структура данных никогда не будет содержать больше, чем n активных пар, т. к. с каждым отдельным значением цены первого отрезка маршрута можно создать только одну пару цен.

Решение было принято. Наш подход естественным образом обобщался для маршрутов с более чем двумя отрезками (при этом сложность возрастала с увеличением количест-

ва этапов маршрута). Метод выбора первого оптимального варианта, свойственный нашей очереди с приоритетами, позволил системе прекращать поиск, как только она находила действительно самый дешевый билет. Такой подход оказался достаточно быстрым, чтобы предоставить интерактивный ответ пользователю. Однако я не заметил, чтобы мои авиабилеты хоть насколько подешевели.

4.5. Сортировка слиянием. Метод "разделяй и властвуй"

Рекурсивные алгоритмы разбивают большую задачу на несколько подзадач. При рекурсивном подходе сортируемые элементы разбиваются на две группы, каждая из этих меньших групп сортируется рекурсивно, после чего два отсортированных списка сливаются воедино, и их элементы чередуются, образуя полностью отсортированный общий список. Этот алгоритм называется сортировкой слиянием (mergesort).

```
Mergesort(A[1,n])
Merge( MergeSort(A[1, [n/2j]), MergeSort(A[[n/2] + 1,n]) )
```

Базовый случай рекурсивной сортировки имеет место, когда исходный массив состоит из одного элемента, вследствие чего перестановки в нем невозможны. На рис. 4.4 показана графическая иллюстрация работы алгоритма сортировки слиянием. Сортировку слиянием можно представлять себе как симметричный обход верхнего дерева, при котором преобразования представлены в нижнем (перевернутом) дереве.

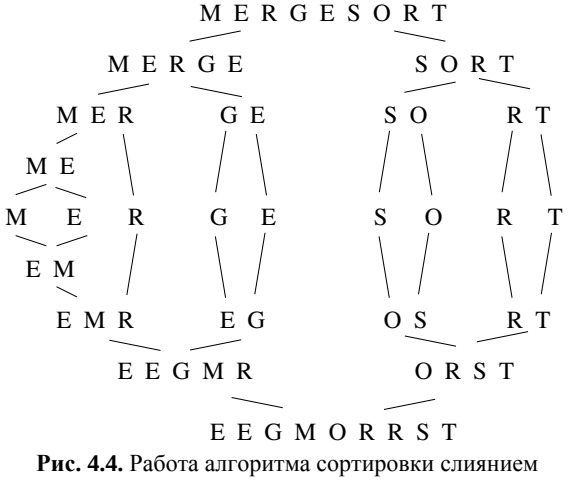


Рис. 4.4. Работа алгоритма сортировки слиянием

Эффективность сортировки слиянием зависит от эффективности слияния двух отсортированных частей в один отсортированный список. Одним из способов было бы объединить обе части и выполнить сортировку получившегося списка методом пирамидальной сортировки или каким-либо другим способом, но это свело бы на нет все наши усилия, потраченные на сортировку этих частей по отдельности.

Поэтому списки *сливаются* в один следующим образом. Оба списка отсортированы в возрастающем порядке, значит, наименьший элемент должен находиться в самом на-

чале одного из них. Этот наименьший элемент перемещается в общий список, при этом один из списков укорачивается на один элемент. Следующий наименьший элемент опять же должен быть самым первым в одном из оставшихся двух списков. Повторяя эту операцию до тех пор, пока не опустеют оба отсортированных списка, мы сливаем эти два списка (с общим количеством элементов n) в один отсортированный список, выполняя, самое большее, $n - 1$ сравнений за время $O(n)$.

Какое же общее время исполнения сортировки слиянием? Чтобы ответить на этот вопрос, следует принять во внимание, какой объем работы выполняется на каждом уровне дерева сортировки. Если мы допустим для простоты, что n является степенью двойки, то k-й уровень содержит все 2^k вызовов процедуры сортировки слиянием, обрабатывающих поддиапазоны из $n/2^k$ элементов.

Работа, выполняемая на нулевом уровне ($k = 0$), состоит в слиянии двух отсортированных списков, каждый размером $n/2$, и при этом происходит, самое большее, $n - 1$ сравнений. Работа, выполняемая на первом уровне ($k = 1$), состоит в слиянии двух отсортированных списков, каждый размером $n/4$, и при этом происходит, самое большее, $n - 2$ сравнений. В общем, работа, выполняемая на k-м уровне, состоит в слиянии 2^k пар отсортированных списков, каждый размером $n/2^{k+1}$, и при этом происходит, самое большее, $n - 2^k$ сравнений. *Слияние всех элементов на каждом уровне выполняется за линейное время.* Каждый из n элементов фигурирует только в одной подзадаче на каждом уровне. Наиболее трудоемким случаем (по количеству сравнений) является самый верхний уровень.

На каждом уровне количество элементов в подзадаче делится пополам. Количество делений пополам числа n до тех пор, пока не будет получена единица, равно $\lceil \lg_2 n \rceil$.

Так как глубина рекурсии составляет $\lg n$ уровней, а каждый уровень обрабатывается за линейное время, то в наихудшем случае время исполнения алгоритма сортировки слиянием равно $O(n \log n)$.

Алгоритм сортировки слиянием хорошо подходит для сортировки связных списков, т. к. он, в отличие от алгоритмов пирамидальной и быстрой сортировки, не основывается на произвольном доступе к элементам. Его основным недостатком является необходимость во вспомогательном буфере при сортировке массивов. Отсортированные связные списки можно легко слить вместе, не требуя дополнительной памяти, а просто упорядочивая указатели. Но для слияния двух отсортированных массивов (или частей массива) требуется третий массив для хранения результатов слияния, чтобы не потерять содержимое сливаемых массивов. Допустим, что мы сливаем множества {4, 5, 6} и {1, 2, 3}, записанные слева направо в одном массиве. Без использования буфера нам придется записывать отсортированные элементы поверх элементов первой половины массива, вследствие чего последние будут утеряны.

Сортировка слиянием является классическим примером алгоритмов типа "разделяй и властвуй". Мы всегда в выигрыше, когда можем разбить одну большую задачу на две подзадачи, т. к. меньшие задачи легче решить. Секрет заключается в том, чтобы воспользоваться двумя частичными решениями для создания решения всей задачи, как это было сделано с операцией слияния.

Реализация

Псевдокод алгоритма сортировки слиянием представлен в листинге 4.10.

Листинг 4.10. Алгоритм сортировки сливанием

```
mergesort(item_type s[], int low, int high)
{
   int i;              /* Счетчик */
   int middle;         /* Индекс среднего элемента */
   if (low < high) {
      middle = (low+high)/2;
      mergesort(s,low,middle);
      mergesort(s,middle+1,high);
      merge(s, low, middle, high);
   }
}
```

Но реализация части алгоритма, в которой осуществляется слияние отсортированных списков, оказывается более сложной. Проблема состоит в том, что нам нужно где-то хранить слитый массив. Чтобы избежать потери элементов в процессе слияния, мы сначала создаем копии исходных массивов (листинг 4.11).

Листинг 4.11. Процедура слияния массивов

```
merge(item_type s[], int low, int middle, int high)
{
   int i;                    /* Счетчики */
   queue buffer1, buffer2; /* Буфера для хранения элементов*/
   init_queue(&buffer1);
   init_queue(&buffer2);
   for (i=low; i<=middle; i++) enqueue(&buffer1,s[i]);
   for (i=middle+1; i<=high; i++) enqueue(&buffer2,s[i]);
   i = low;
   while (!(empty_queue(&buffer1) || empty_queue(&buffer2))) {
      if (headq(&buffer1) <= headq(&buffer2))
         s[i++] = dequeue(&buffer1);
      else
         s[i++] = dequeue(&buffer2);
   }
   while (!empty_queue(&buffer1)) s[i++] = dequeue(&buffer1);
   while (!empty_queue(&buffer2)) s[i++] = dequeue(&buffer2);
}
```

4.6. Быстрая сортировка. Рандомизированная версия

Алгоритм *быстрой сортировки* (*quicksort*) работает таким образом. Из массива размером n выбирается произвольный элемент p. Оставшиеся $n - 1$ элементов массива раз-

деляются на две части: левую (или нижнюю), содержащую все элементы, меньшие элемента *p*, и правую (или верхнюю), содержащую все элементы, большие *p*. Графическая иллюстрация работы алгоритма быстрой сортировки представлена на рис. 4.5.

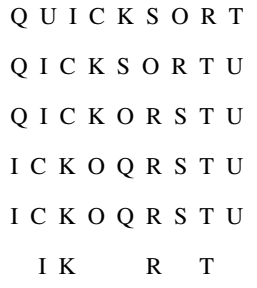


Рис. 4.5. Графическая иллюстрация работы алгоритма быстрой сортировки

Элемент *p* занимает отдельную ячейку между этими двумя частями.

Такое разбиение массива на две части преследует две цели. Во-первых, опорный элемент *p* находится в точно той же позиции массива, в которой он будет находиться в конечном отсортированном массиве. Во-вторых, после разделения массива на части, в конечной отсортированной последовательности элементы не перемещаются из одной части в другую. *Таким образом, сортировку элементов в левой и правой части массива можно выполнять независимо друг от друга.* Это дает нам рекурсивный алгоритм сортировки, т. к. мы можем использовать подход с разбиением массива на две половины для сортировки каждой первоначальной половины. Такой алгоритм должен быть правильным, поскольку, в конечном счете, каждый элемент оказывается в правильном месте. В листинге 4.12 приведен код алгоритма на языке C.

Листинг 4.12. Алгоритм быстрой сортировки

```
quicksort(item_type s[], int l, int h)
{
   int p;    /* Индекс элемента-разделителя */
   if ((h-l)>0) {
      p = partition(s,l,h);
      quicksort(s,l,p-l);
      quicksort(s,p+l,h);
   }
}
```

Массив можно разбить на части за один проход с линейным временем исполнения для определенного опорного элемента посредством постоянного сопровождения трех частей массива: содержащей элементы меньшие, чем опорный элемент (слева от `firsthigh`), содержащей элементы равные или большие, чем опорный (между `firsthigh` и `i`) и содержащей непроверенные элементы (справа от `i`). Реализация процедуры разбиения показана в листинге 4.13.

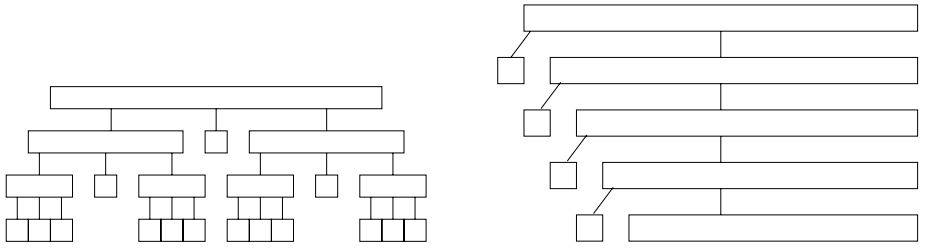
Листинг 4.13. Процедура разбиения массива на части

```
int partition(item_type s[], int l, int h)
{
   int i;          /* Счетчик */
   int p;          /* Индекс элемента-разделителя */
   int firsthigh;  /* Позиция для элемента-разделителя */
   p= h;
   firsthigh = l;
   for (i=l; i<h; i++)
      if (s[i] <s[p]) {
         swap(&s[i],&s[firsthigh]);
         firsthigh ++;
      }
   swap(&s[p],&s[firsthigh]);
   return(firsthigh);
}
```

Так как процедура разделения содержит, самое большее, n операций обмена местами двух элементов, то разбиение массива выполняется за линейное время. Но каково общее время исполнения алгоритма быстрой сортировки? Подобно алгоритму сортировки слиянием, алгоритм быстрой сортировки создает рекурсивное дерево вложенных поддеревьев массива из n элементов. Подобно алгоритму сортировки слиянием алгоритм быстрой сортировки обрабатывает (не слиянием частей в один массив, а наоборот, разбиением массива на части) элементы каждого подмассива на каждом уровне за линейное время. Опять же, подобно алгоритму сортировки слиянием, общее время исполнения алгоритма быстрой сортировки равно $O(n \cdot h)$, где h — высота рекурсивного дерева.

Но здесь трудность состоит в том, что высота дерева зависит от конечного местонахождения опорного элемента в каждой части массива. Если нам очень повезет, и опорным каждый раз будет элемент, находящийся посередине массива, то размер полученных вследствие такого деления подзадач всегда будет равен половине размера задачи предыдущего уровня. Высота представляет количество делений, которым подвергается массив и последующие подмассивы до тех пор, пока полученный подмассив не будет состоять из одной ячейки. Количество таких делений равно, самое большее, $\lceil \lg_2 n \rceil$. Такая благоприятная ситуация показана на рис. 4.6, *а* и представляет наилучший случай алгоритма быстрой сортировки.

Теперь допустим, что нам постоянно не везет, и что наш выбор опорного элемента все время разбивает массив самым неравномерным образом. Это означает, что в качестве опорного всегда выбирается наибольший или наименьший элемент текущего массива. После установки этого опорного элемента в должную позицию у нас остается одна подзадача размером $n - 1$ элементов. Таким образом, мы тратим линейное время на ничтожно малое уменьшение задачи — всего лишь на один элемент (рис. 4.6, *б*). Чтобы разбить массив так, что на каждом уровне будет находиться один элемент, требуется дерево высотой $n - 1$ и время $O(n)$.



а) *б)*

Рис. 4.6. Рекурсивные деревья алгоритма быстрой сортировки: наилучший случай (*а*) и наихудший случай (*б*)

Таким образом, время исполнения алгоритма быстрой сортировки в наихудшем случае хуже, чем для пирамидальной сортировки или сортировки слиянием. Чтобы оправдать свое название, алгоритму быстрой сортировки следовало бы работать лучше в среднем случае. Для понимания этого требуется почувствовать произвольную выборку на интуитивном уровне.

4.6.1. Ожидаемое время исполнения алгоритма быстрой сортировки

Ожидаемое время исполнения алгоритма быстрой сортировки зависит от высоты дерева разбивки первоначального массива, создаваемого произвольными опорными элементами на каждом шаге разбивки. Время исполнения алгоритма слиянием равно $O(n\log n)$, потому что мы рекурсивно разбиваем общее количество элементов на две равные части, после чего сливаем в требуемом порядке за линейное время. Таким образом, всякий раз, когда опорный элемент находится возле центра сортируемого массива (т. е. разделение проходит возле среднего элемента), мы получаем хорошее разбиение и такую же производительность, как и для алгоритма сортировки слиянием.

Я дам не строгое, а опирающееся на интуицию объяснение, почему время исполнения алгоритма быстрой сортировки в среднем случае равно $O(n\log n)$. Какова вероятность того, что выбранный в произвольном порядке разделяющий элемент окажется хорошим? Самым лучшим разделяющим элементом был бы средний элемент массива, т. к. по каждую его сторону оказалось бы ровно по половине элементов исходного массива. К сожалению, вероятность выбрать наудачу точно средний элемент довольно низка, а именно равна $1/n$.

Но допустим, что разделитель является *достаточно хорошим*, если он находится в центральной половине массива, т. е. в диапазоне элементов для сортировки от $n/4$ до $3n/4$. Таких достаточно хороших разделительных элементов имеется довольно много, т. к. полов на всех элементов расположена ближе к центру массива, чем к его краям (рис. 4.7). Таким образом, вероятность выбора достаточно хорошего разделительного элемента при каждом выборе равна 1/2.

Возможно ли, чтобы при бросании монеты постоянно выпадала решка? При бросании честным образом нет. Если бросить монету n раз, то примерно в половине случаев выпадет орел. Вероятность выбора достаточно хорошего разделителя можно рассматривать как выпадение орла при бросании монеты.

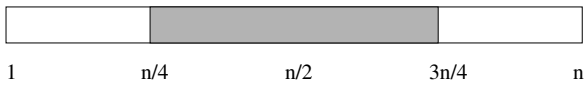


Рис. 4.7. В половине случаев разделительный элемент расположен ближе к середине массива, чем к его краям

При выборе самого худшего возможного достаточно хорошего разделителя большая часть разделенного массива содержит $3n/4$ элемента. Какой будет высота h_g дерева алгоритма быстрой сортировки, созданного в результате последовательного выбора наихудших достаточно хороших разделителей? Самый длинный путь по этому дереву проходит через части размером n, $(3/4)n$, $(3/4)^2 n$, ..., и т. д. до 1 элемента. Сколько раз можно умножить n на 3/4, пока мы не дойдем до 1? Так как

$$(3/4)^{h_g} n = 1 \Rightarrow n = (4/3)^{h_g}, \text{ то } h_g = \log_{4/3} n.$$

Но только половина выбираемых произвольно разделителей являются *достаточно хорошими*, а другую половину мы назовем *плохими*. Наихудшие из плохих разделителей по существу не уменьшают размера раздела вдоль самого глубокого пути. Самый глубокий путь от корня вниз по типичному дереву разделов быстрой сортировки, созданного посредством выбора произвольных разделителей, проходит через приблизительно одинаковое количество достаточно хороших и плохих разделителей. Так как ожидаемое количество достаточно хороших и плохих разделителей одинаково, то плохие разделители могут увеличить высоту дерева, самое большее, вдвое, поэтому $h \approx 2h_g = 2\log_{4/3} n$, что сводится к $\Theta(\log n)$.

В среднем, деревья разделов алгоритма быстрой сортировки, созданные посредством произвольного выбора (и, аналогично, двоичные деревья поиска, созданные произвольными вставками), дают очень хорошие результаты. Более подробный анализ показывает, что после n вставок средняя высота дерева составляет приблизительно $2\ln n$. Так как $2\ln n \approx 1{,}386 \lg_2 n$, то это всего лишь на 39% выше, чем высота идеально сбалансированного двоичного дерева. Поскольку время обработки каждого уровня составляет $O(n)$, то среднее время исполнения алгоритма быстрой сортировки равно $O(n \log n)$. Если нам *чрезвычайно* не повезет и наши произвольно выбранные разделители всегда будут оказываться наибольшими или наименьшими элементами массива, то алгоритм быстрой сортировки превратится в сортировку методом выбора с временем исполнения $O(n^2)$. Но вероятность такого развития событий очень мала.

4.6.2. Рандомизированные алгоритмы

Следует отметить одну важную тонкость в ожидаемом времени исполнения $O(n \log n)$ алгоритма быстрой сортировки. В нашей реализации алгоритма в предыдущем разделе мы выбирали в качестве разделителя последний элемент каждого подмассива (части предыдущего массива). Допустим, мы используем этот алгоритм на отсортированном массиве. В таком случае при каждой разбивке будет выбираться наихудший изо всех возможных элемент-разделитель, а время исполнения будет квадратичным. Для любого детерминистического способа выбора элемента-разделителя существует наихудший входной экземпляр с квадратичным временем исполнения. В представленном ранее анализе утверждается лишь следующее: существует высокая вероятность, что время

исполнения алгоритма быстрой сортировки будет равно $\Theta(n\log n)$, *при условии*, что подлежащие сортировке данные идут в произвольном порядке.

Теперь допустим, что прежде чем приступать к сортировке n элементов, мы переупорядочим их произвольным образом. Эту операцию перестановки можно выполнить за время $O(n)$ (подробности см. в *разделе 13.7*). Эта кажущаяся расточительность гарантирует ожидаемое время исполнения $O(n\log n)$ для *любого* входного экземпляра задачи. Хотя наихудший случай времени исполнения продолжает оставаться возможным, он зависит исключительно от того, насколько нам повезет или не повезет. Но явно определенного наихудшего входного экземпляра больше нет. Следовательно, теперь мы можем сказать: существует высокая вероятность, что время исполнения рандомизированного алгоритма быстрой сортировки будет равно $\Theta(n\log n)$ для *любого* входного экземпляра задачи.

В качестве альтернативного варианта мы могли бы прийти к такому же утверждению, выбирая произвольный элемент-разделитель на каждом шаге.

Рандомизация является мощным инструментом для улучшения алгоритмов с плохой временной сложностью в наихудшем случае, но с хорошей сложностью в среднем случае. С ее помощью алгоритмы можно сделать более устойчивыми в граничных случаях и более эффективными на высоко структурированных вводных экземплярах, которые делают неэффективными эвристические механизмы принятия решений (как в случае с отсортированным входом для алгоритма быстрой сортировки). Рандомизацию часто можно применять с простыми алгоритмами, обеспечивая таким образом производительность, которую в противном случае можно получить, только используя сложные детерминистические алгоритмы.

Чтобы должным образом анализировать рандомизированные алгоритмы, необходимо иметь определенные познания в области теории вероятностей, обсуждение которой выходит за рамки данной книги. Но некоторые подходы к разработке эффективных рандомизированных алгоритмов можно с легкостью объяснить.

- *Рандомизированная выборка.* Допустим, мы хотим получить общее представление о среднем значении n элементов, но у нас нет ни достаточного времени, ни места, чтобы просмотреть все значения. В таком случае мы делаем выборку произвольных элементов из всего множества и исследуем эти элементы. Полученный результат должен быть репрезентативным для всего множества.

  Эта идея лежит в основе исследований путем опроса. Но если не обеспечить действительно *произвольную* выборку, а опросить x первых встречных, то возможны искажения в ту или иную сторону. Во избежание таких искажений выполняющие опрос агентства обычно звонят по произвольным телефонным номерам и надеются, что кто-либо ответит.

- *Рандомизированное хэширование.* Мы уже говорили, что посредством хэширования можно реализовать словарные операции с ожидаемым временем исполнения $O(1)$. Но для любой хэш-функции имеется наихудший случай в виде набора ключей, которые хэшируются в одну корзину. Но допустим, что первым шагом алгоритма мы выбираем произвольную функцию хэширования из большого семейства подходящих функций. Таким образом, мы получаем такую же улучшенную гарантию, как и для рандомизированного алгоритма быстрой сортировки.

- *Рандомизированный поиск.* Рандомизацию можно также применять для организации методов поиска, таких как имитация отжига. Подробности см. в *разделе 7.5.3*.

Остановка для размышлений. Болты и гайки

ЗАДАЧА. Задача *болтов и гаек* определяется таким образом. Есть набор разных n гаек и такое же количество соответствующих болтов. Вам нужно найти для каждого болта подходящую гайку. Сравнивать можно только болты и гайки, т. е. нельзя сравнивать гайки с гайками или болты с болтами.

Разработайте алгоритм для решения этой задачи за время $O(n^2)$, а потом разработайте рандомизированный алгоритм для решения этой задачи за ожидаемое время $O(n\log n)$.

Решение. Алгоритм полного перебора решает эту задачу, сравнивая первый болт со всеми гайками, пока не найдет подходящую, после чего переходит к следующему болту и сравнивает его со всеми оставшимися гайками, и т. д. В худшем случае для первого болта потребуется n сравнений. Повторение этой процедуры для всех последующих болтов со всеми остающимися гайками дает нам квадратичное число сравнений.

А если вместо последовательного перебора болтов, начиная с первого, мы каждый раз будем брать произвольный болт? В среднем, мы можем ожидать перебора около половины гаек, пока не найдем подходящую для данного болта, поэтому этот рандомизированный алгоритм будет иметь ожидаемое время исполнения наполовину меньшее, чем в худшем случае. Это можно рассматривать как определенное улучшение, хотя и не асимптотического типа.

Так как рандомизированный алгоритм быстрой сортировки обеспечивает ожидаемое время исполнения, то будет естественной идея эмулировать его для решения данной задачи. Действительно, в отсортированных последовательностях для i-го болта подойдет i-я гайка.

Основной операцией алгоритма быстрой сортировки является разбиение массива элементов на две части по элементу-разделителю. Можем ли мы разбить множества гаек и болтов по произвольно выбранному болту b? Определенно, что, сравнивая размер гаек с размером произвольно выбранного болта b, мы можем разбить множество гаек на две части — меньших и больших чем гайка, подходящая для болта b. Но нам также нужно разбить на части множество болтов по этому же произвольно выбранному болту размером b, а мы не можем сравнивать болты друг с другом. Но ведь когда мы найдем гайку подходящего размера для болта размером b, то мы можем использовать ее для разбиения болтов, точно так же, как мы использовали болт для разбиения на части множества гаек. Разбиение на части гаек и болтов происходит за $2n - 2$ сравнения, а время исполнения оставшихся операций следует непосредственно из анализа рандомизированного алгоритма быстрой сортировки.

Эта задача интересна тем, что для нее не существует простого детерминистического алгоритма. Это хорошая иллюстрация, как использование рандомизации позволяет избавиться от плохих входных экземпляров задачи посредством простого и изящного алгоритма. ■

4.6.3. Действительно ли алгоритм быстрой сортировки работает быстро?

Существует четкое, асимптотическое различие между алгоритмом с временем исполнения $\Theta(n\log n)$ и алгоритмом с временем исполнения $\Theta(n^2)$. Поэтому только самый недоверчивый читатель будет сомневаться в моем утверждении, что алгоритмы сортировки слиянием, пирамидальной сортировки и быстрой сортировки покажут лучшую производительность на достаточно больших входных экземплярах задачи, чем алгоритмы сортировки вставкам и или сортиров и методом выбора.

Но как можно сравнить два алгоритма с временной сложностью $\Theta(n\log n)$, чтобы решить, который из них быстрее? Как можно доказать, что алгоритм быстрой сортировки действительно быстрый? К сожалению, модель RAM и асимптотический анализ являются слишком грубыми инструментами для сравнений такого типа. В случае алгоритмов с одинаковой асимптотической сложностью, детали их реализации и особенности программной и аппаратной платформы, на которой они исполняются, такие как объем оперативной памяти и производительность кэша, могут вполне оказаться решающим фактором.

Что можно сказать, так это то, что в процессе экспериментирования было установлено, что должным образом реализованный алгоритм быстрой сортировки обычно в 2–3 раза быстрее, чем алгоритм сортировки слиянием или пирамидальной сортировки. Основной причиной этому является тот факт, что в алгоритме быстрой сортировки операции внутреннего цикла менее сложные. Но если вы не верите, что алгоритм быстрой сортировки быстрее, я не смогу вам этого доказать. Ответ на этот вопрос лежит за рамками применения аналитических инструментов, рассматриваемых в этой книге. Самым лучшим способом будет реализовать оба алгоритма и определить их эффективность экспериментальным способом.

4.7. Сортировка распределением. Метод блочной сортировки

Имена в телефонной книге можно отсортировать по первой букве фамилии. Таким образом у нас получится 26 разных корзин для имен (в английском алфавите 26 букв — *прим. перев.*). Обратите внимание, что любая фамилия в корзине J должна находиться после всех фамилий из корзины I, но перед любой фамилией из корзины K. Вследствие этого обстоятельства фамилии можно отсортировать в каждой отдельной корзине, после чего просто объединить отсортированные корзины.

Если имена распределены среди корзин равномерно, то каждая из получившихся 26 подзадач сортировки должна быть значительно меньшего размера, чем первоначальная задача. Далее, разделяя каждую корзину по второй букве фамилии, потом третьей и т. д., мы создаем корзины все меньшего и меньшего размера (т. е. содержащие все меньше и меньше элементов-фамилий). Список фамилий будет отсортирован, поскольку вследствие такого деления каждая корзина содержит только одну фамилию. Только что описанный алгоритм сортировки называется *блочной сортировкой* (bucket sort) или *сортировкой распределением* (distribution sort).

Применение корзин является очень эффективным подходом, когда м ы уверены, что данные распределены приблизительно равномерно. Эта же идея лежит в основе хэш-таблиц, kd-деревьев и многих других практических структур данных. Обратной стороной этой медали является то, что производительность может быть ужасной, если распределение данных окажется не таким, на какое мы рассчитывали . Хотя для таких структур данных, как сбалансированные двоичные деревья, гарантируется производи тельность худшего случая для входных данных с любым распределением, такая гарантия отсутствует для эвристических структур данных с неожиданным распределением ввода.

Неравномерное распределение данных встречается и в реальной жизни. Возьмем, например, такую необычную американскую фамилию, как Shiffiett. Когда я последний раз смотрел телефонный справочник Манхэттена (в котором свыше миллиона фамилий), то в нем было пять человек с такой фамилией. Как вы думаете, сколько людей по фамилии Shiffiett проживает в небольшом городке с населением в 50 000? На рис. 4.8 показан фрагмент телефонного справочника для города Шарлотсвилл в штате Вирджиния. Фамилия Shiffiett занимает в справочнике более *двух с половиной страниц*.

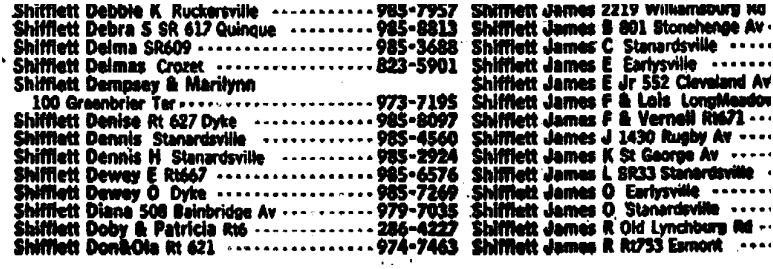


Рис. 4.8. Фрагмент телефонного справочника

Клан Shiffiett присутствует в этом регионе много лет, и это обстоятельство приведет в расстройство любую программу сортировки распределен ием, т. к. при последователь ном разбиении корзины *S* на *Sh* на *Shi* на *Shif* на ... на *Shifiett* в действительности не происходит никакого значительного разбиения.

Подведение итогов

Сортировку можно использовать для иллюстрации большинства парадигм разработки алгоритмов. Методы структур данных, принцип "разделя й и властвуй", рандомизация и поэтапная обработка — все эти подходы позволяют разрабатывать эффективные алгоритмы сортировки.

4.7.1. Нижние пределы для сортировки

Обсудим последний вопрос, касающийся сложности алгоритмов сортировки. Мы знаем несколько алгоритмов сортировки. Для всех время исполнения в наихудшем случае было равно $O(n\log n)$, и ни один из них не выполнялся за линейное время. Для сортировки n элементов неизбежно требуется рассмотреть каждый из них, поэтому временная сложность любого алгоритма сортировки в наихудшем случае должна быть $\Omega(n)$. Можем ли мы закрыть эту брешь в $\Theta(\log n)$?

К сожалению, нет. Нижнюю границу $\Omega(n\log n)$ можно установить на основе того обстоятельства, что любой алгоритм сортировки должен вести себя по-разному при сортировке каждой из возможных разных $n!$ перестановок n элементов. Время исполнения любого алгоритма сортировки на основе сравнений определяется результатом каждого попарного сравнения. Набор всех возможных исполнений такого алгоритма можно представить в виде дерева с $n!$ листьями. Минимальная высота дерева соответствует самому быстрому возможному алгоритму, и получается, что $\lg(n!) = \Theta(n\log n)$.

Эта нижняя граница важна по нескольким причинам. Прежде всего, этот подход можно расширить, чтобы получить нижние границы для многих приложений сортировки, включая определение уникальности элемента, поиск наиболее часто встречающегося элемента, а также создание выпуклых оболочек. Среди алгоритмических задач сортировка является одной из немногих, обладающих нетривиальной нижней границей. В *главе 9* мы рассмотрим альтернативный подход к доказательству маловероятности существования быстрых алгоритмов.

4.8. История из жизни. Адвокат Скиена

Я веду тихий, достаточно честный образ жизни. Одной из наград за такой образ жизни является то, что сплю спокойно, не опасаясь никаких неприятных сюрпризов. Поэтому я был крайне поражен, когда мне позвонила женщина-адвокат, которая хотела не просто поговорить со мной, но поговорить об алгоритмах сортировки.

Ее фирма работала над делом, касающимся высокопроизводительных программ сортировки, и им был нужен эксперт, который мог бы объяснить присяжным технические подробности. По первому изданию этой книги они поняли, что я кое-что знаю об алгоритмах, и поэтому решили обратиться именно ко мне. Но прежде, чем нанять меня, они захотели узнать, как студенты оценивают мои преподавательские способности, чтобы удостовериться в том, что я могу доходчиво объяснять людям сложные понятия[1]. Участие в этом деле оказалось увлекательной возможностью узнать, как работают *действительно* быстрые программы сортировки. Я полагал, что, наконец, смогу ответить на вопрос, какой из алгоритмов сортировки в памяти является самым быстрым. Будет это пирамидальная сортировка или быстрая сортировка? Какие особенности алгоритмов способствовали сведению к минимуму количества сравнений в практических приложениях?

Ответ был довольно отрезвляющим. *Сортировка в памяти никого не интересовала.* Главное заключалось в сортировке *громадных* файлов, намного больших, чем которые могли поместиться в оперативную память. Основная работа состояла в записывании и считывании данных с диска, и хитроумные алгоритмы для сортировки в памяти никому не были интересны, т. к. в реальной жизни сортировать нужно многие гигабайты данных.

Вспомним, что время обращения к жесткому диску относительно велико из-за необходимости позиционировать магнитную головку чтения/записи. После того как головка

[1] Один мой цинично настроенный коллега по профессорско-преподавательскому составу сказал, что это первый случай, когда кто-либо поинтересовался мнением студентов относительно преподавателей.

установлена в нужном месте, передача данных осуществляется очень быстро, и для чтения большого блока данных требуется приблизительно такое же время, что и для одного байта. Таким образом, целью является сведение к минимуму количества блоков данных для чтения/записи и никогда не простаивал, координация этих операций, чтобы алгоритм сортировки никогда не простаивал, ожидая данные.

Необходимость интенсивной работы с диском во время сортировки лучше всего демонстрируется на ежегодных соревнованиях по сортировке Minutesort. Перед участниками стоит задача отсортировать наибольший объем данных за одну минуту. Действующим чемпионом в этом соревновании является Джим Вилли (Jim Wyllie) из отдела исследований IBM. На своем скромном 40-узловом кластере из 80 процессоров Itanium, оснащенном массивом из 2 520 сетевых дисков, он смог отсортировать 116 гигабайт данных за 58,7 секунд. Существует и другое, более приближенное к практике, соревнование Pennysort, целью которого является получение максимальной производительности сортировки на один цент стоимости оборудования. Действующим чемпионом в этом соревновании является представленный китайскими разработчиками алгоритм BSIS, который на персональном компьютере стоимостью 760 долларов, оснащенном четырьмя приводами SATA, отсортировал 32 гигабайта за 1 679 секунд. Информацию о текущих рекордах сортировки можно получить на веб-сайте Sort Benchmark по адресу **http://sortbenchmark.org/**.

Итак, какой алгоритм лучше всего подходит для сортировки данных вне оперативной памяти? Оказывается, это сортировка многоканальным слиянием с применением множества инженерных и других специальных приемов. Создается пирамида из членов верхнего блока каждого из k отсортированных списков. Последовательно снимая с этой пирамиды верхний элемент и сливая вместе эти k списков, алгоритм создает общий отсортированный список.

Так как пирамида находится в памяти, то эти операции выполняются с высокой скоростью. Когда имеется достаточно большой объем отсортированных данных, они записываются на диск, тем самим освобождая память для новых данных. Когда начинают заканчиваться элементы верхнего блока одного из сливаемых отсортированных k списков, загружается следующий верхний блок данных из этого списка.

Оценить на этом уровне производительность программ/алгоритмов сортировки и решить, какой из них действительно быстрее, очень трудно. Будет ли справедливо сравнивать производительность коммерческой программы, предназначенной для обработки общих файлов, с производительностью кода, в котором убрано все лишнее и который оптимизирован для обработки целых чисел? В соревновании *Minutesort* в качестве входных данных для сортировки используются произвольно генерируемые записи размером в 100 байтов. Сортировка таких записей существенно отличается от сортировки имен или целых чисел. Например, при сортировке этих записей широко применяется прием, при котором от каждого элемента берется короткий префикс, и сортировка сначала выполняется по этим префиксам, чтобы избежать перемещения лишних байтов.

Какие же уроки можно извлечь из всего этого?

Самый важный состоит в том, что следует всячески избегать втягивания себя в судебное разбирательство в качестве как истца, так и ответчика.

Суды не являются инструментом для скорого разрешения разногласий. Юридические баталии во многом похожи на военные битвы: они очень быстро обостряются, стано-

вятся очень дорогостоящими в денежном и временном отношении, а также в отношении душевного состояния, и обычно кончаются только тогда, когда обе стороны истощены и идут на компромисс. Мудрые люди могут решить свои проблемы, не прибегая к помощи судов. Усвоив этот урок должным образом сейчас, вы сможете сэкономить средства, в тысячи раз превышающие стоимость этой книги.

Что касается технических аспектов, то важно уделять должное внимание производительности внешних средств хранения данных при обработке очень больших наборов данных алгоритмами с низкой временной сложностью (например, линейной или равной $n\log n$).

В таких случаях даже такие постоянные множители, как 5 или 10, могут означать разницу между возможностью и невозможностью сортировки.

Конечно же, для наборов данных большого объема алгоритмы с квадратичным временем исполнения обречены на неудачу независимо от времени доступа к данным на дисках.

4.9. Двоичный поиск и связанные с ним алгоритмы

Алгоритм двоичного поиска позволяет осуществлять быстрый поиск в массиве S отсортированных ключей. Чтобы найти ключ q, мы сравниваем значение q со средним ключом массива $S[n/2]$. Если значение ключа q меньше, чем значение ключа $S[n/2]$, значит, данный ключ должен находиться в верхней половине массива S; в противном случае он должен находиться в его нижней половине. Рекурсивно повторяя этот процесс на половине, содержащей элемент q, мы находим его за $\lceil \lg n \rceil$ сравнений, что является большим улучшением по сравнению с ожидаемыми $n/2$ сравнениями при последовательном поиске. Реализация алгоритма двоичного поиска на языке C показана в листинге 4.14.

Листинг 4.14. Реализация алгоритма двоичного поиска

```
int binary_search(item_type s[], item_type key, int low, int high)
{
   int middle;  /* Индекс среднего элемента  */
   if (low > high) return (-1); /* Ключ не найден */
   middle = (low+high)/2;
   if (s[middle] == key) return(middle);
   if (s[middle] > key)
      return (binary_search(s, key, low, middle-1));
   else
      return (binary_search(s, key, middle+1, high) );
}
```

Вероятно, все это вы уже знаете. Но важно почувствовать, насколько быстрым является алгоритм двоичного поиска. Существует популярная детская игра "Двадцать вопросов". Суть этой игры состоит в том, что один из игроков загадывает слова, а второй

пытается угадать его. Если после 20 вопросов слово не отгадано, то выигрывает первый игрок, в противном случае — второй. Но в действительности второй игрок всегда находится в выигрышном положении, т. к. он для угадывания слова может применить стратегию двоичного поиска. Для этого он берет словарь, открывает его посередине, выбирает слово (например, "ночь") и спрашивает первого игрока, находится ли загаданное им слово перед словом "ночь". Процесс рекурсивно повторяется для соответствующей половины, пока слово не будет отгадано.

Так как стандартные словари содержат от 50 000 до 200 000 слов, то можно быть уверенным, что 20 попыток будет более чем достаточно.

4.9.1. Частота вхождения элемента

Простые варианты двоичного поиска порождают несколько интересных алгоритмов. Допустим, мы хотим сосчитать, сколько раз данный ключ k (например, "Skiena") встречается в данном отсортированном массиве. Так как при сортировке все копии k собираются в один непрерывный блок, то задача сводится к поиску этого блока и последующего измерения его размера.

Представленная ранее процедура двоичного поиска позволяет найти индекс одного из элементов в соответствующем блоке (x) за время $O(\lg n)$. Естественным способом определения границ блока будет последовательная проверка элементов слева от x до тех пор, пока не будет найден элемент, отличающийся от ключа, и повторение процесса проверки для элементов справа от x. Разница между индексами левой и правой границы блока, увеличенная на единицу, и будет количеством вхождений элемента k в данный набор данных.

Этот алгоритм исполняется за время $O(\lg n + s)$, где s — количество вхождений ключа. Но если весь массив состоит из одинаковых ключей, то это время может ухудшиться до линейного. Алгоритм двоичного поиска можно ускорить, модифицировав его для поиска *границ* блока, содержащего элемент k, вместо самого k. Допустим, мы удалим проверку на равенство:

```
if (s[middle] == key) return(middle);
```

из реализации двоичного поиска в листинге 4.14 и для каждого неуспешного поиска вместо –1 будем возвращать индекс `low`. Теперь *любой* поиск будет заканчиваться неудачей по причине отсутствия проверки на равенство. При сравнении ключа с одинаковым элементом массива процесс поиска будет переходить в правую половину массива, в конце концов, останавливаясь на правой границе блока одинаковых элементов. Левая граница блока определяется изменением направления двоичного сравнения на обратное и повторением поиска. Так как поиск выполняется за время $O(\lg n)$, то подсчет количества вхождений элемента занимает логарифмическое время, независимо от размера блока.

4.9.2. Односторонний двоичный поиск

Теперь допустим, что у нас есть массив, заполненный последовательностью нулей, за которыми следует неограниченная последовательность единиц, и нужно найти границу между этими двумя последовательностями. Если бы мы знали количество n элементов

массива, то на определение точки перехода посредством двоичного поиска ушло бы $\lceil \lg n \rceil$ операций сравнения. При отсутствии границы мы можем последовательно выполнять сравнения по увеличивающимся интервалам ($A[1]$, $A[2]$, $A[4]$, $A[8]$, $A[16]$, ...), пока не найдем первый ненулевой элемент. Теперь у нас имеется окно, содержащее целевой элемент, и мы можем применить двоичный поиск. Такой односторонний двоичный поиск возвращает границу p за, самое большее, $2\lceil \lg p \rceil$ операций сравнения, независимо от размера массива. Односторонний двоичный поиск особенно хорошо подходит для локализации элемента, расположенного недалеко от текущей позиции просмотра.

4.9.3. Корни числа

Квадратным корнем числа n является такое число r, для которого $r^2 = n$. Хотя операция вычисления квадратного корня имеется в любом карманном калькуляторе, нам будет полезно разработать эффективный алгоритм для его вычисления.

Заметим, что квадратный корень числа $n > 1$ должен находиться в интервале от 1 до n. Пусть $l = 1$, $r = n$. Теперь рассмотрим среднюю точку этого интервала $m = (l + r)/2$ и отношение m^2 к n. Если $n > m^2$, то квадратный корень должен быть больше, чем m, поэтому мы устанавливаем $l = m$ и повторяем процедуру. Если $n < m^2$, то квадратный корень должен быть меньшим, чем m, поэтому устанавливаем $r = m$ и повторяем процедуру. В любом случае мы сократили интервал наполовину посредством всего лишь одного сравнения. Продолжая действовать подобным образом, мы найдем квадратный корень без учета знака за $\lg n$ сравнений.

Этот метод деления интервала пополам можно также применять для решения более общей задачи поиска корней уравнения. Число x называется *корнем* функции f, если $f(x) = 0$. Возьмем два числа l и r, для которых $f(l) > 0$ и $f(r) < 0$. Если f является непрерывной функцией, то ее корень должен находиться в интервале между l и r. В зависимости от знака $f(m)$, принимая $m = (l + r)/2$, мы можем уменьшить это содержащее корень окно наполовину за одно сравнение, прекращая поиск, как только наша оценка корня становится достаточно точной.

Для обоих типов задач поиска корня известны алгоритмы их решения, которые выдают результат быстрее, чем двоичный поиск. В частности, вместо исследования средней точки интервала в этих алгоритмах применяется интерполяция для поиска подходящей точки, расположенной ближе к искомому корню. Тем не менее, метод двоичного поиска является простым и надежным, и работает так хорошо, насколько это возможно, не требуя дополнительной информации о самой функции.

4.10. Метод "разделяй и властвуй"

Один из наиболее эффективных подходов к решению задач состоит в разбиении их на меньшие части, поддающиеся решению с большей легкостью. Задачи меньшего размера являются менее сложными, что позволяет фокусировать внимание на деталях, которые не попадают в поле зрения при исследовании всей задачи. Когда мы можем разбить задачу на более мелкие экземпляры задачи этого же типа, то становится очевид-

ным использование для ее решения рекурсивного алгоритма. Для эффективной параллельной обработки задачу необходимо разложить, по крайней мере, на столько меньших задач, сколько имеется процессоров. Это требование становится еще более важным с развитием кластерных вычислений и многоядерных процессоров.

Принцип разбиения задачи на меньшие части лежит в основе двух важных парадигм разработки алгоритмов. В частности, в *главе 8* обсуждается динамическое программирование. Суть этого метода состоит в удалении из задачи некоторого элемента, решении получившейся меньшей задачи и использовании найденного решения, для корректного возвращения элемента на место. А в методе "разделяй и властвуй" задача рекурсивно разбивается на, скажем, половины, каждая половина решается по отдельности, после чего решения каждой половины объединяются в общее решение.

Эффективный алгоритм получается в том случае, когда слияние решений половин занимает меньше времени, чем их решение. Классическим примером алгоритма типа "разделяй и властвуй" является алгоритм сортировки слиянием, рассмотренный в *разделе 4.5*. Слияние двух отсортированных списков, содержащих по $n/2$ элементов и полученных за время $O(n\lg n)$, занимает только линейное время.

Принцип "разделяй и властвуй" применяется во многих важных алгоритмах, включая сортировку слиянием, быстрое преобразование Фурье и умножение матриц методом Страссена. Но я нахожу этот принцип трудным для практической разработки алгоритмов иных, чем двоичный поиск и его разновидности. Возможность анализа алгоритмов типа "разделяй и властвуй" зависит от нашего умения выяснить асимптотику рекуррентных соотношений, определяющих сложность таких рекурсивных алгоритмов.

4.10.1. Рекуррентные соотношения

Временная сложность многих алгоритмов типа "разделяй и властвуй" естественно формируется рекуррентными соотношениями. Оценка рекуррентных соотношений является важным аспектом для понимания, в каких обстоятельствах от алгоритмов типа "разделяй и властвуй" можно ожидать хорошую производительность, а также является важным инструментом для общего анализа. Читатели, которых идея анализа не приводит в особый восторг, могут пропустить этот раздел, но должное представление о разработке можно получить, лишь понимая поведение рекуррентных соотношений.

Что же собой представляет рекуррентное соотношение? Это уравнение, которое определено посредством самого себя. В качестве примера рекуррентного соотношения можно привести последовательность чисел Фибоначчи, определяемую равенством $F_n = F_{n-1} + F_{n-2}$ (см. *раздел 8.1.1*). С помощью рекуррентных соотношений можно выразить также многие другие аналитические функции. В частности, посредством рекуррентного соотношения можно представить любой многочлен, например, линейную функцию:

$$a_n = a_{n-1} + 1, a_1 = 1 \longrightarrow a_n = n$$

Любую показательную функцию также можно выразить посредством рекуррентного соотношения, например:

$$a_n = 2a_{n-1}, a_1 = 1 \longrightarrow a_n = 2^{n-1}$$

Наконец, многие необычные функции, которые непросто выразить посредством обычной нотации, можно представить с помощью рекуррентного соотношения, например:

$$a_n = na_{n-1}, a_1 = 1 \longrightarrow a_n = n!$$

Все это означает, что рекуррентные соотношения являются очень гибким средством для представления функций. Кроме рекуррентных соотношений, свойством ссылаться на самих себя также обладают рекурсивные программы или алгоритмы, как можно видеть по общему корню обоих терминов. По существу, рекуррентные соотношения предоставляют способ анализировать рекурсивные структуры, такие как алгоритмы.

4.10.2. Рекуррентные соотношения метода "разделяй и властвуй"

Как уже упоминалось, алгоритмы типа "разделяй и властвуй" разбивают задачу на несколько (скажем, a) меньших подзадач, каждая из которых имеет размер n/b. Кроме этого, слияние решений подзадач в общее решение занимает время $f(n)$. Пусть $T(n)$ — время решения алгоритмом наихудшего случая задачи размером n. Тогда $T(n)$ предоставляется следующим рекуррентным соотношением:

$$T(n) = aT(n/b) + f(n)$$

Рассмотрим примеры использования рекуррентных соотношений для решения задач.

- *Сортировка.* Время исполнения алгоритма сортировки слиянием определяется рекуррентным соотношением $T(n) = 2T(n/2) + O(n)$, т. к. алгоритм рекурсивно разделяет входные данные на равные половины, после чего выполняет слияние частных решений в общее за линейное время. Фактически это рекуррентное соотношение сводится к соотношению $T(n) = O(n\lg n)$, которое было получено ранее.
- *Двоичный поиск.* Время исполнения алгоритма двоичного поиска представляется рекуррентным соотношением $T(n) = T(n/2) + O(1)$, т. к. каждый шаг уменьшения размера задачи вдвое выполняется за линейное время. Фактически это рекуррентное соотношение сводится к соотношению $T(n) = O(n\lg n)$, которое было получено ранее.
- *Быстрое создание пирамиды.* Процедура `bubble_down` (см. листинг 4.5) создает пирамиду из n элементов, создавая две пирамиды, каждая из которых содержит $n/2$ элементов, а потом сливая их с корнем. Эта процедура занимает логарифмическое время. Мы получим рекуррентное соотношение $T(n) = 2T(n/2) + O(\lg n)$. Фактически оно сводится к соотношению $T(n) = O(n)$, которое было получено ранее.
- *Умножение матриц.* Как описано в *разделе 2.5.4*, время исполнения стандартного алгоритма для умножения двух матриц размером $n \times n$ равно $O(n^3)$, т. к. для каждого из n^2 элементов в матрице произведений мы вычисляем скалярное произведение n членов.

Но в книге [Str69] рассматривается алгоритм типа "разделяй и властвуй", который для умножения двух матриц размером $n \times n$ манипулирует произведениями семи матриц размером $n/2 \times n/2$. Временная сложность этого рекуррентного соотношения равна $T(n) = 7T(n/2) + O(n^2)$. Фактически, это рекуррентное соотношение сводится к $T(n) = O(n^{2,81})$, что невозможно предсказать, не решая соотношение.

4.10.3. Решение рекуррентных соотношений типа "разделяй и властвуй" (*)

В действительности, рекуррентные соотношения типа "разделяй и властвуй" в форме $T(n) = aT(n/b) + f(n)$ обычно очень легко решаются, т. к. решения обычно относятся к одному из трех отдельных классов:

1. Если для некоторой константы $\varepsilon > 0$ существует функция $f(n) = O(n^{\log_b a-\varepsilon})$, тогда $T(n) = \Theta(n^{\log_b a})$.
2. Если $f(n) = \Theta(n^{\log_b a})$, тогда $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Если для некоторой константы $\varepsilon > 0$ существует функция $\Omega(n^{\log_b a+\varepsilon})$ и для некоторой константы $c < 1$ существует функция, такая что $f(n/b) \le cf(n)$, тогда $T(n) = \Theta(f(n))$.

Хотя все эти формулы выглядят устрашающе, в действительности их совсем не трудно использовать. Вопрос заключается в определении, какой случай так называемой *основной теоремы* является действительным для данного рекуррентного соотношения. Первый случай применим для создания пирамиды и умножения матриц, а второй случай действителен для сортировки слиянием и двоичного поиска. Третий случай обычно нужен для не столь элегантных алгоритмов, в которых затраты на слияние подзадач превышают затраты на все остальные операции.

Основную теорему можно представлять себе в виде черного ящика, которым мы умеем пользоваться, но устройство которого нам неизвестно. Однако после некоторого размышления появляется понимание, как работает основная теорема.

На рис. 4.9 показано рекурсивное дерево для типичного алгоритма типа "разделяй и властвуй", выражаемого рекуррентным соотношением $T(n) = aT(n/b) + f(n)$.

Задача размером в n элементов разбивается на a подзадач размером n/b элементов. Каждая подзадача размером в k элементов выполняется за время $O(f(k))$. Общее время исполнения алгоритма будет равно сумме временных затрат на выполнение подзадач, сложенной с накладными расходами на создание рекурсивного дерева. Высота этого дерева равна $h = \log_b n$, а количество листьев равно $a^h = a^{\log_b n}$. Посредством определенных алгебраических манипуляций последнее выражение можно упростить до $n^{\log_b a}$.

Три случая основной теоремы соответствуют трем разным ситуациям, которые могут доминировать в зависимости от a, b и $f(n)$:

1. *Слишком много листьев.* Если количество листьев превышает сумму затрат на внутреннюю обработку, то общее время исполнения будет равно $O(n^{\log_b a})$.

2. Одинаковый объем работы на каждом уровне. По мере прохождения вниз по дереву размер каждой задачи уменьшается, но количество задач, подлежащих решению, увеличивается. Если сумма затрат на внутреннюю обработку одинакова для каждого уровня, то общее время исполнения вычисляется умножением затрат на каждом уровне ($n^{\log_b n}$) на количество уровней ($\log_b n$) и будет равно $O(n^{\log_b a} \lg n)$.

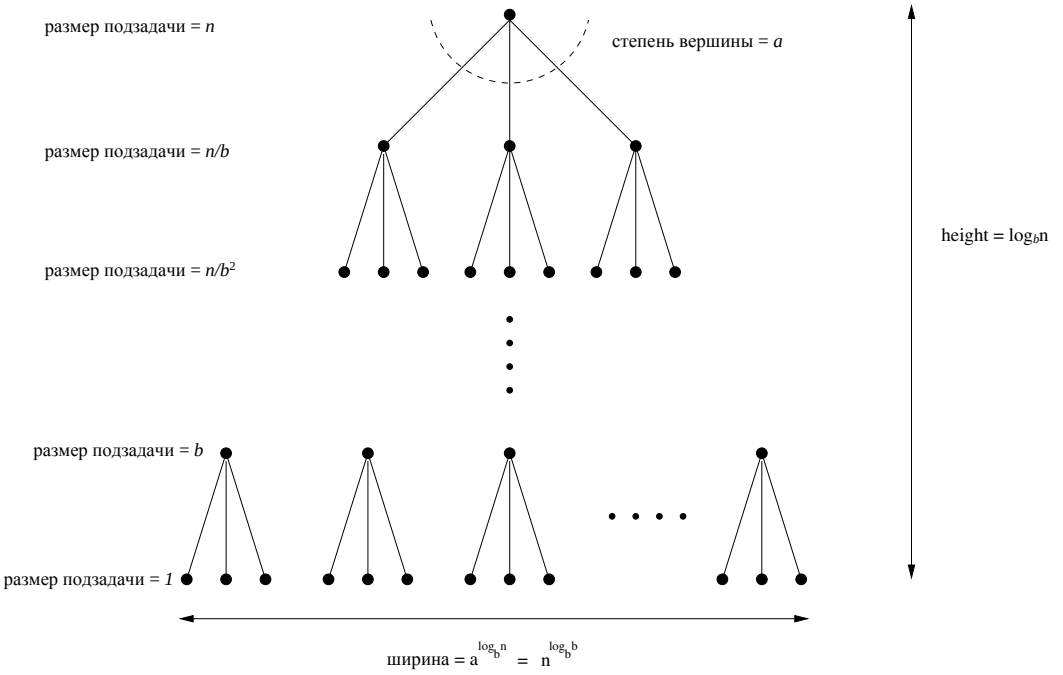


Рис. 4.9. Рекурсивное дерево, полученное в результате разложения каждой задачи размером n на a задач размером n/b

3. *Слишком большое время обработки корня.* Если с возрастанием n затраты на внутреннюю обработку возрастают быстрыми темпами, то будут доминировать затраты на обработку корня. В таком случае общее время исполнения будет равно $O(f(n))$.

Замечания к главе

Из алгоритмов сортировки, которые не рассмотрены в этой главе, наиболее интересен *алгоритм сортировки методом Шелла* (shellsort), представляющий собой более эффективную версию алгоритма сортировки вставками, и *алгоритм поразрядной сортировки* (radix sort), являющийся эффективным алгоритмом для сортировки строк. Узнать больше об этих и всех других алгоритмах сортировки можно в книге [Knu98], содержащей сотни страниц интересного материала по сортировке.

В том виде, в каком он реализован в этой главе, алгоритм сортировки слиянием копирует сливаемые элементы во вспомогательный буфер, чтобы не потерять оригинальные значения сортируемых элементов. Посредством сложных манипуляций с буфером, этот алгоритм можно реализовать для сортировки элементов массива, не используя больших объемов дополнительной памяти. В частности, алгоритм Кронрода (Kronrod) для слияния в памяти рассматривается в книге [Knu98].

Рандомизированные алгоритмы рассматриваются более подробно в книгах [MR95] и [MU05]. Задача подбора болтов и гаек была впервые представлена в книге [Raw92]. Сложный, но детерминированный алгоритм для ее решения с временной сложностью $O(n\log n)$ рассматривается в книге [KMS96].

Более подробное обсуждение алгоритмов типа "разделяй и властвуй" можно найти в книгах [CLRS01], [KT06] и [Man89]. Отличный обзор основной теоремы дается в книге [CLRS01].

4.11. Упражнения

Применение сортировки

1. [3] Вам нужно разделить $2n$ игроков на две команды по n игроков в каждой. Каждому игроку присвоен числовой рейтинг, указывающий его игровые способности. Нужно разделить игроков наиболее несправедливым способом, т. е. создать самое большое неравенство игровых способностей между командой A и командой B. Покажите, как можно решить данную задачу за время $O(n\log n)$.

2. [3] Для каждой из следующих задач предоставьте алгоритм, который находит требуемые числа за данное время. Чтобы уменьшить объем решений, можете свободно использовать алгоритмы из этой книги в качестве процедур. Например, для множества $S = \{6, 13, 19, 3, 8\}$ разность $19 - 3$ максимальна, а разность $8 - 6$ — минимальна.

   а) Пусть S — *неотсортированный* массив n целых чисел. Предоставьте алгоритм для поиска пары элементов $x, y \in S$ с наибольшей разностью $|x - y|$. Время исполнения алгоритма в наихудшем случае должно быть равным $O(n)$.

   б) Пусть S — *отсортированный* массив n целых чисел. Предоставьте алгоритм для поиска пары элементов $x, y \in S$ с наибольшей разностью $|x - y|$. Время исполнения алгоритма в наихудшем случае должно быть равным $O(1)$.

   в) Пусть S — *неотсортированный* массив n целых чисел. Предоставьте алгоритм для поиска пары элементов $x, y \in S$ с наименьшей разностью $|x - y|$ для $x \neq y$. Время исполнения алгоритма в наихудшем случае должно быть равным $O(n\log n)$.

   г) Пусть S — *отсортированный* массив n целых чисел. Предоставьте алгоритм для поиска пары элементов $x, y \in S$ с наименьшей разностью $|x - y|$ для $x \neq y$. Время исполнения алгоритма в наихудшем случае должно быть равным $O(n)$.

3. [3] Для входной последовательности из $2n$ действительных чисел разработайте алгоритм с временем исполнения $O(n\log n)$, который разбивает эти числа на n пар таким образом, чтобы минимизировать максимальную сумму значений в парах. Например, рассмотрим множество чисел {1, 3, 5, 9}. Эти числа можно разбить на следующие наборы: ({1, 3}, {5, 9}), ({1, 5},{3, 9}) и ({1, 9},{3, 5}). Суммы значений пар в этих наборах равны (4, 14), (6, 12) и (10, 8). Таким образом, в третьем наборе максимальная сумма равна 10, что является минимумом для всех наборов.

4. [3] Допустим, что у нас есть n пар элементов, где первый член пары является числом, а второй — одним из трех цветов: красный, синий или желтый. Эти пары элементов отсортированы по числу. Разработайте алгоритм с временем исполнения $O(n)$ для сортировки пар элементов по цвету (красный предшествует синему, а синий желтому) таким образом, чтобы сохранить сортировку по числам для одинаковых цветов.

   Например: последовательность (1, синий), (3, красный), (4, синий), (6, желтый), (9, красный) после сортировки будет такой: (3, красный), (9, красный), (1, синий), (4, синий), (6, желтый).

5. [3] *Модой* набора чисел называют число с наибольшим количеством вхождений в набор. Например, модой набора (4, 6, 2, 4, 3, 1) является число 4. Разработайте эффективный алгоритм поиска моды набора из n чисел.

6. [3] Дано: два набора элементов S_1 и S_2 (оба размером n) и число z. Опишите алгоритм с временем исполнения $O(n\log n)$ для определения, существует ли пара элементов, один из набора S_1, а другой из набора S_2, сумма которых равна x. (Чтобы решение было зачтено частично, алгоритм может иметь время исполнения $\Theta(n^2)$.)

7. [3] Предложите краткое описание метода для решения каждой из следующих задач. Укажите степень сложности в наихудшем случае для каждого из ваших методов.

   а) Вам дали огромное количество телефонных счетов и столь же огромное количество чеков по оплате этих счетов. Узнайте, кто не оплатил свой телефонный счет.

   б) Вам дали список всех книг школьной библиотеки, в котором указаны название каждой книги, ее автор, идентификационный номер и издательство. Также вам дали список 30 издательств. Узнайте, сколько книг в библиотеке были выпущены каждым издательством.

   в) Вам дали все карточки регистрации выдачи книг из библиотеки института за последний год, на каждой из которых указаны имена читателей, бравших данную книгу. Определите, сколько людей брали из библиотеки хотя бы одну книгу.

8. [4] Имеется набор S, содержащий n действительных чисел, и действительное число x. Разработайте алгоритм для определения, содержит ли набор S два таких элемента, сумма которых равна x.

   а) Допустим, что набор S не отсортирован. Разработайте алгоритм для решения задачи за время $O(n\log n)$.

   б) Допустим, что набор S отсортирован. Разработайте алгоритм для решения задачи за время $O(n)$.

9. [4] Разработайте эффективный алгоритм для вычисления объединения множеств A и B, где $n = \max(|A|,|B|)$. Выход должен быть представлен в виде массива элементов, образующих объединение множеств и входящих в это объединение больше, чем один раз.

   а) Допустим, что множества A и B не отсортированы. Разработайте алгоритм для решения задачи за время $O(n\log n)$.

   б) Допустим, что множества A и B отсортированы. Разработайте алгоритм для решения задачи за время $O(n)$.

10. [5] Имеется набор S, содержащий n целых чисел, и целое число T. Разработайте алгоритм с временем исполнения $O(n^{k-1}\log n)$ для определения, равна ли сумма двух целых чисел из S целому числу T.

11. [6] Разработайте алгоритм с временем исполнения $O(n)$, позволяющий найти все элементы, входящие больше чем $n/2$ раза в список из n элементов. Потом разработайте алгоритм с временем исполнения $O(n)$, позволяющий найти все элементы, входящие в список из n элементов больше чем $n/4$ раза.

Пирамиды

12. [3] Разработайте алгоритм для поиска k наименьших элементов в неотсортированном наборе из n целых чисел за время $O(n+ k\log n)$.

13. [5] Вы можете сохранить набор из n чисел в виде невозрастающей бинарной пирамиды или отсортированного массива. Для каждой из перечисленных задач укажите, какая из

этих структур данных является лучшей, или не имеет значения, какую из них использовать. Обоснуйте свои ответы.

а) Найти наибольший элемент.

б) Удалить элемент.

в) Сформировать структуру.

г) Найти наименьший элемент.

14. [5] Разработайте алгоритм с временем исполнения $O(n\log k)$ для слияния k отсортированных списков с общим количеством n элементов в один отсортированный список. (Подсказка: используйте пирамиду, чтобы ускорить работу простейшего алгоритма со временем исполнения $O(kn)$.)

15. [5] а) Разработайте эффективный алгоритм для поиска второго по величине элемента из n элементов. Задача решается меньше чем за $2n - 3$ сравнений.

б) Потом разработайте эффективный алгоритм для поиска третьего по величине элемента из n элементов. Сколько операций сравнения выполняет ваш алгоритм в наихудшем случае? Приходится ли вашему алгоритму в процессе работы находить максимальный и второй по величине элементы?

Быстрая сортировка

16. [3] Используя применяемый в быстрой сортировке принцип разбиения основной задачи на меньшие подзадачи, разработайте алгоритм для определения срединного элемента (median) массива n целых чисел с ожидаемым временем исполнения $O(n)$. (Подсказка: нужно ли исследовать обе стороны раздела?)

17. [3] Срединным элементом n значений является $\lceil n/2 \rceil$-е наименьшее значение.

а) Допустим, что алгоритм быстрой сортировки всегда выбирает в качестве элемента-разделителя срединное значение текущего подмассива. Сколько операций сравнений выполнит алгоритм быстрой сортировки в наихудшем случае при таком условии?

б) Допустим, что алгоритм быстрой сортировки всегда выбирает в качестве элемента-разделителя $\lceil n/3 \rceil$-е наименьшее значение текущего подмассива. Сколько операций сравнений выполнит алгоритм быстрой сортировки в наихудшем случае при таком условии?

18. [5] Дан массив A из n элементов, каждый из которых окрашен в один из трех цветов: красный, белый или синий. Нужно отсортировать элементы по цвету в следующем порядке: красные, белые, синие. Разрешены только две операции:

- *examine(A,i)* — возвращает цвет i-го элемента массива A.
- *swap(A,i,j)* — меняет местами i-й и j-й элементы массива A.

Создайте эффективный алгоритм сортировки элементов в указанном порядке за линейное время.

19. [5] *Инверсией* перестановки является пара элементов, расположенных в неправильном порядке.

а) Покажите, что в перестановке из n элементов может быть самое большее $n(n-1)/2$ инверсий. Какая перестановка (или перестановки) может содержать точно $n(n-1)/2$ инверсий?

б) Допустим, что P является перестановкой, а P^r — инволюция этой перестановки. Покажите, что P и P^r содержат в точности $n(n-1)/2$ инверсий.

(в) На основе предыдущего результата докажите, что ожидаемое количество инверсий в произвольной перестановке равно $n(n-1)/4$.

20. [3] Предоставьте эффективный алгоритм для упорядочивания n элементов таким образом, чтобы все отрицательные элементы находились перед всеми положительными элементами. Использование вспомогательного массива для временного хранения элементов не разрешается. Определите время исполнения вашего алгоритма.

Другие алгоритмы сортировки

21. [5] Устойчивыми называются такие алгоритмы сортировки, которые оставляют элементы с одинаковыми ключами в таком же порядке, в каком они находились до сортировки. Объясните, что нужно сделать, чтобы обеспечить устойчивость алгоритма сортировки слиянием.

22. [3] Продемонстрируйте, что n положительных целых чисел в диапазоне от 1 до k можно отсортировать за время $O(n\log k)$. Интересен случай $k \ll n$.

23. [5] Нам нужно отсортировать последовательность S из n целых чисел, содержащую много дубликатов; количество различных целых чисел в S равно $O(\log n)$. Разработайте алгоритм для сортировки таких последовательностей с временем исполнения в наихудшем случае $O(n\log\log n)$.

24. [5] Пусть $A[1..n]$ — массив, в котором первые $n-\sqrt{n}$ элементов уже отсортированы. О порядке остальных элементов нам ничего не известно. Разработайте алгоритм для сортировки массива A за значительно лучшее время, чем $n\log n$.

25. [5] Допустим, что массив $A[1..n]$ может содержать числа из множества $\{1, ..., n^2\}$, но в действительности содержит самое большее $\log\log n$ этих чисел. Разработайте алгоритм для сортировки массива A за время значительно меньшее, чем $O(n\log n)$.

26. [5] Необходимо отсортировать последовательность нулей (0) и единиц (1) посредством сравнений. При каждом сравнении значений x и y алгоритм определяет, какое из следующих отношений имеет место: $x < y$, $x = y$ или $x > y$.

а) Разработайте алгоритм. Для сортировки данной последовательности за $n-1$ сравнений в наихудшем случае. Докажите, что ваш алгоритм является оптимальным.

б) Разработайте алгоритм для сортировки данной последовательности за $2n/3$ сравнений в среднем случае (допуская, что каждый из n элементов может быть 0 или 1 с одинаковой вероятностью). Докажите, что ваш алгоритм является оптимальным.

27. [6] Пусть P — простой, но не обязательно выпуклый, многоугольник, а q — произвольная точка, не обязательно находящаяся в P. Разработайте эффективный алгоритм поиска прямой линии, начинающейся в точке q и пересекающей наибольшее количество ребер многоугольника P. Иными словами, в каком направлении нужно целиться из ружья, находясь в точке q, чтобы пуля пробила наибольшее количество стен? Прохождение пули через одну из вершин многоугольника P засчитывается, как пробивание только одной стены. Для решения этой задачи возможно создание алгоритма с временем исполнения $O(n\log n)$.

Нижние пределы

28. [5] В одной из моих научных статей (см. книгу [Ski88]) я привел пример алгоритма сортировки методом сравнений с временной сложностью $O(n\log(\sqrt{a}))$.

    Почему такой алгоритм оказался возможным, несмотря на то, что нижний предел сортировки равен $\Omega(n\log n)$?

29. [5] Один из ваших студентов утверждает, что он разработал новую структуру данных для очередей с приоритетами, которая поддерживает операции *insert*, *maximum* и *extract-max*, с временем исполнения в наихудшем случае $O(1)$ для каждой. Докажите, что он ошибается. (Подсказка: для доказательства просто подумайте, каким образом такое время исполнения отразится на нижнем пределе времени исполнения сортировки, равном $\Omega(n\log n)$.)

Поиск

30. [3] База данных содержит записи о 10 000 клиентов в отсортированном порядке. Из них сорок процентов считаются хорошими клиентами, т. е. на них приходится в сумме 60% обращений к базе данных. Такую базу данных и поиск в ней можно реализовать двумя способами:

    - Поместить все записи в один массив и выполнять поиск требуемого клиента посредством двоичного поиска.
    - Поместить хороших клиентов в один массив, а остальных в другой. Двоичный поиск сначала выполняется в первом массиве, и только в случае отрицательного результата — во втором.

    Выясните, какой из этих подходов дает лучшую ожидаемую производительность. Будут ли результаты иными, если в обоих случаях вместо двоичного поиска применить линейный поиск в неотсортированном массиве?

31. [3] Допустим, имеется массив A отсортированных n чисел, циклически сдвинутых вправо на k позиций. Например, последовательность {35, 42, 5, 15, 27, 29} представляет собой отсортированный массив, циклически сдвинутый вправо на $k = 2$ позиций, а последовательность {27, 29, 35, 42, 5, 15} — массив, циклически сдвинутый на $k = 4$ позиций.

    а) Допустим, что значение k известно. Разработайте алгоритм с временем исполнения $O(1)$ для поиска наибольшего числа в массиве A.

    б) Допустим, что значение k неизвестно. Разработайте алгоритм с временем исполнения $O(\lg n)$ для поиска наибольшего числа в массиве A. Чтобы решение было зачтено частично, алгоритм может иметь время исполнения $O(n)$.

32. [3] В игре "20 вопросов" первый игрок задумывает число от 1 до n. Второй игрок должен угадать это число, задав как можно меньше вопросов, требующих ответа "да" или "нет". Предполагается, что оба играют честно.

    а) Какой должна быть оптимальная стратегия второго игрока, если число n известно?

    б) Какую стратегию можно применить, если число n неизвестно?

33. [5] Дана отсортированная последовательность $\{a_1, a_2, \ldots, a_n\}$ *разных* целых чисел. Разработайте алгоритм с временем исполнения $O(\lg n)$ для определения, содержит ли массив такой индекс i, для которого $a_i = i$. Например, в массиве $\{-10, -3, 3, 5, 7\}$, $a_3 = 3$. Но массив {2, 3, 4, 5, 6, 7} не имеет такого индекса i.

34. [5] Дана отсортированная последовательность $\{a_1, a_2, ..., a_n\}$ *разных* целых чисел в диапазоне от 1 до m, где $n < m$. Чтобы решение было зачтено частично, разработайте алгоритм с временем исполнения $O(\lg n)$ для поиска целого числа $x \leq m$, отсутствующего в этой последовательности. Чтобы решение было засчитано полностью, алгоритм должен находить наименьшее такое целое число.

35. [5] Пусть M— матрица размером $n \times m$, в которой элементы каждой строки отсортированы в возрастающем порядке слева направо, а элементы каждого столбца отсортированы в возрастающем порядке сверху вниз. Разработайте эффективный алгоритм для определения местонахождения целого числа x в матрице M или для определения, что матрица не содержит данное число. Сколько сравнений числа x с элементами матрицы выполняет ваш алгоритм в наихудшем случае?

Задачи по реализации

36. [5] Возьмем двумерный массив A размером $n \times n$, содержащий целые числа (положительные, отрицательные и ноль). Допустим, что элементы в каждой строке данного массива отсортированы в строго возрастающем порядке, а элементы каждого столбца — в строго убывающем. (Соответственно, строка или столбец не может содержать двух нулей.) Опишите эффективный алгоритм для подсчета вхождений элемента 0 в массив A. Выполните анализ времени исполнения данного алгоритма.

37. [6] Реализуйте несколько различных алгоритмов сортировки, таких как сортировка методом выбора, сортировка вставками, пирамидальная сортировка, сортировка слиянием и быстрая сортировка. Экспериментальным путем оцените сравнительную производительность этих алгоритмов в простом приложении, считывающим текстовый файл большого объема и отмечающим только один раз каждое встречающееся в нем слово. Это приложение можно реализовать, отсортировав все слова в тексте, после чего просканировав отсортированную последовательность для определения одного вхождения каждого отдельного слова. Напишите краткий доклад с вашими выводами.

38. [5] Реализуйте алгоритм внешней сортировки, использующий промежуточные файлы для временного хранения файлов, которые не помещаются в оперативную память. В качестве основы для такой программы хорошо подходит алгоритм сортировки слиянием. Протестируйте свою программу как на файлах с записями малого размера, так и на файлах с записями большого размера.

39. [8] Разработайте и реализуйте алгоритм параллельной сортировки, распределяющий данные по нескольким процессорам. Подходящим алгоритмом будет разновидность алгоритма сортировки слиянием. Оцените ускорение работы алгоритма с увеличением количества процессоров. Потом сравните время исполнения этого алгоритма с временем исполнения реализации чисто последовательного алгоритма сортировки слиянием. Каковы ваши впечатления?

Задачи, предлагаемые на собеседовании

40. [3] Какой алгоритм вы бы использовали для сортировки миллиона целых чисел? Сколько времени и памяти потребует такая сортировка?

41. [3] Опишите преимущества и недостатки наиболее популярных алгоритмов сортировки.

42. [3] Реализуйте алгоритм, который возвращает только однозначные элементы массива.

43. [5] Как отсортировать файл размером в 500 Мбайт на компьютере, оснащенным оперативной памятью размером всего лишь в 2 Мбайт?

44. [5] Разработайте стек, поддерживающий выполнение операций занесения в стек, снятия со стека и извлечения наименьшего элемента. Каждая операция должна иметь постоянное время исполнения. Возможна ли реализация стека, удовлетворяющего этим требованиям?

45. [5] Дана строка из трех слов. Найдите наименьший (т. е. содержащий наименьшее количество слов) отрывок документа, в котором присутствуют все три слова. Предоставляются индексы расположения этих слов в строках поиска, например *word1: (1,4,5), word2: (4,9,10)* и *word3: (5,6,15)*. Все списки отсортированы.

46. [6] Есть 12 монет, одна из которых тяжелее или легче, чем остальные. Найдите эту монету, выполнив лишь три взвешивания.

Задачи по программированию

Эти задачи доступны на сайтах **http://www.programming-challenges.com** и **http://uva.onlinejudge.org**.

1. Vito's Family. 110401/10041.
2. Stacks of Flapjacks. 110402/120.
3. Bridge. 110403/10037.
4. ShoeMaker's Problem. 110405/10026.
5. ShellSort. 110407/10152.

ГЛАВА 5

Обход графов

Графы являются одной из важнейших областей теории вычислительных систем. Это абстрактное понятие, посредством которого можно описывать разнообразные реальные явления — организацию транспортных систем, человеческих взаимоотношений, сети передачи данных и т. п. Возможность формального моделирования такого множества разных реальных структур позволяет программисту решать широкий круг прикладных задач.

Более конкретно, граф $G = (V, E)$ состоит из набора *вершин* V и набора *ребер* E, соединяющих пары вершин. С помощью графов можно представить практически *любые* взаимоотношения. Например, посредством графов можно создать модель сети дорог, представляя населенные пункты вершинами, а дороги между ними — соединяющими соответствующие вершины ребрами. С помощью графов можно также моделировать электронные схемы, представляя компоненты вершинами, а соединения компонентов — ребрами. Такие графы изображены на рис. 5.1.

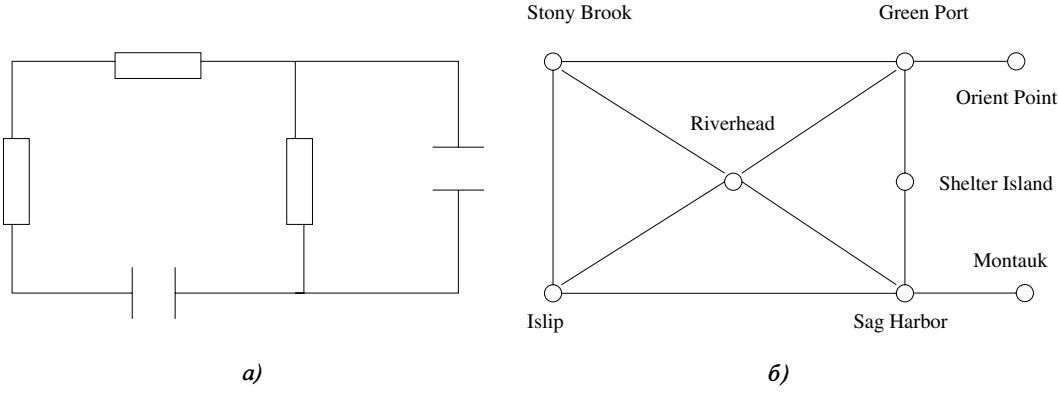


Рис. 5.1. Моделирование электронных схем *(а)* и сети дорог *(б)* посредством графов

Представление задачи в виде графа является ключевым подходом к решению многих алгоритмических задач. Теория графов предоставляет язык для описания свойств взаимоотношений, и просто поразительно, как часто запутанные прикладные задачи поддаются простому описанию и решению посредством применения свойств классических графов.

Разработка по-настоящему оригинальных алгоритмов графов является очень трудной задачей. Ключевым аспектом эффективного использования алгоритмов на графах является правильное моделирование задачи, чтобы можно было воспользоваться уже существующими алгоритмами. Знакомство с разными типами алгоритмических *задач*

важнее знания деталей отдельных алгоритмов, особенно если учесть, что во второй части этой книги можно найти решение задачи по ее названию.

В этой главе рассматриваются основные структуры данных и операции обхода графов, которые можно будет использовать при поиске решений базовых задач на графах. В *главе 6* рассматриваются более сложные алгоритмы для работы с графами: построение минимальных остовных деревьев (minimum spanning tree), кратчайшего пути и потоков в сети. При этом правильное моделирование задачи остается наиболее важным аспектом ее решения. Затратив время на ознакомление с задачами и их решениями во второй части этой книги, вы сэкономите много времени при решении реальных задач в будущем.

5.1. Разновидности графов

Графом называется упорядоченная пара множеств $G = (V, E)$, где V — подмножество вершин (или узлов), а $E = V \times V$ — упорядоченное или неупорядоченное подмножество ребер, соединяющих пары вершин из V. В моделировании дорожной сети вершины представляют населенные пункты (или пересечения), определенные пары которых соединены дорогами (ребрами). В анализе исходного кода компьютерной программы вершины могут представлять операторы языка, а ребра будут соединять операторы x и y, если y выполняется после x. В анализе человеческих взаимоотношений вершины представляют отдельных людей, а ребра соединяют тех, кто близок друг другу.

На выбор конкретного типа структуры данных для представления графов и алгоритмов для работы с ними оказывают влияние несколько фундаментальных свойств графов. Поэтому первым шагом в решении любой задачи на графах будет определение подходящего типа графа. На рис. 5.2 показано несколько основных типов графов, а далее дано их краткое описание.

- *Неориентированные/ориентированные.* Граф $G = (V, E)$ является *неориентированным* (undirected), если из $(x, y) \in E$ следует, что (y, x) также является членом E. В противном случае говорят, что граф *ориентированный* (directed). Дорожные сети между городами обычно неориентированные, т. к. по любой обычной дороге можно двигаться в обоих направлениях. А вот дорожные сети внутри городов почти всегда ориентированы, т. к. в большинстве городов найдется, по крайней мере, несколько улиц с односторонним движением. Графы последовательности исполнения программ обычно ориентированы, т. к. программа исполняется от одной строчки кода к другой в одном направлении и меняет направление исполнения только при ветвлениях. Большинство графов, рассматриваемых в теории графов, являются неориентированными.
- *Взвешенные/невзвешенные.* Каждому ребру (или вершине) *взвешенного* (weighted) графа G присваивается числовое значение, или вес. Например, в зависимости от приложения, весом ребер графа дорожной сети может быть их протяженность, максимальная скорость движения, пропускная способность и т. п. Разные вершины и ребра *невзвешенных* (unweighted) графов не различаются по весу.

  Разница между взвешенными и невзвешенными графами становится особенно очевидной при поиске кратчайшего маршрута между двумя вершинами. В случае не-

взвешенных графов самый короткий маршрут должен состоять из наименьшего количества ребер, и его можно найти посредством поиска в ширину, рассматриваемого далее в этой главе. Поиск кратчайшего пути во взвешенных графах требует использования более сложных алгоритмов и рассматривается в *главе 6*.

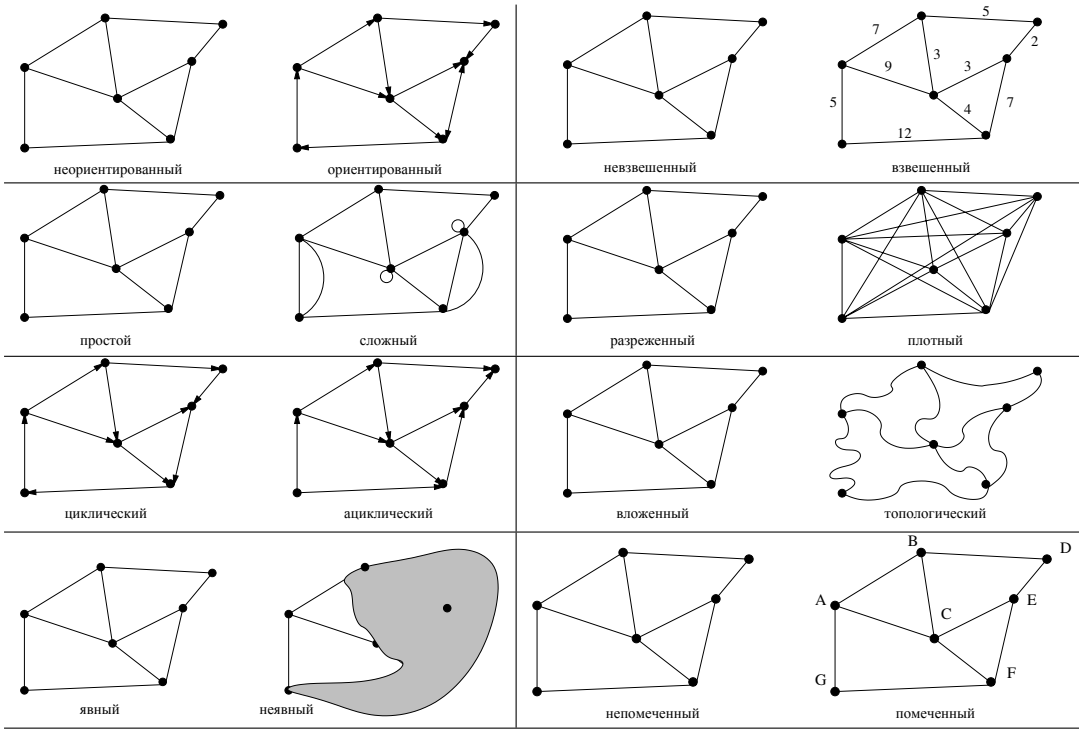


Рис. 5.2. Основные разновидности графов

- *Простые/сложные.* Наличие ребер некоторых типов затрудняют работу с графами. *Петлей* (self-loop, loop) называется ребро (x, x), т. е. ребро, имеющее только одну вершину. *Кратными* (multiedge) называются ребра, соединяющие одну и ту же пару вершин (x, y).

  Наличие в графе обеих этих структур требует особого внимания при реализации алгоритма работы с графом, поэтому графы, которые не содержат их, называются *простыми* (simple) или обыкновенными.

- *Разреженные/плотные.* Граф является *разреженным* (sparse), когда в действительности ребра определены только для малой части возможных пар вершин ($\binom{n}{2}$ для простого неориентированного графа из n вершин). Граф, у которого ребра определены для большей части возможных пар вершин, называется *плотным* (dense). Не существует четкой границы между разреженными и плотными графами; но, как правило, у плотных графов отношение количества ребер к количеству вершин обычно описывается квадратичной функцией, а у разреженных — линейной.

  Как правило, разреженность графов диктуется конкретным приложением. Графы дорожных сетей должны быть разреженными по причине физического и практиче-

ского ограничения на количество дорог, которые могут пересекаться в одной точке. На самом ужасном перекрестке, который мне когда-либо приходилось проезжать, сходилось всего лишь семь дорог. Подобным образом количество проводников электрической или электронной схемы, которые можно соединить в одной точке, также ограничено, за исключением, возможно, проводов питания или заземления.

- *Циклические/ациклические.* *Ациклический* (acyclic) граф не содержит циклов. *Деревья* являются связными ациклическими неориентированными графами. Деревья представляют собой самые простые графы, рекурсивные по своей природе, т. к., разорвав любое ребро, мы получим два меньших дерева.

  Для обозначения ориентированных ациклических графов часто используется аббревиатура DAG (directed acyclic graph). Графы DAG часто возникают в задачах календарного планирования, где ориентированное ребро (x, y) обозначает, что мероприятие x должно выполниться раньше, чем мероприятие y. Для соблюдения таких ограничений на предшествование вершины графа DAG упорядочиваются операцией, называющейся *топологической сортировкой* (topological sort). Топологическая сортировка обычно является первой выполняемой операцией в любом алгоритме на графе DAG (подробности см. в *разделе 5.10.1*).

- *Вложенные/топологические.* Граф является *вложенным* (embedded), если его вершинам и ребрам присвоены геометрические позиции. Таким образом, любое визуальное изображение графа является *укладкой* (embedding), но это обстоятельство необязательно является важным для алгоритма.

  Иногда структура графа полностью определяется геометрией его укладки. Например, если у нас имеется набор точек в плоскости и мы ищем самый короткий маршрут их обхода (т. е. решаем задачу коммивояжера), то в основе решения будет лежать *полный граф* (complete), т. е. граф, в котором каждая вершина соединена со всеми остальными. Веса обычно определяются расстоянием между каждой парой точек.

  Другим примером топологии графа, имеющей геометрическое происхождение, являются решетки точек. В большинстве задач выполняется обход соседних точек решетки, вследствие чего ребра определяются неявно, исходя из геометрии.

- *Неявные/явные.* Некоторые графы не создаются заранее с целью последующего обхода, а возникают по мере решения задачи. Хорошим примером такого графа будет граф поиска с возвратом. Вершины этого неявного графа поиска представляют состояния вектора поиска, а ребра соединяют пары состояний, которые можно генерировать непосредственно друг из друга. Часто бывает, что работать с неявным графом проще (в силу отсутствия необходимости сохранять его полностью), чем явным образом создавать граф для последующего анализа.

- *Помеченные/непомеченные.* В *помеченном* (labeled) графе каждая вершина имеет свою метку (идентификатор), что позволяет отличать ее от других вершин. В *непомеченных* (unlabeled) графах такие обозначения не применяются.

  Вершины графов, возникающие в прикладных задачах, часто помечаются значащими метками, например, названиями городов в графе транспортной сети. Распространенной задачей является проверка на изоморфизм, т. е. выяснение, является ли

топологическая структура двух графов идентичной, если игнорировать метки графов. Такие задачи обычно следует решать методом поиска с возвратом, пытаясь присвоить каждой вершине каждого графа такую метку, чтобы структуры были идентичными.

5.1.1. Граф дружеских отношений

Чтобы продемонстрировать важность моделирования задачи должным образом, рассмотрим граф, в котором вершины представляют людей, а вершины соединяются ребрами в тех случаях, когда между обозначаемыми соответствующими вершинами людьми существуют дружеские отношения. Такие графы называются *социальными сетями* и их можно четко определить для любого набора людей, будь то ваши соседи, однокурсники или коллеги, или жители всего земного шара. В последние годы возникла целая наука анализа социальных сетей, т. к. многие интересные аспекты поведения людей лучше всего поддаются пониманию в виде свойств графа дружеских отношений. Пример графа дружеских отношений между людьми представлен на рис. 5.3.

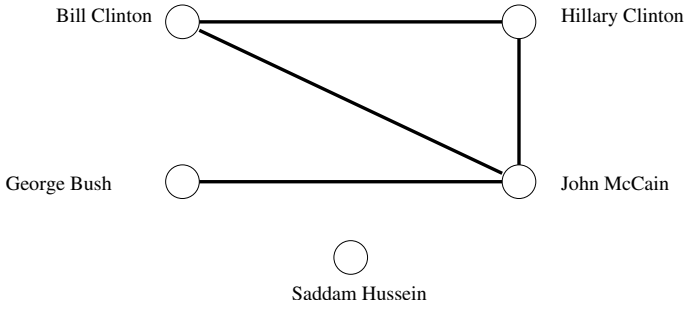


Рис. 5.3. Пример графа дружеских отношений

Для решения реальных задач в большинстве случаев применяются графы разреженного типа. Граф дружеских отношений является хорошим примером разреженных графов. Даже самый общительный человек в мире знает лишь незначительную часть населения планеты.

Мы воспользуемся графами дружеских отношений, чтобы продемонстрировать описанную ранее терминологию графов. Понимание терминологии является важной частью умения работать с графами.

- *Если я твой друг, то означает ли это, что ты мой друг?* Иными словами, мы хотим выяснить, является ли граф ориентированным. Граф называется *неориентированным*, если существование ребра (x, y) всегда влечет за собой существование ребра (y, x). В противном случае говорят, что граф является *ориентированным*. Граф типа "знаю о нем/ней" является ориентированным, т. к. каждый из нас знает о многих людях, которые ничего не знают о нас. Граф типа "провел ночь с ним/ней" предположительно является неориентированным, т. к. для такого занятия требуется партнер. Лично мне хотелось бы, чтобы граф дружеских отношений также был неориентированным.
- *Насколько ты хороший друг?* Во *взвешенных* графах каждому ребру присваивается числовой атрибут. Мы можем смоделировать уровень дружеских отношений, при-

своив каждому ребру соответствующее числовое значение, например, от –10 (враги) до 10 (родные братья). Также, в зависимости от приложения, весом ребер графа дорожной сети может быть их протяженность, максимальная скорость движения, пропускная способность и т. п. Граф называется *невзвешенным*, если все его ребра имеют одинаковый вес.

- *Друг ли я сам себе?* Иными словами, мы хотим выяснить, является ли граф простым, т. е. не содержит петель и кратных ребер. Петлей называется ребро типа (x, x). Иногда люди поддерживают несколько типов отношений друг с другом. Например, возможно, x и y были однокурсниками в институте, а сейчас работают вместе на одном предприятии. Такие взаимоотношения можно моделировать посредством кратных ребер, снабженных разными метками.

  Так как с простыми графами легче работать, то нам может быть лучше объявить, что никто не является другом самому себе.

- *У кого больше всех друзей? Степенью* вершины называется количество ее ребер. В графе дружеских отношений наиболее общительный человек будет представлен вершиной наивысшей степени, а одинокие отшельники — вершинами нулевой степени.

  В *плотных* графах большинство вершин имеет высокую степень, в отличие от *разреженных* графов, в которых количество ребер сравнительно небольшое. В *регулярных*, или однородных, графах все вершины имеют одинаковую степень. Регулярный граф дружеских отношений представляет по-настоящему предельный уровень социальности.

- *Живут ли мои друзья рядом со мной?* Социальные сети в большой степени находятся под влиянием географического фактора. Многие из наших друзей являются таковыми лишь потому, что они просто живут рядом с нами (например, соседи) или когда-то жили вместе с нами (например, соседи по комнате в студенческом общежитии).

  Таким образом, для полного понимания социальной сети требуется вложенный граф, где каждая вершина связана с географической точкой, в которой живет конкретный член сети. Эта географическая информация может быть и не закодирована в графе явным образом, но тот факт, что граф дружеских отношений по своей природе является вложенным, влияет на нашу интерпретацию любого анализа.

- *О, ты ее тоже знаешь?* Службы социальных сетей, такие как Myspace или LinkedIn, основаны на *явном* определении связей между членами этих сетей и их друзьями, которые также являются членами. Графы таких социальных сетей состоят из направленных ребер от члена (вершины) x, заявляющего о своей дружбе с другим членом, к этому члену (вершине) y.

  С другой стороны, полный граф дружеских отношений всего населения Земли является *неявным* по той причине, что каждый знает, кто его друг, но не может знать о дружеских отношениях других людей. Гипотеза шести шагов утверждает, что любые два человека в мире (например, профессор Скиена и президент США) связаны короткой цепочкой из промежуточных людей, но не предоставляет никакой информации, как именно определить эту связывающую цепочку. Самый короткий известный мне путь содержит три звена — Стивен Скиена — Боб МакГрэт — Джон Мар-

бергер — Джордж У. Буш (Steven Skiena — Bob McGrath — John Marberger — George W. Bush). Но может существовать и более короткий, неизвестный мне, путь, например, если Джордж Буш учился в колледже вместе с моим дантистом. Но т. к. граф дружеских отношений является неявным, то проверить эту или другие возможные цепочки не так-то просто.

- *Вы действительно личность или всего лишь лицо в толпе?* Этот вопрос сводится к тому, является ли граф дружеских отношений помеченным или нет. То есть, имеет ли каждая вершина метку, отражающую ее личность, и важна ли такая метка для нашего анализа?

  Во многих исследованиях социальных сетей метки графов не играют большой роли. В качестве метки вершинам графа часто присваивается порядковый номер, что обеспечивает удобство идентификации и в то же время сохраняет анонимность представляемого ею лица. Вы можете протестовать, что номер вас обезличивает, и у вас есть имя, но попробуйте доказать это программисту, который реализует алгоритм. В графе, отображающем распространение инфекционного заболевания, достаточно пометить вершины информацией о том, болен ли данный человек, а его имя роли не играет.

Подведение итогов

С помощью графов можно моделировать большое разнообразие структур и взаимоотношений. Для работы с графами и обмена информацией о них используется терминология теории графов.

5.2. Структуры данных для графов

Выбор правильной структуры данных для графа может иметь огромное влияние на производительность алгоритма. Двумя основными структурами данных для графов являются матрицы смежности (adjacency matrix) и списки смежности (adjacency list), показанные на рис. 5.4.

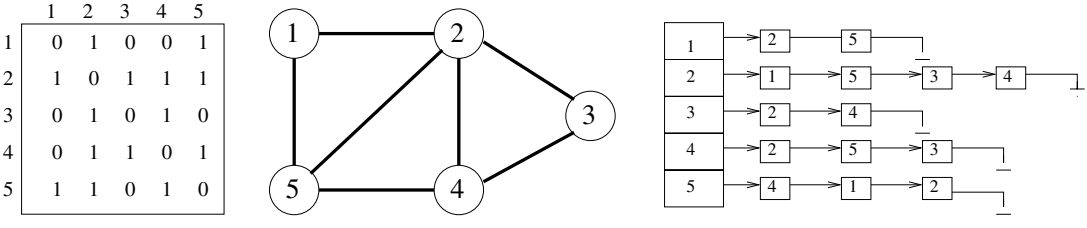


Рис. 5.4. Матрица смежности и список смежности

Предположим, что граф $G = (V, E)$ содержит n вершин и m ребер.

- *Матрица смежности.* Граф G можно представить с помощью матрицы M размером $n \times n$, где элемент $M[i, j] = 1$, если (i, j) является ребром графа G, и 0 в противном случае. Таким образом мы можем дать быстрый ответ на вопрос "Содержит ли граф G ребро (i, j)?", а также быстро отобразить вставки и удаления ребер. Но такая мат-

рица может потребовать большой объем памяти для графов с большим количеством вершин и относительно небольшим количеством ребер.

Рассмотрим, например, граф, представляющий карту улиц Манхэттена. Каждое пересечение улиц отображается на графе в виде вершины, а соседние пересечения соединяются ребрами. Каким будет размер такого графа? Уличная сеть Манхэттена, по сути, представляет собой решетку из 15 авеню, пересекаемых 200 улицами.Это дает нам около 3 000 вершин и 6 000 ребер, т. к. каждая вершина соседствует с четырьмя другими вершинами, а каждое ребро является общим для двух вершин. Эффективное сохранение такого скромного объема данных не должно вызывать никаких проблем, но матрица смежности заняла бы 3 000 × 3 000 = 9 000 000 ячеек, почти все из которых были бы пустыми.

Можно было бы сэкономить некоторый объем памяти, упаковывая в одном слове несколько бит состояния или эмулируя треугольную пирамиду на неориентированных графах. Но с использованием этих методов теряется простота, которая делает матрицу смежности такой привлекательной, и, что более критично, время исполнения на разреженных графах остается по своей сути квадратичным.

- *Списки смежности.* Более эффективным способом представления разреженных графов является использование связных списков для хранения соседствующих вершин. Хотя списки смежности требуют использования указателей, вы легко с ними справитесь, когда наберетесь немного опыта работы со связными структурами данных.

  В списках смежности проверку на присутствие определенного ребра (i, j) в графе G выполнить труднее, т. к. для того, чтобы найти данное ребро, необходимо выполнить поиск по соответствующему списку. Однако разработать алгоритм, не нуждающийся в таких запросах, на удивление легко. Обычно перебираются все ребра графа при одном его обходе в ширину или глубину, и при посещении текущего ребра обновляется его состояние. Преимущества той или иной структуры смежности для представления графов показаны в табл. 5.1.

Таблица 5.1. Сравнение матриц и списков смежности

Задача	Оптимальный вариант
Проверка на вхождение ребра (x, y) в граф	Матрица смежности
Определение степени вершины	Списки смежности
Объем памяти для небольших графов	Списки смежности $(m - n)$, т. к. матрица смежности (n^2)
Объем памяти для больших графов	Матрица смежности (с небольшим преимуществом)
Вставка или удаление ребра	Матрица смежности $O(1)$, т. к. списки смежности $O(d)$
Обход графа	Списки смежности $\Theta(m + n)$, т. к. матрица смежности $\Theta(n^2)$
Пригодность для решения большинства проблем	Списки смежности

Подведение итогов

Для большинства приложений на графах списки смежности являются более подходящей структурой данных, чем матрицы смежности.

В этой главе для представления графов мы будем использовать списки смежности. В частности, представление графа осуществляется таким образом: для каждого графа ведется подсчет количества вершин, каждой из которых присваивается идентификационный номер в диапазоне от 1 до n. Ребра представляются посредством массива связных списков. Соответствующий код показан в листинге 5.1.

Листинг 5.1. Реализация графов посредством списков смежности

```
#define MAXV 1000           /* Максимальное количество вершин */
typedef struct {
   int y;                   /* Информация о смежности */
   int weight;              /* Вес ребра, если есть */
   struct edgenode *next;   /* Следующее ребро в списке */
} edgenode;
typedef struct {
   edgenode *edges[MAXV+1]; /* Информация о смежности */
   int degree[MAXV+1];      /* Степень каждой вершины */
   int nvertices;           /* Количество вершин в графе */
   int nedges;              /* Количество ребер в графе */
   bool directed;           /* Граф ориентированный? */
} graph;
```

Ориентированное ребро (x, y) представляется структурой типа `edgenode` в списке смежности. Поле `degree` содержит степень данной вершины. Неориентированное ребро (x, y) входит дважды в любую структуру графа на основе смежности — один раз в виде y в списке для x и второй раз в виде x в списке для y. Булев флаг `directed` указывает, является ли данный граф ориентированным.

Для демонстрации использования этой структуры данных мы покажем, как выполнить считывание графа из файла. Типичный формат графа (листинг 5.2) таков: первая строчка содержит количество вершин и ребер в графе, а за ней следуют строчки, в каждой из которых указаны вершины очередного ребра.

Листинг 5.2. Формат графов

```
initialize_graph(graph *g, bool directed)
{
   int i;                   /* Счетчик */
   g -> nvertices = 0;
   g -> nedges = 0;
   g -> directed = directed;
   for (i=1; i<=MAXV; i++) g->degree[i] = 0;
   for (i=1; i<=MAXV; i++) g->edges[i] = NULL;
}
```

Собственно чтение графа заключается во вставке каждого ребра в эту структуру. Соответствующий код представлен в листинге 5.3.

Листинг 5.3. Считывание графа

```
read_graph(graph *g, bool directed)
{
   int i;              /* Счетчик */
   int m;              /* Количество вершин */
   int x, y;           /* Вершины в ребре (x,y) */
   initialize_graph(g, directed);
   scanf("%d %d", & (g->nvertices), &m);
   for (i=1; i<=m; i++) {
      scanf("%d %d", &x, &y);
      insert_edge(g, x, y, directed);
   }
}
```

В листинге 5.3 критичной является процедура `insert_edge`. Новый узел `edgenode` вставляется в начало соответствующего списка смежности, т. к. порядок не имеет значения. Процедуре вставки передается параметр в виде булева флага `directed` для обозначения, сколько копий каждого ребра нужно вставить — одну или две. Код процедуры вставки показан в листинге 5.4. Обратите внимание на использование рекурсии для решения этой задачи.

Листинг 5.4. Вставка ребра

```
insert_edge(graph *g, int x, int y, bool directed)
{
   edgenode *p;                    /* Временный указатель */
   p = malloc(sizeof(edgenode));   /* Выделяем память для edgenode*/
   p->weight = NULL;
   p->y = y;
   p->next = g->edges [x] ;
   g->edges[x] = p;                /* Вставка в начало списка */
   g->degree[x] ++;
   if (directed == FALSE)
      insert_edge(g,y,x,TRUE);
   else
      g->nedges ++;
}
```

Для вывода графа на экран достаточно двух вложенных циклов — один для вершин, другой для их смежных ребер (листинг 5.5).

Листинг 5.5. Вывод графа на экран

```
print_graph(graph *g)
{
   int i;           /* Счетчик */
   edgenode *p;     /* Временный указатель */
   for (i=1; i<=g->nvertices; i++) {
      printf("%d: ",i);
```

```
      p = g->edges[i];
      while (p != NULL) {
         printf(" %d",p->y);
         p = p->next;
      }
      printf("\n");
   }
}
```

Было бы разумно использовать хорошо спроектированный тип данных графа в качестве модели для создания своего собственного, а еще лучше, в качестве основы для разрабатываемого приложения. Я рекомендую использовать библиотеку типов LEDA (см. *раздел 19.1.1*) или Boost (см. *раздел 19.1.3*), которые я считаю лучше всего спроектированными структурами данных графов общего назначения, имеющимися в настоящее время. Эти библиотеки, скорее всего, будут более мощными (и, следовательно, будут иметь больший размер и замедлять работу в большей степени), чем вам нужно, но они предоставляют такое количество правильно работающих функций, которые вы, вероятнее всего, не сможете реализовать самостоятельно с такой степенью эффективности.

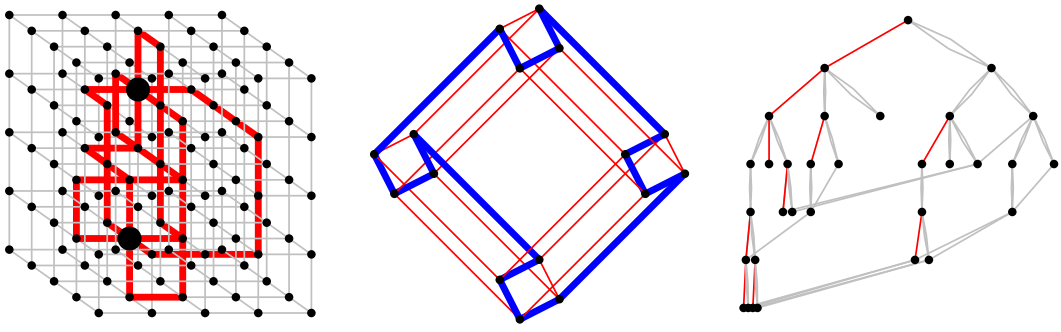
5.3. История из жизни. Жертва закона Мура

Я являюсь автором библиотеки алгоритмов для работы с графами, называющейся Combinatorica (**www.combinatorica.com**), которая предназначена для работы с компьютерной системой алгебраических вычислений Mathematica. Производительность является большой проблемой в Mathematica, вследствие применяемой в ней аппликативной модели вычислений (в ней не поддерживаются операции записи в массивы с постоянным временем исполнения) и накладных расходов на интерпретацию кода (в отличие от компилирования). Код на языке программирования пакета Mathematica обычно исполняется от 1 000 до 5 000 раз медленнее, чем код на языке C.

Эти особенности данного приложения могут значительным образом замедлять его работу. Что еще хуже, Mathematica занимала большой объем памяти, требуя для эффективной работы целых 4 Мбайт оперативной памяти, что в 1990 году, когда я завершил работу над Combinatorica, было потрясающе большим объемом. При этом любая попытка вычислений на достаточно больших структурах неизбежно вызывала переполнение виртуальной памяти. В такой среде мой графический пакет имел надежду эффективно работать только на графах *очень* небольшого размера (рис. 5.5).

Одним из проектных решений, принятых мной вследствие вышеописанных недостатков Mathematica, было использование матриц смежности вместо списков смежности в качестве основной структуры данных графов для Combinatorica. Такое решение может сначала показаться странным. Разве в ситуации нехватки памяти не выгоднее использовать списки смежности, экономя каждый байт? Вообще говоря, да, но ответ не будет таким простым в случае очень малых графов. Представление взвешенного графа, состоящего из n вершин и m ребер посредством списка смежности требует приблизительно $n + 2m$ слов, где составляющая $2m$ отражает требования к памяти для хранения информации о конечных точках и весе каждого ребра. Таким образом, использование списков смежности предоставляет преимущество в отношении объема требуемой па-

мяти только в том случае, если выражение $n + 2m$ значительно меньше, чем n^2. Размер матрицы смежности остается управляемым для $n \leq 100$ и, конечно же, для плотных графов этот размер вдвое меньше, чем размер списков смежности.



а) *б)* *в)*

Рис. 5.5. Репрезентативные графы в Combinatorica: пути, не имеющие общих ребер *(а)*, Гамильтонов цикл в гиперкубе *(б)*, обход в глубину дерева поиска *(в)*

Для меня более насущной заботой была необходимость минимизации накладных расходов, обусловленных использованием интерпретируемого языка. Результаты эталонных тестов представлены в табл. 5.2.

Таблица 5.2. Результаты эталонных тестов старой Comblnatorica на рабочих станциях Sun начального уровня с 1990 по 2004 г. (время исполнения в секундах)

Год	1990	1991	1998	2000	2004
Команда/Машина	Sun-3	Sun-4	Sun-5	Ultra 5	Sun Blade
PlanarQ[GridGraph[4,4]]	234,10	69,65	27,50	3,60	0,40
Length [Partitions[30]]	289,85	73,20	24,40	3,44	1,58
VertexConnectivity [GridGraph[3,3]]	239,67	47,76	14,70	2,00	0,91
RandomPartition[1000]	831,68	267,5	22,05	3,12	0,87

В 1990 году решение двух довольно сложных, но имеющих полиномиальное время выполнения, задач на графах из 9 и 16 вершин занимало на моем настольном компьютере несколько минут! Квадратичный размер структуры данных определенно не мог оказать большое влияние на время исполнения этих задач, т. к. 9×9 равно всего лишь 81. По своему опыту я знал, что язык программирования пакета Mathematica работает лучше со структурами данных постоянного размера, наподобие матриц смежности, чем со структурами данных, имеющими непостоянный размер, какими являются списки смежности.

Тем не менее, несмотря на все эти проблемы с производительностью, библиотека Combinatorica оказалась очень хорошим приложением, и тысячи людей использовали

этот пакет для всевозможных интересных экспериментов с графами. Combinatorica никогда не претендовала на звание высокопроизводительной библиотеки алгоритмов. Большинство пользователей быстро осознало, что вычисления на больших графах нереальны, но, тем не менее, с энтузиазмом работало с этой библиотекой как с инструментом для математических исследований и среды моделирования. Все были довольны.

Но по прошествии нескольких лет пользователи Combinatorica начали спрашивать, почему вычисления на графах небольшого размера занимают так много времени. Это меня не удивляло, так как я знал, что моя программа всегда была медленной. Но почему людям потребовались годы, чтобы заметить это?

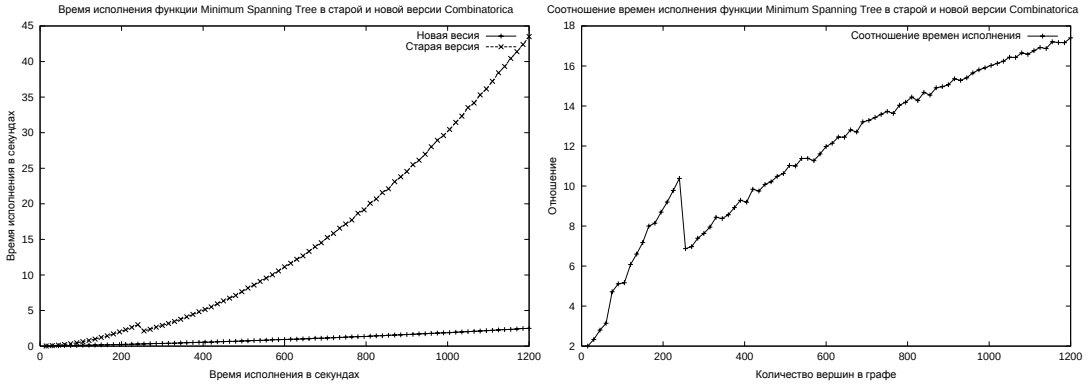
Объяснение заключается в том, что скорость работы компьютеров удваивается приблизительно каждые два года. Это явление носит названия *закона Мура*. Ожидания пользователей относительно производительности прикладных программ возрастают в соответствии с этими улучшениями в аппаратном обеспечении. Частично из-за того, что Combinatorica была рассчитана на работу со структурами данных графов квадратичного размера, она недостаточно хорошо масштабировалась на разреженные графы.

С годами требования пользователей становились все жестче, и, наконец, я решил, что Combinatorica нуждается в обновлении. Мой коллега, Срирам Пемараджу (Sriram Pemmaraju), предложил мне свою помощь. Через десять лет после первоначального выпуска библиотеки мы (преимущественно он) полностью переписали Combinatorica, воспользовавшись более быстрыми структурами данных.

В новой версии Combinatorica для хранения графов используется очень эффективная структура данных в виде списка ребер. Размер списков ребер, как и размер списков смежности, линейно зависит от размера графа (ребра плюс вершины). Это заметно улучшило производительность большинства функций для работы с графами. Повышение производительности особенно драматично для "быстрых" алгоритмов обработки графов. Это алгоритмы с линейным или почти линейным временем исполнения — алгоритмы обхода графов, топологической сортировки и поиска компонент связности и/или двусвязности. Последствия этой модификации проявляются во всем пакете в виде уменьшения времени работы и более экономного расхода памяти. Теперь Combinatorica может обрабатывать графы, которые в 50–100 раз больше, чем те, с которыми могла работать старая версия.

На рис. 5.6, *а* приводится график времени исполнения функции `MinimumSpanningTree` для обеих версий Combinatorica.

Для сравнения производительности новой и старой версий использовались разреженные (решетчатые) графы, разработанные специально, чтобы выделить разницу между этими двумя структурами данных. Да, новая версия библиотеки *намного* быстрее, но обратите внимание, что разница в производительности становится заметной только для графов больших, чем те, для работы с которыми предназначалась старая версия Combinatorica. Но относительная разница во времени исполнения увеличивается с возрастанием n. На рис. 5.6, *б* показано соотношение времени исполнения старой и новой версии в зависимости от размера графа. Разница между линейным и квадратичным ростом размера является асимптотической, поэтому с возрастанием n последствия перехода на новую версию становятся еще более заметными.



а) *б)*

Рис. 5.6. Сравнение производительности старой и новой версий Combinatorica: абсолютное время исполнения для каждой версии *(а)* и соотношение времен исполнения *(б)*

Обратите внимание на странный всплеск на графике при $n \approx 250$. Скорее всего, это следствие перехода между разными уровнями иерархии памяти. Такие явления не редкость в современных сложных компьютерных системах. При разработке структур данных производительность кэша должна быть одним из главных, но не важнейшим, принимаемых во внимание обстоятельств.

Повышение производительности, достигнутое благодаря использованию списков смежности, намного превышает любое улучшение, обусловленное применением кэша.

Из нашего опыта разработки и модифицирования библиотеки Combinatorica можно извлечь такие два основных урока.

- *Чтобы ускорить время исполнения программы, нужно лишь подождать некоторое время.* Передовое аппаратное обеспечение со временем доходит до пользователей всех уровней. В результате улучшений в аппаратном обеспечении, имевших место в течение 15 лет, скорость работы первоначальной версии библиотеки Combinatorica возросла больше чем в 200 раз. В этом контексте дальнейшее повышение производительности вследствие модернизации библиотеки является особенно значительным.
- *Асимптотика в конечном счете является важной.* Я не сумел предвидеть развитие технологии, и это было ошибкой с моей стороны. Хотя никто не может предсказывать будущее, можно с довольно большой степенью уверенности утверждать, что в будущем компьютеры будут обладать большим объемом памяти и работать быстрее, чем компьютеры сейчас. Это дает преимущество более эффективным алгоритмам и структурам данных, даже если сегодня их производительность не намного выше, чем у альтернативных решений. Если сложность реализации вас не останавливает, подумайте о будущем и выберите самый лучший алгоритм.

5.4. История из жизни. Создание графа

— Только на чтение данных у этого алгоритма уходит пять минут. На получение сколько-нибудь интересных результатов не хватит *никакого* времени.

Молодая аспирантка была полна энтузиазма, но не имела ни малейшего понятия о мощи правильно выбранных структур данных.

Как было описано в *разделе 3.6*, мы экспериментировали с алгоритмами для разбиения сетки треугольников на полосы для быстрого рендеринга триангулированных поверхностей. Задачу поиска наименьшего количества полос, которые покрывают каждый треугольник в сетке, можно смоделировать как задачу на графах. В таком графе каждый треугольник сетки представляется вершиной, а смежные треугольники представляются ребром между соответствующими вершинами. Такое представление в виде двойственного графа содержит всю информацию, необходимую для разбиения треугольной сетки на полосы треугольников (рис. 5.7).

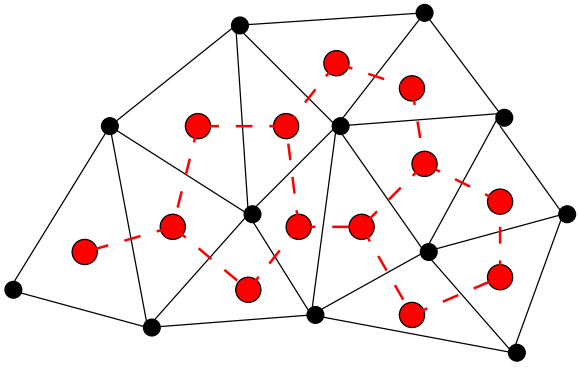


Рис. 5.7. Двойственный граф (пунктирные линии) триангулированной поверхности

Первым шагом в разработке программы, которая генерирует хороший набор полос треугольников, является создание двойственного графа триангулированной поверхности. Решение этой задачи я и поручил аспирантке. Через несколько дней она заявила, что одно лишь создание такого графа для поверхности из нескольких тысяч треугольников занимает около пяти минут процессорного времени.

— Не может быть! — сказал я. — Ты, должно быть, строишь граф крайне нерационально. Какой формат данных ты используешь?

— Вначале идет список 3D-координат используемых в модели вершин, а за ним — список треугольников. Каждый треугольник описывается списком из трех индексов координат вершин. Вот небольшой пример (листинг 5.6).

Листинг 5.6. Пример описания треугольников

```
VERTICES 4
0.000000 240.000000 0.000000
204.000000 240.000000 0.000000
204.000000 0.000000 0.000000
0.000000 0.000000 0.000000
TRIANGLES 2
0 1 3
1 2 3
```

— Понятно. Значит, первый треугольник использует все точки, кроме третьей, т. к. все индексы начинаются с нуля. Два треугольника должны иметь общую сторону, определяемую точками 1 и 3.

— Так оно и есть, — подтвердила она.

— Хорошо. Теперь расскажи мне, как ты строишь двойственный граф.

— Когда я знаю количество имеющихся вершин, я могу игнорировать информацию о вершинах. Геометрическое расположение точек не влияет на структуру графа. В моем двойственном графе будет такое же количество вершин, как и треугольников. Я создаю структуру данных в виде списка смежности, содержащего такое количество вершин. При считывании каждого треугольника я сравниваю его со всеми другими треугольниками, чтобы узнать, не имеют ли они две общие конечные точки. Если имеют, то я добавляю ребро между новым треугольником и этим.

Я не мог сдержать раздражение. — Но ведь именно в этом и заключается твоя проблема! Ты сравниваешь каждый треугольник со всеми остальными треугольниками, вследствие чего создание двойственного графа будет квадратичным по числу треугольников. Считывание входного графа должно занимать линейное время!

— Я не сравниваю каждый треугольник со всеми другими треугольниками. В действительности, в среднем он сравнивается с половиной или с третью других треугольников.

— Превосходно. Тем не менее, у тебя алгоритм имеет квадратичную сложность (т. е. $O(n^2)$). Он слишком медленный.

Признавать свое поражение она не собиралась. — Вы только критикуете мой алгоритм, а можете ли Вы помочь мне исправить его?

Вполне разумно. Я начал думать. Нам был нужен какой-то быстрый способ отбросить большинство треугольников, которые не могли быть смежными с новым треугольником (i, j, k). Что нам в действительности было нужно, так это отдельный список всех треугольников, проходящих через точки i, j и k. По формуле Эйлера для планарных графов средняя точка соединяет меньше чем шесть треугольников. Это позволило бы сравнивать каждый новый треугольник менее чем с двадцатью другими треугольниками.

— Нам нужна структура данных, состоящая из массива элементов для каждой вершины в первоначальном наборе данных. Этим элементом будет список всех треугольников, которые проходят через данную вершину. При считывании нового треугольника мы находим три соответствующих списка в массиве и сравниваем каждый из них с новым треугольником. В действительности, нужно проверить только два из этих трех списков, т. к. любые смежные треугольники будут иметь две общие точки. Нам нужно будет добавить в наш граф признак смежности для каждой пары треугольников, имеющей две общие вершины. Наконец, новый треугольник добавляется в каждый из трех обработанных списков, чтобы они были текущими для считывания следующего треугольника.

Она немного поразмыслила над этим и улыбнулась. — Понятно. Я сообщу Вам о результатах.

На следующий день она доложила, что создание графа занимает несколько секунд даже для больших моделей. Затем она написала хорошую программу для разбиения триангулированной поверхности на полосы треугольников, как описано в *разделе 3.6*.

Урок, который следует извлечь из этой истории, заключается в том, что даже такие элементарные задачи, как инициализация структур данных, могут оказаться узким местом при разработке алгоритмов. Большинство программ, работающих с большими объемами данных, должно иметь линейное или почти линейное время исполнения. Такие высокие требования к производительности не прощают небрежности. Сфокусировавшись на необходимости получения линейной производительности, вы обычно можете найти соответствующий детерминистический или эвристический алгоритм для решения поставленной задачи.

5.5. Обход графа

Самой фундаментальной задачей на графах, возможно, является систематизированное посещение каждой вершины и каждого ребра графа. В действительности, все основные служебные операции по работе с графами (такие как распечатка или копирование графов или преобразования графа из одного представления в другое) являются приложениями обхода графа (graph traversal).

Лабиринты обычно представляются в виде графов, где вершины обозначают пересечения путей, а ребра — пути лабиринта. Таким образом, любой алгоритм обхода графа должен быть достаточно мощным, чтобы вывести нас из произвольного лабиринта. Чтобы обеспечить *эффективность* такого алгоритма, мы должны гарантировать, что не будем постоянно возвращаться в одну и ту же точку, оставаясь в лабиринте навечно. А для *правильности* нашего алгоритма нам нужно выполнять обход лабиринта системным образом, который гарантирует, что мы выберемся из лабиринта. В поисках выхода нам нужно посетить каждую вершину и каждое ребро графа.

Ключевая идея обхода графа — пометить каждую вершину при первом ее посещении и помнить о том, что не было исследовано полностью. Хотя в сказках для обозначения пройденного пути использовались такие способы, как хлебные крошки и нитки, в наших обходах графов мы будем пользоваться булевыми флагами или перечислимыми типами.

Каждая вершина будет находиться в одном из следующих трех состояний:

- *неоткрытая* (undiscovered) — первоначальное, нетронутое состояние вершины;
- *открытая* (discovered) — вершина обнаружена, но мы еще не проверили все инцидентные ей ребра;
- *обработанная* (processed) — все инцидентные данной вершине ребра были посещены.

Очевидно, что вершину нельзя обработать до того, как она открыта, поэтому в процессе обхода графа состояние каждой вершины начинается с неоткрытого, переходит в открытое и заканчивается обработанным.

Нам также нужно иметь структуру, содержащую все открытые, но еще не обработанные вершины. Первоначально открытой считается только одна вершина — начало обход<;1 графа. Для полного исследования вершины v нужно изучить каждое исходящее из нее ребро. Если какие-либо ребра идут к неоткрытой вершине x, то эта вершина помечается как *открытая* и добавляется в список для дальнейшей обработки. Ребра, идущие к *обработанным* вершинам, игнорируются, т. к. их дальнейшее исследование не

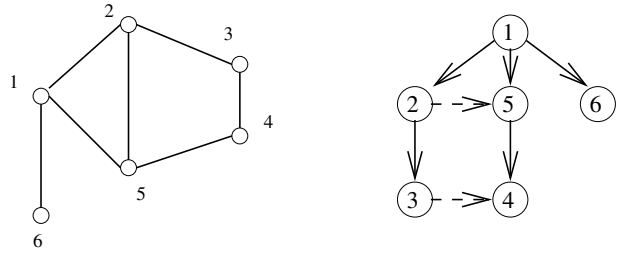
сообщит нам ничего нового о графе. Также можно игнорировать любое ребро, идущее к *открытой*, но не *обработанной* вершине, т. к. эта вершина уже внесена в список вершин, подлежащих обработке.

Каждое неориентированное ребро рассматривается дважды, по одному разу при исследовании каждой из его вершин. Ориентированные ребра рассматриваются только один раз, при исследовании его источника. В конечном счете, все ребра и вершины в связном графе должны быть посещены. Почему? Допустим, что имеется непосещенная вершина *u*, чья соседняя вершина *v* была посещена. Эта соседняя вершина *v* будет со временем исследована, после чего мы непременно посетим вершину *u*. Таким образом, мы в конечном счете найдем все, что можно найти.

Далее мы обсудим механизм работы алгоритмов обхода графов и важность того, в каком порядке выполняется обход.

5.6. Обход в ширину

Обход в ширину (breadth-first traversal) является основой для многих важных алгоритмов для работы с графами. Далее приводится базовый алгоритм обхода графа в ширину. На определенном этапе каждая вершина графа переходит из состояния *неоткрытая* в состояние *открытая*. При обходе в ширину неориентированного графа каждому ребру присваивается направление, от "открывающей" к открываемой вершине. В этом контексте вершина *u* называется родителем (parent) или предшественником (predecessor) вершины *v*, а вершина *v* — потомком вершины *u*. Так как каждый узел, за исключением корня, имеет только одного родителя, получится дерево вершин графа. Это дерево (рис. 5.8) определяет кратчайший путь от корня ко всем другим узлам графа.



а) *б)*

Рис. 5.8. Неориентированный граф и его дерево обхода в ширину

Это свойство делает обход в ширину очень полезным в решении задач поиска кратчайшего пути. В листинге 5.7 приводится псевдокод алгоритма обхода графа в ширину.

Листинг 5.7. Обход графа в ширину

```
BFS(G, s)
   for each vertex u ∈ V[G] − {s} do
      state[u} = "undiscovered"
      p[u] = nil, т. е. в начале вершины-родители отсутствуют
```

```
   state[s]   "discovered"
   p[s] = nil
   Q = {s}
   while Q ≠ ∅ do
      u = dequeue[Q]
        обрабатываем вершину u требуемым образом
      for each v ∈ Adj[u] do
         обрабатываем ребро (u, v) требуемым образом
         if state[v] = "undiscovered" then
            state[v] = "discovered"
            p[v] = u
            enqueue[Q,v]
   state[u] = "processed"
```

Ребра графа, которые не включены в дерево обхода в ширину, также имеют особые свойства. Для неориентированных графов не попавшие в дерево ребра могут указывать только на вершины на том же уровне, что и родительская вершина, или на вершины, расположенные на уровень ниже. Эти свойства естественно следуют из того факта, что каждое ребро в дереве должно быть кратчайшим путем в графе. А для ориентированных графов ребро (*u*, *v*), указывающее в обратном направлении, может существовать в любом случае, когда вершина *v* расположена ближе к корню, чем вершина *u*.

Реализация

Процедура обхода в ширину `bfs` использует два массива булевых значений для хранения информации о каждой вершине графа. Вершина открывается при первом ее посещении. Когда все исходящие из вершины ребра были исследованы, то вершина считается обработанной. Таким образом, в процессе обхода состояние каждой вершины начинается с неоткрытого, переходит в открытое и заканчивается обработанным. Эту информацию можно было бы хранить с помощью одной переменной перечислимого типа, но мы используем две булевы переменные.

```
bool processed[MAXV+1];        /* Обработанные вершины*/
bool discovered[MAXV+1];       /* Открытые вершины */
int parent[MAXV+1];            /* Отношения открытия */
```

Сначала каждая вершина помечается как неоткрытая (листинг 5.8).

Листинг 5.8. Инициализация вершин

```
initialize_search(graph *g)
{
   int i; /* Счетчик */
   for (i=1; i<=g->nvertices; i++)   {
      processed[i] = discovered[i]   FALSE;
      parent[i] = -1;
   }
}
```

После открытия вершина помещается в очередь. Так как эти вершины обрабатываются в порядке FIFO, то первыми обрабатываются вершины, поставленные в очередь пер-

вымй, т. е. ближайшие к корню. Процедура обхода графа в ширину приводится в листинге 5.9.

Листинг 5.9. Обход графа в ширину

```
bfs(graph *g, int start)
{
   queue q;              /* Очередь вершин для обработки  */
   int v;                /* Текущая вершина */
   int y;                /* Следующая вершина */
   edgenode *p;          /* Временный указатель  */
   init_queue(&q);
   enqueue(&q,start);
   discovered[start] = TRUE;
   while (empty_queue(&q) == FALSE) {
      v = dequeue(&q);
      process_vertex_early(v);
      processed[v] = TRUE;
      p = g->edges[v];
      while (p != NULL) {
         y = p->y;
         if ((processed[y] == FALSE) || g->directed)
            process_edge(v,y);
         if (discovered[y] == FALSE) {
            enqueue(&q,y);
            discovered[y] = TRUE;
            parent[y] = v;
         }
         p = p->next;
      }
      process_vertex_late(v);
   }
}
```

5.6.1. Применение обхода

Процедура `bfs` использует функции `process_vertex_early()`, `process_vertex_late()` и `process_edge()`. С их помощью мы можем настроить действие, предпринимаемое процедурой обхода при посещении каждого ребра и вершины. Первоначально вся обработка вершин будет осуществляться при входе, поэтому функция `process_vertex_late()` не выполняет никаких действий (листинг 5.10).

Листинг 5.10. Функция `process_vertex_late()`

```
process_vertex_late(int v)
{
   /* Нет действий */
```

Следующие функции предназначены для вывода на экран (или принтер) каждой вершины и каждого ребра (листинг 5.11).

Листинг 5.11. Функция `process_vertex_early()` и `process_edge()`

```
process_vertex_early(int v)
{
   printf("processed vertex %d\n",v);
}
process_edge(int x, int y)
{
   printf("processed edge (%d,%d)\n",x,y);
}
```

С помощью функции `process_edge()` можно также подсчитывать количество ребер (листинг 5.12).

Листинг 5.12. Функция `process_edge()` для подсчета количества ребер

```
process_edge(int x, int y)
{
   nedges = nedges + 1;
}
```

Разные алгоритмы предпринимают разные действия при посещении вершин и ребер. Использование этих функций предоставляет нам гибкость в настройке требуемого действия.

5.6.2. Поиск путей

Массив `parent`, который строится в процедуре `bfs`, очень полезен для поиска разнообразных путей в графе. Вершина, открывшая вершину i, определяется как `parent[i]`. Так как в процессе обхода открываются все вершины, то каждая вершина, за исключением корневой, имеет родителя. Родительское отношение определяет "дерево открытий", корнем которого является первоначальный узел обхода.

Так как вершины открываются в порядке возрастающего расстояния от корня, то это дерево обладает очень важным свойством. Однозначно определенный путь от корня до каждой вершины $x \in V$ использует наименьшее количество ребер (или, что эквивалентно, промежуточных вершин), возможное в любом маршруте графа от корня до вершины x.

Этот путь можно воссоздать, следуя по цепи предшественников от вершины x по направлению к корню. Обратите внимание, что в этом случае нам нужно двигаться в обратном направлении. Мы не можем найти путь от корня к вершине x, потому что указатели родителей имеют противоположное направление. Вместо этого путь нужно искать по направлению от вершины x к корню. Так как это направление противоположно тому, в котором требуется выполнять проход, мы можем либо сохранить путь, а потом явно обратить его с помощью стека, либо переложить эту работу на рекурсивную процедуру, показанную в листинге 5.13.

Листинг 5.13. Изменение направления пути посредством рекурсии

```
find_path(int start, int end, int parents[])
{
   if ((start == end) || (end == -1))
      printf("\n%d",start);
   else {
      find_path(start,parents[end],parents);
     printf(" %d",end);
   }
}
```

При обходе в ширину графа, изображенного на рис. 5.8, наш алгоритм выдал следующие отношения "вершина/родитель":

Вершина	1	2	3	4	5	6
Родитель	-1	1	2	5	1	1

Согласно этим родительским отношениям, самый короткий путь от вершины 1 к вершине 4 проходит через набор вершин {1,5,4}.

При использовании обхода в ширину для поиска кратчайшего пути от вершины x к вершине y нужно иметь в виду следующее: дерево кратчайшего пути полезно только в том случае, если корнем поиска в ширину является вершина x; кроме того, поиск в ширину дает самый короткий путь только для невзвешенных графов. Алгоритмы поиска кратчайшего пути во взвешенных графах рассматриваются в *разделе 6.3.1*.

5.7. Применение обхода в ширину

Большинство элементарных алгоритмов для работы с графами выполняют один или два обхода графа для получения всей информации о графе. Любой из таких алгоритмов, если он корректно реализован с использованием списков смежности, обязательно имеет линейное время исполнения, т. к. обход в ширину исполняется за время $O(n + m)$ как на ориентированных, так и на неориентированных графах. Это оптимальное время, т. к. именно за такое время можно *прочитать* граф из n вершин и m ребер.

Секрет мастерства заключается в умении видеть ситуации, в которых применение обхода гарантированно даст положительные результаты. Далее приводится несколько примеров использования обхода.

5.7.1. Компоненты связности

Граф называется *связным* (connected), если имеется путь между любыми двумя его вершинами. Гипотеза шести шагов утверждает, что любые два человека в мире связаны короткой цепочкой из людей, попарно знакомых друг с другом. Если теория шести шагов правильна, то граф дружеских отношений должен быть связным.

Компонентой связности (connected component) неориентированного графа называется максимальный набор его вершин, для которого существует путь между каждой парой

вершин. Эти компоненты являются отдельными "кусками" графа, которые не соединены между собой. В качестве примера отдельных компонент связности в графе дружеских отношений можно привести первобытные племена где-то в джунглях, которые еще не были открыты для остального мира. А отшельник в пустыне или крайне неприятный человек будет примером компоненты связности, состоящей из одной вершины.

Удивительно, какое большое количество кажущихся сложными проблем сводится к поиску или подсчету компонент связности. Например, вопрос, можно ли решить какую-то головоломку (скажем, кубик Рубика), начав с определенной позиции, по сути, представляет собой вопрос, является ли связным граф разрешенных конфигураций.

Компоненты связности можно найти с помощью обхода в ширину, т. к. порядок перечисления вершин не имеет значения. Начнем обход с первой вершины. Все элементы, обнаруженные в процессе этого обхода, должны быть членами одной и той же компоненты связности. Потом повторим обход, начиная с любой неоткрытой вершины (если таковая имеется), чтобы определить вторую компоненту связности, и т. д., до тех пор, пока не будут обнаружены все вершины. Соответствующая процедура приводится в листинге 5.14.

Листинг 5.14. Процедура поиска компонент связаности

```
connected_components(graph *g)
{
   int c;          /* Номер компоненты */
   int i;          /* Счетчик */

   initialize_search(g);

   c = 0;
   for (i=1; i<=g->nvertices; i++)
      if (discovered[i] == FALSE) {
         c = c+1;
         printf("Component %d:",c);
         bfs(g,i);
         printf("\n");
      }
}
process_vertex_early(int v)
{
   printf(" %d",v);
}
process_edge(int x, int y)
{
}
```

Обратите внимание на увеличение значения счетчика `c`, содержащего номер текущей компоненты, при каждом вызове функции `bfs`. Изменив соответствующим образом действие функции `process_vertex()`, каждую вершину можно было бы явно связать

с номером ее компоненты (вместо того, чтобы выводить на экран вершины каждой компоненты).

Для ориентированных графов существуют два понятия связности, и это определяет существование алгоритмов поиска компонент сильной и слабой связности. Любую из них можно найти за время $O(n + m)$, что показано в *разделе 15.1*.

5.7.2. Раскраска графов двумя цветами

В задаче *раскраски вершин* (vertex colouring) требуется присвоить метку (или цвет) каждой вершине графа таким образом, чтобы любые две соединенные ребром вершины были разного цвета. Присвоение каждой вершине своего цвета позволяет избежать любых конфликтов. Но при этом нужно использовать как можно меньшее количество цветов. Задачи раскраски вершин часто возникают в приложениях календарного планирования, например, при выделении регистров в компиляторах. Подробное обсуждение алгоритмов раскраски вершин можно найти в *разделе 16.7*.

Граф называется *двудольным* (bipartite), если его вершины можно правильно раскрасить двумя цветами. Важность двудольных графов заключается в том, что они возникают естественным образом во многих приложениях. Рассмотрим граф сексуальных связей среди гетеросексуалов. Мужчины здесь могут вступать в сексуальные связи только с женщинами и наоборот. Таким образом, в этой простой модели правильная двухцветная раскраска определяется полом.

Но как мы можем правильно раскрасить граф в два цвета, таким образом разделив мужчин и женщин? Предположим, что начальная вершина обхода представляет мужчину. Тогда все смежные вершины должны представлять женщин, при условии, что граф в действительности является двудольным.

Мы можем расширить алгоритм обхода в ширину таким образом, чтобы раскрашивать каждую новую открытую вершину цветом, противоположным цвету ее предшественника. Потом мы проверяем, не связывает какое-либо ребро, которое не задействовалось в процессе открытия вершин, две вершины одного цвета. Такая связь будет означать, что граф нельзя раскрасить в два цвета. Процедура двухцветной раскраски графа показана в листинге 5.15.

Листинг 5.15. Процедура двухцветной раскраски графа

```
twocolor(graph *g)
{
   int i;    /* счетчик */
   for (i=1; i<=(g->nvertices); i++)
      color[i] = UNCOLORED;
   bipartite = TRUE;
   initialize_search(&g);
   for (i=1; i<=(g->nvertices); i++)
      if (discovered[i] == FALSE) {
         color[i] = WHITE;
         bfs(g,i);
   }
}
```

```
process_edge(int x, int y)
{
   if (color[x] == color[y]) {
      bipartite = FALSE;
      printf("Warning: not bipartite due to (%d,%d)\n",x,y);
   }
   color[y] = complement(color[x]);
}
complement(int color)
{
   if (color == WHITE) return(BLACK);
   if (color == BLACK) return(WHITE);
   return(UNCOLORED);
}
```

Первой вершине в любой компоненте связности можно присвоить любой цвет (пол). Алгоритм обхода в ширину может разделить мужчин и женщин, но, исходя только из структуры графа, не может определить, какой цвет представляет какой пол.

> ***Подведение итогов***
>
> Обходы в ширину и в глубину предоставляют механизмы для посещения каждой вершины и каждого ребра графа. Они лежат в основе большинства простых, эффективных алгоритмов для работы с графами.

5.8. Обход в глубину

Существует два основных алгоритма обхода графов: *обход в ширину* (breadth-first search, BFS) и *обход в глубину* (depth-first search, DFS). Для некоторых задач нет абсолютно никакой разницы, какой тип обхода (поиска) использовать, но для других эта разница является критической.

Разница между поиском в ширину и поиском в глубину заключается в порядке исследования вершин. Этот порядок зависит полностью от структуры-контейнера, используемой для хранения *открытых*, но не *обработанных* вершин.

- *Очередь.* Помещая вершины в очередь типа FIFO, мы исследуем самые старые неисследованные вершины первыми. Таким образом, наше исследование медленно распространяется вширь, начиная от стартовой вершины. В этом суть обхода в ширину.
- *Стек.* Помещая вершины в стек с порядком извлечения LIFO, мы исследуем их, отклоняясь от пути для посещения очередного соседа, если таковой имеется, и возвращаясь назад, только если оказываемся в окружении ранее открытых вершин. Таким образом, мы в своем исследовании быстро удаляемся от стартовой вершины, и в этом заключается суть обхода в глубину.

Наша реализация процедуры обхода в глубину отслеживает время обхода для каждой вершины. Каждый вход в любую вершину и выход из нее считаются затратой времени. Для каждой вершины ведется учет затрат времени на вход и выход.

Процедуру обхода в глубину можно реализовать рекурсивным методом, что позволяет избежать явного использования стека. Псевдокод алгоритма обхода в глубину показан в листинге 5.16.

Листинг 5.16. Обход в глубину

```
DFS(G,u)
   state[u] = "discovered"
   обрабатываем вершину u, если необходимо
   entry[u] = time
   time = time + 1
   for each v ∈ Adj[u] do
      обрабатываем ребро (u, v), если необходимо
      if state[v] = "undiscovered" then
         p[v] = u
         DFS(G,v)
   state[u] = "processed"
   exit[u] = time
   time = time + 1
```

При обходе в глубину интервалы времени, потраченного на посещение вершин, обладают интересными свойствами. В частности:

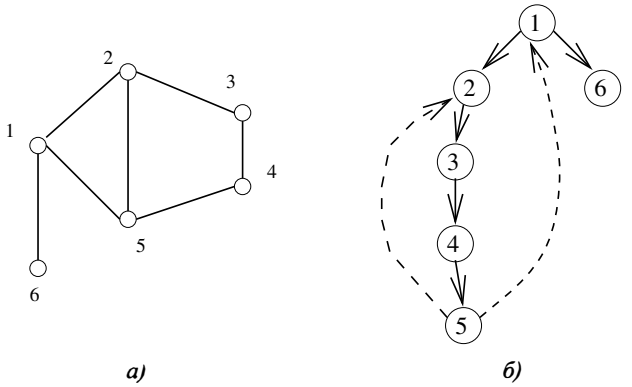
- *Посещение предшественника.* Допустим, что вершина x является предшественником вершины y в дереве обхода в глубину. Это подразумевает, что вершина x должна быть посещена раньше, чем вершина y, т. к. невозможно быть рожденным раньше своего отца или деда. Кроме этого, выйти из вершины x можно только после выхода из вершины y, т. к. механизм поиска в глубину предотвращает выход из вершины x до тех пор, пока мы не вышли из всех ее потомков. Таким образом, временной интервал посещения y должен быть корректно учтен его предшественником x.
- *Количество потомков.* Разница во времени выхода и входа для вершины v свидетельствует о количестве потомков этой вершины в дереве обхода в глубину. Показания часов увеличиваются на единицу при каждом входе и каждом выходе из вершины, поэтому количество потомков данной вершины будет равно половине разности между моментом выхода и моментом входа.

Мы будем использовать время входа и выхода в разных приложениях обхода в глубину, в частности, в топологической сортировке и в поиске компонент двусвязности (или компонент сильной связности). Нам нужно будет выполнять разные действия при каждом входе и выходе из вершины, для чего из процедуры `dfs` будут вызываться функции `process_vertex_early()` и `process_vertex_late()` соответственно.

Другим важным свойством обхода в глубину является то, что он разбивает ребра неориентированного графа на два класса: *древесные* (tree edges) и *обратные* (back edges). Древесные ребра используются при открытии новых вершин и закодированы в родительском отношении. Обратные ребра — это те ребра, у которых второй конец является предшественником расширяемой вершины, и поэтому они направлены обратно к дереву.

Удивительным свойством обхода в глубину является то, что все ребра попадают в одну из этих двух категорий. Почему ребро не может соединять одноуровневые узлы, а только родителя с потомком? Потому, что все вершины, достижимые из данной вершины *v*, уже исследованы ко времени окончания обхода, начатого из вершины *v*, и такая топология невозможна для неориентированных графов. Данная классификация ребер является принципиальной для алгоритмов, основанных на обходе в глубину.

На рис. 5.9 изображен неориентированный граф и дерево его обхода в глубину.



а) *б)*

Рис. 5.9. Неориентированный граф *(а)* и дерево его обхода в глубину *(б)*

Реализация

Обход в глубину можно рассматривать как обход в ширину с использованием стека вместо очереди. Достоинством реализации обхода в глубину посредством рекурсии (листинг 5.17) является то, что она позволяет обойтись без явного использования стека.

Листинг 5.17. Обход в глубину

```
dfs(graph *g, int v)
{
   edgenode *p;           /* Временный указатель */
   int y;                 /* Следующая вершина */
   if (finished) return; /* Завершение поиска */
   discovered[v] = TRUE;
   time = time + 1;
   entry_time[v] = time;
   process_vertex_early(v);
   p = g->edges[v];
   while (p != NULL) {
      y = p->y;
      if (discovered[y] == FALSE) {
         parent[y] = v;
         process_edge(v,y);
         dfs(g,y);
      }
```

```
        else if ((!processed[y]) || (g->directed))
        process_edge(v,y);
        if (finished) return;
        p = p->next;
    }
    process_vertex_late(v);
    time = time + 1;
    exit_time[v] = time;
    processed[v] = TRUE;
}
```

Обход в глубину, по сути, использует ту же самую идею, что и поиск с возвратом, который рассматривается в *разделе 7.1*. В обоих алгоритмах мы перебираем все возможности, продвигаясь вперед, когда это удается, и возвращаясь обратно, когда больше не осталось неисследованных элементов для дальнейшего продвижения. Оба алгоритма легче всего воспринимать как рекурсивные алгоритмы.

Подведение итогов

При обходе в глубину вершины упорядочиваются по времени входа/выхода, а ребра разбиваются на два типа — древесные и обратные. Именно такая организация и делает обход в глубину столь мощным алгоритмом.

5.9. Применение обхода в глубину

По сравнению с другими парадигмами разработки алгоритмов, обход в глубину не кажется чем-то страшным. Но он довольно сложен, вследствие чего для его корректной работы необходимо правильно реализовать все до мельчайших подробностей.

Правильность алгоритма обхода в глубину зависит от того, *когда* именно обрабатываются ребра и вершины. Вершину v можно обработать до обхода исходящих из нее ребер (процедура `process_vertex_early()`) или после этого (процедура `process_vertex_late()`). Иногда действия выполняются в обоих случаях, например, инициализация до обхода ребер посредством функции `process_vertex_early()` специфической структуры данных вершин, которая будет модифицирована операциями обработки ребер, а после обхода подвергнута анализу посредством функции `process_vertex_late()`.

В неориентированных графах каждое ребро (x, y) находится в списке смежности как для вершины x, так и для вершины y. Таким образом, для обработки каждого ребра (x, y) существуют два момента — при исследовании вершины x и вершины y. Ребра разбиваются на древесные и обратные при первом исследовании ребра. Момент, в который мы видим ребро в первый раз, естественно подходит для выполнения специфической обработки ребра. В тех случаях, когда мы встречаемся с ребром во второй раз, может возникнуть необходимость в каком-то другом действии.

Но если мы встречаем ребро (x, y), двигаясь из вершины x, как нам определить, не было ли это ребро уже пройдено из вершины y? На этот вопрос легко ответить, если вершина y неоткрытая: ребро (x, y) становится древесным ребром, следовательно, мы видим его в первый раз. Ответ также очевиден, если вершина y не была полностью обработана: поскольку мы рассматривали ребро при исследовании вершины y, это второе

посещение этого ребра. А если вершина y является предшественником вершины x и, вследствие этого, находится в открытом состоянии? Подумав, мы поймем, что это должен быть первый проход, если только вершина y не является непосредственным предшественником вершины x, т. е. ребро (y, x) является древесным. Это можно установить проверкой равенства `y == parent[x]`.

Каждый раз, когда я пробую реализовать алгоритм на основе обхода в глубину, я убеждаюсь, что этот тип алгоритмов достаточно сложен. Поэтому я рекомендую вам тщательно изучить рассмотренные реализации этого алгоритма, чтобы понять, где и почему могут возникать проблемы.

5.9.1. Поиск циклов

Наличие обратных ребер является ключевым фактором при поиске циклов в неориентированных графах. Если в графе нет обратных ребер, то все ребра являются древесными и дерево не содержит циклов. Но любое обратное ребро, идущее от вершины x к предшественнику y, создает цикл, или замкнутый маршрут между верши нами y и x. Цикл легко найти посредством обхода в глубину, как показано в листинге 5.18.

Листинг 5.18. Поиск цикла

```
process_edge(int x, int y)
{
   if (parent[x] != y) {     /* Найдено обратное ребро! */
      printf("Cycle from %d to %d:",y,x);
      find_path(y,x,parent);
      printf("\n\n");
      finished = TRUE;
   }
}
```

Правильность данного алгоритма поиска цикла зависит от обработки каждого неориентированного ребра только один раз. В противном случае двойной проход по любому неориентированному ребру может создать фиктивный двухвершинный цикл (x, y, x). Поэтому после обнаружения первого цикла устанавливается флаг `finished`, что ведет к завершению работы процедуры.

5.9.2. Шарниры графа

Допустим, вы — диверсант в тылу врага, которому нужно вывести из строя телефонную сеть противника. Какую из показанных на рис. 5.10 телефонных станций вы бы решили взорвать, чтобы причинить как можно больший ущерб?

Обратите внимание, что в графе на рис. 5.10 имеется одна точка, удаление которой отсоединяет связную компоненту графа. Такая вершина называется шарниром (или разделяющей вершиной, или точкой сочленения). Любой граф, содержащий шарнир, является уязвимым по своей природе, т. к. удаление этой одной вершины ведет к потере связности между другими вершинами.

В *разделе 5.7.1* мы рассмотрели алгоритм на основе поиска в ширину для определения связных компонент графа. В общем, *связностью* графа (graph connectivity) называется наименьший набор вершин, в результате удаления которых граф станет несвязным. Если граф имеет шарнир, то его связность равна единице. Более устойчивые графы, в которых шарниры отсутствуют, называются *двусвязными*. Связность также рассматривается в *разделе 15.8*.

Шарниры графа можно с легкостью найти простым перебором. Для этого мы временно удаляем вершину v, после чего выполняем обход оставшегося графа в ширину или в глубину, чтобы выяснить, является ли он после этого связным. Эта процедура повторяется для каждой вершины графа. Общее время для n таких обходов равно $O(n(m + n))$. Но существует изящный алгоритм с линейным временем исполнения, который проверяет все вершины связного графа за один обход в глубину.

Какую информацию о шарнирах может предоставить нам дерево обхода в глубину? Прежде всего, это дерево содержит все вершины графа. Если дерево обхода в глубину представляет весь граф, то все внутренние (не листовые) вершины будут шарнирами, т. к. удаление любой из них отделит лист от корня. А удаление листа разъединить дерево не может, т. к. листовая вершина не соединяет с деревом никакие другие вершины. На рис. 5.11 изображено дерево обхода графа в глубину, содержащее два шарнира — вершины 1 и 2. Обратное ребро (5,2) не дает вершинам 3 и 4 быть шарнирами. Вершины 5 и 6 не являются шарнирами, потому что они листья.

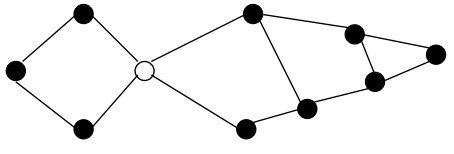


Рис. 5.10. Шарнир является самой слабой точкой графа

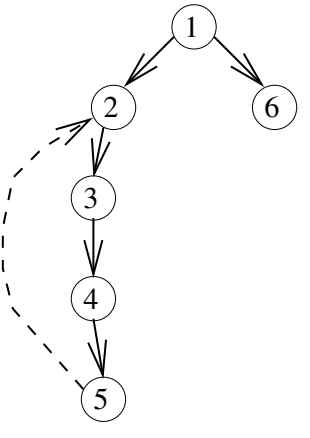


Рис. 5.11. Дерево обхода графа в глубину

Корень дерева обхода представляет специальный случай. Если он имеет только одного потомка, то корень также является листом. Но если корень имеет больше одного потомка, то удаление корня разделяет потомков, что означает, что корень является шарниром.

Общие графы более сложные, чем деревья. Но обход общего графа в глубину разделяет ребра на древесные и обратные. Обратные ребра можно рассматривать как страховочные фалы, связывающие вершину с одним из ее предшественников. Например, наличие обратного ребра от вершины x к вершине y гарантирует, что ни одна из вершин на пути между x и y не может быть шарниром. Удаление любой из этих вершин не нару-

шит связности дерева, т. к. обратное ребро, как страховочный фал, будет удерживать их связанными с остальными компонентами дерева.

При поиске шарниров графа требуется информация о том, в какой степени обратные ребра связывают части дерева обхода в глубину после возможного шарнира с предшествующими узлами. Пусть переменная `reachable_ancestor[v]` обозначает самый старший предшественник вершины *v*, достижимый через древесные и обратные. ребра. Сначала значение этой переменной равно `reachable_ancestor[v]=v` (листинг 5.19).

Листинг 5.19. Инициализация массива достижимых предшественников

```
int reachable_ancestor[MAXV+1];  /* Самый старший достижимый
                                 предшественник вершины v */
int tree_out_degree[MAXV+1];     /* Степень исхода вершины v
                                   дерева обхода в глубину*/

process_vertex_early(int v)
{
   reachable_ancestor[v] = v;
}
```

Переменная `reachable_ancestor[v]` обновляется при каждом обнаружении обратного ребра, которое ведет к более старшему предшественнику, чем текущий. Относительный возраст (ранг) предшественников можно определить по значению переменной `entry_time`, содержащей время входа в вершину (листинг 5.20).

Листинг 5.20. Определение возраста предшественников

```
process_edge(int x, int y)
{
  int class;       /* класс ребра */
  class = edge_classification(x,y);
  if (class == TREE)
      tree_out_degree[x] = tree_out_degree[x] + 1;
  if ((class == BACK) && (parent[x] != y)) {
     if (entry_time[y] < entry_time[reachable_ancestor[x]])
       reachable_ancestor[x]   y;
  }
}
```

Здесь исключительно важно определить, как достижимость влияет на то, является ли вершина *v* шарниром. Существует три типа шарниров, как показано на рис. 5.12.

- *Корневые шарниры.* Если корень дерева обхода в глубину имеет свыше одного потомка, то он должен быть шарниром, так как при его удалении никакое ребро не может соединить поддерево второго потомка с поддеревом первого потомка.
- *Мостовые шарниры.* Если самой первой достижимой из вершины *v* является вершина *родитель*[*v*], то удаление ребра (*родитель*[*v*], *v*) разрывает граф. Очевидно, что вершина (*родитель*[*v*]) должна быть шарниром, т. к. ее удаление отсоединяет вершину *v* от остального дерева обхода графа в глубину. По этой же причине шар-

ниром является и вершина *v*, если только она не является листом дерева обхода в глубину. Удаление любого листа удаляет только сам лист, но ничего после него.

- *Родительские шарниры.* Если самой первой вершиной, достижимой из вершины *v*, является родитель вершины *v*, то удаление этого родителя должно отрезать вершину *v* от остальной части дерева, если только данный родитель не является корнем дерева.

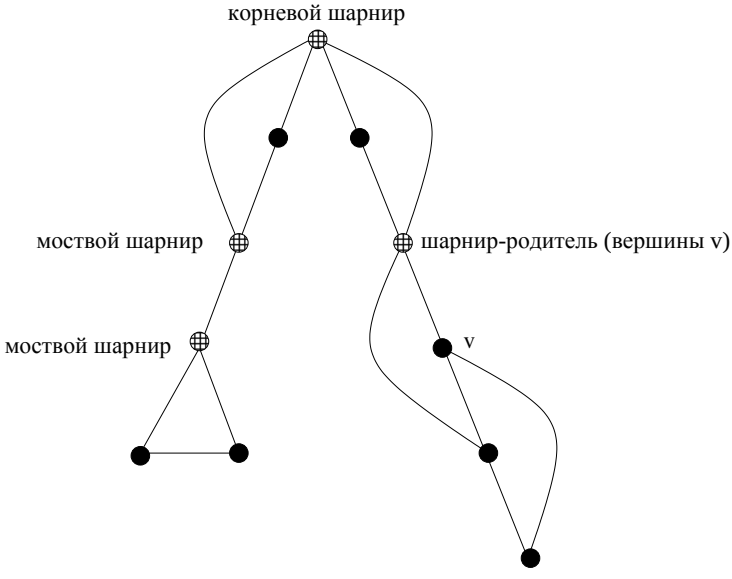


Рис. 5.12. Типы шарниров

В листинге 5.21 приводится процедура для определения соответствия вершины, из которой выполняется возврат после обхода всех ее исходящих ребер, одному из этих трех типов шарниров.

"Возраст" вершины *v* представляется переменной `entry_time[v]`. Вычисляемое время `time_v` обозначает самую старшую вершину, достигаемую посредством обратных ребер. Возвращение к предшественнику выше вершины *v* исключает возможность, что вершина *v* может быть шарниром.

Листинг 5.21. Определение типа шарнира

```
process_vertex_late(int v)
{
   bool root;        /* Эта вершина — корень дерева обхода в глубину? */
   int time_v;       /* Самое раннее время достижимости вершины v */
   int time_parent; /* Самое раннее время достижимости
                       вершины parent[v] */
   if (parent[v] < 1) { /* Вершина v — корень? */
      if (tree_out_degree[v] > 1)
         printf("root articulation vertex: %d \n",v);
      return;
   }
```

```
   root = (parent[parent[v]] < 1);   /* Вершина parent[v] — корень? */
   if ((reachable_ancestor[v] == parent[v]) && (!root))
      printf("parent articulation vertex: %d \n",parent[v]);
   if (reachable_ancestor[v] == v) {
      printf("bridge articulation vertex: %d \n",parent[v]);
      if (tree_out_degree[v] > 0)    /* Вершина v — лист? */
         printf("bridge articulation vertex: %d \n",v);
   }
   time_v = entry_time[reachable_ancestor[v]];
   time_parent = entry_time[ reachable_ancestor[parent[v]] ];
   if (time_v < time_parent)
      reachable_ancestor[parent[v]] = reachable_ancestor[v];
}
```

В последних строчках кода данной процедуры определяется момент, в который мы превращаем самый старый достижимый предшественник вершины в ее родителя, а именно тот момент, когда он выше, чем текущий предшественник родителя.

В качестве альтернативы, надежность можно рассматривать не с точки зрения слабости вершин, а с точки зрения слабости ребер. Возможно, нашему разведчику вместо вывода из строя телефонной станции было бы легче повредить кабель. Ребро, чье удаление разъединяет граф, называет *мостом* (bridge). Граф, не имеющий мостов, называется *реберно-двусвязным* (edge-biconnected).

Является ли мостом данное ребро (x, y), можно с легкостью определить за линейное время, удалив это ребро и проверив, является ли получившийся граф связным. Более того, можно найти все мосты за такое же линейное время $O(n + m)$. Ребро (x, y) является мостом, если, во-первых, оно древесное, а во-вторых, нет обратных ребер, которые соединяли бы вершину y или нижележащие вершины с вершиной x или вышележащими вершинами. Соответствие ребра этим условиям можно проверить с помощью слегка модифицированной функции для определения достижимого предшественника (см. листинг 5.21).

5.10. Обход в глубину ориентированных графов

Обход в глубину неориентированного графа полезен тем, что он аккуратно упорядочивает ребра графа, как показано на рис. 5.9.

При обходе неориентированных графов каждое ребро или находится в дереве обхода в глубину, или является обратным ребром к предшествующему ребру в дереве. Давайте повторно рассмотрим, почему это так. Допустим, что при обходе мы выходим на прямое ребро (x, y), направленное к вершине-потомку. В таком случае мы бы открыли ребро (x, y) при исследовании вершины y, что делает его обратным ребром. Или допустим, что мы выходим на поперечное ребро (x, y), связывающее две вершины, не имеющие отношения друг к другу. Опять же, мы открыли бы это ребро при исследовании вершины y, что делает его древесным ребром.

Для ориентированных графов диапазон допустимых маркировок обхода в глубину может быть более широким. В самом деле, при обходе ориентированных графов могут использоваться все четыре типа ребер, показанные на рис. 5.13.

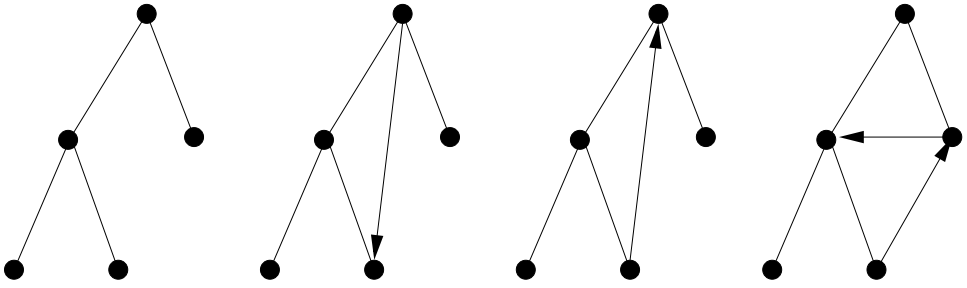


Рис. 5.13. Возможные типы ребер

Но эта классификация оказывается полезной при разработке алгоритмов для работы с ориентированными графами. Для каждого типа ребра обычно предпринимается соответствующее ему действие.

Тип ребра можно без труда определить, исходя из состояния вершины, времени ее открытия и ее родителя. Соответствующая процедура показана в листинге 5.22.

Листинг 5.22. Определение типа ребра

```
int edge_classification(int x, int y)
{
   if (parent[y] == x) return(TREE);
   if (discovered[y] && !processed[y]) return(BACK);
   if (processed[y] && (entry_time[y]>entry_time[x])) return(FORWARD);
   if (processed[y] && (entry_time[y]<entry_time[x])) return(CROSS);
   printf("Warning: unclassified edge (%d,%d)\n",x,y);
   }
```

Как и в случае с процедурой обхода в ширину, реализация алгоритма обхода в глубину содержит места, в которые можно вставить функцию выборочной обработки каждой вершины или ребра, например, копировать, выводить на экран или подсчитывать их. Оба алгоритма начинают с обхода всех ребер в одной компоненте связности. Так как для обхода несвязного графа нам нужна начальная вершина в каждой компоненте, то обход должен начинаться с любой вершины, оставшейся неоткрытой после обхода компоненты. При условии правильной инициализации получаем законченный алгоритм обхода ориентированного графа в глубину (листинг 5.23).

Листинг 5.23. Алгоритм обхода в глубину ориентированного графа

```
DFS-graph(G)
   for each vertex u ∈ V[G]do
      state[u] = "undiscovered"
         for each vertex u ∈ V[G] do
      if state[u] = "undiscovered" then
         инициализируем новую компоненту, если необходимо
         DFS(G,u)
```

Я рекомендую читателям проверить и убедиться в правильности этих четырех состояний вершины. Все, сказанное мною о сложности алгоритма обхода в глубину, вдвойне справедливо для орграфов.

5.10.1. Топологическая сортировка

Топологическая сортировка является наиболее важной операцией на бесконтурных орграфах. Данный тип сортировки упорядочивает вершины вдоль линии таким образом, что все ориентированные ребра направлены слева направо. Такое упорядочивание ребер невозможно в графе, содержащем ориентированный цикл, т. к. невозможно, чтобы ребра шли по одной линии в одну сторону и при этом возвращались назад в исходную точку.

Любой бесконтурный орграф имеет, по крайней мере, одно топологическое упорядочивание (рис. 5.14).

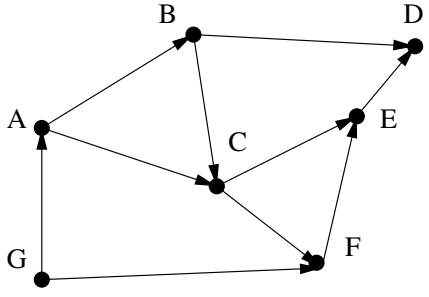


Рис. 5.14. Бесконтурный орграф
с единственным топологическим упорядочиванием (G, A, B, C, F, E, D)

Важность топологической сортировки состоит в том, что она позволяет упорядочить вершины графа таким образом, что каждую вершину можно обработать перед обработкой ее потомков. Допустим, что ребра представляют управление очередностью таким образом, что ребро (x, y) означает, что работу x нужно выполнить раньше, чем работу y. Тогда любое топологическое упорядочивание определяет правильное календарное расписание. Более того, бесконтурный орграф может содержать несколько таких упорядочиваний.

Но топологическая сортировка имеет и другие применения. Например, предположим, что нам нужно найти самый короткий (или самый длинный) путь в бесконтурном орграфе от точки x до точки y. Никакая вершина, расположенная в топологическом упорядочивании после вершины y, не может находиться в этом пути, т. к. из такой вершины невозможно попасть обратно в вершину y. Все вершины можно обработать слева направо в топологическом порядке, принимая во внимание влияние их исходящих ребер, и будучи в полной уверенности, что мы просмотрим все, что нужно, перед тем как оно потребуется. Топологическая сортировка оказывается очень полезной для решения практически любых алгоритмических задач по орграфам, что подробно обсуждается в *разделе 15.2*.

Обход в глубину является эффективным средством для осуществления топологической сортировки. Орграф является бесконтурным, если не содержит обратных ребер. Топологическое упорядочивание бесконтурного орграфа осуществляется посредством мар-

кировки вершин в порядке, обратном тому, в котором они отмечаются как *обработанные*. Почему это возможно? Рассмотрим, что происходит с каждым ориентированным ребром (x, y), обнаруженным при исследовании вершины x.

- Если вершина y *не открыта*, то мы начинаем обход в глубину из вершины y, прежде чем мы можем продолжать исследование вершины x. Таким образом, вершина y помечается как *обработанная* до того, как такой статус присваивается вершине x, и в топологическом упорядочивании вершина x будет находиться перед вершиной y, что и требуется.
- Если вершина y *открыта*, но не *обработана*, то ребро (x, y) является обратным ребром, что запрещено в бесконтурном орграфе.
- Если вершина y *обработана*, то она помечается соответствующим образом раньше вершины x. Следовательно, в топологическом упорядочивании вершина x будет находиться перед вершиной y; что и требуется.

В листинге 5.24 приводится реализация топологической сортировки.

Листинг 5.24. Топологическая сортировка

```
process vertex_late(int v)
{
   push(&sorted,v);
}
process_edge(int x, int y)
{
   int class; /* Класс ребра */
   class = edge_classification(x,y);
   if (class == BACK)
      printf("Warning: directed cycle found, not a DAG\n");
}
topsort(graph *g)
{
   int i; /* Счетчик */
   init_stack(&sorted);
   for (i=1; i<=g->nvertices; i++)
      if (discovered[i] == FALSE)
         dfs(g,i);
   print_stack(&sorted);  /* Выводим топологическое упорядочивание */
}
```

Каждая вершина кладется в стек после обработки всех исходящих ребер. Самая верхняя вершина в стеке не имеет входящих ребер, идущих от какой-либо вершины, имеющейся в стеке. Последовательно снимая вершины со стека, получаем их топологическое упорядочивание.

5.10.2. Сильно связные компоненты

Классическим применением поиска в глубину является разложение ориентированного графа на *сильно связные компоненты* (strongly connected components). Орграф является

сильно связным (strongly connected), если для любой пары его вершин (v, u) вершина v достижима из вершины u и наоборот. В качестве практического примера сильно связного графа можно привести граф дорожных сетей с двусторонним движением.

Является ли граф $G = (V, E)$ сильно связным, можно с легкостью проверить посредством обхода графа за линейное время. Начнем обход с произвольной вершины v. Каждая вершина графа должна быть достижимой из этой вершины и, следовательно, открыта обходом в ширину или глубину с начальной точкой в вершине v. В противном случае граф G не может быть сильно связным. Потом создадим граф $G' = (V, E')$ с точно таким же набором вершин и ребер, как и граф G, но с обратной ориентацией ребер, т. е., ориентированное ребро $(x, y) \in E$ тогда и только тогда, когда $(y, x) \in E'$. Таким образом, любой путь от вершины v к вершине z в графе G' соответствует пути от вершины z к вершине v в графе G. Выполнив обход в глубину графа G' от вершины v, мы найдем все вершины с путями к этой вершине в графе G. Граф является сильно связным тогда и только тогда, когда вершина v достижима из любой вершины в графе G и все вершины в графе G достижимы из вершины v.

Графы, которые сами не являются сильно связными, можно разбить на сильно связные компоненты, как показано на рис. 5.15, *а*.

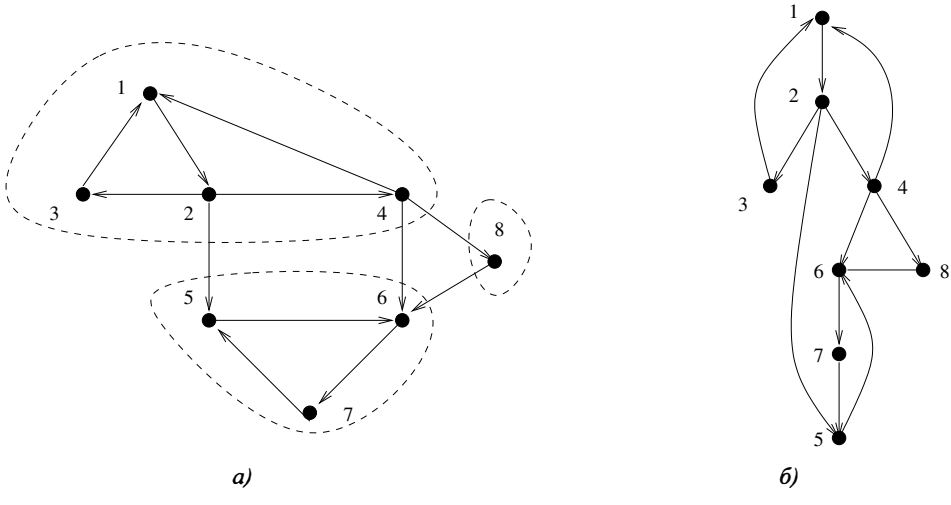


Рис. 5.15. Сильно связные компоненты графа *(а)* и дерево обхода в глубину *(б)*

Сильно связные компоненты графа и соединяющие их слабо связные ребра можно найти с помощью обхода в глубину. Этот алгоритм основан на том обстоятельстве, что с помощью обхода в глубину легко найти ориентированный цикл, поскольку такой цикл дает любое обратное ребро и путь вниз в дереве обхода в глубину. Все вершины в этом цикле должны быть в одной и той же сильно связной компоненте. Таким образом мы можем сжать вершины в этом цикле в одну вершину, представляющую всю компоненту, после чего повторить обход. Этот процесс прекращается, когда исчерпаны все ориентированные циклы, а каждая вершина представляет отдельную сильно связную компоненту.

Наш подход к реализации этой идеи похож на поиск компонент двусвязности в *разделе 5.9.2*. Однако мы пересмотрим понятие самой старшей достижимой вершины в связи с наличием "недревесных" ребер и необходимостью возвращения из вершин. Так как мы имеем дело с ориентированным графом, то нам нужно принимать во внимание прямые ребра (ведущие от вершины к потомку) и поперечные ребра (ведущие от вершины обратно к ранее открытой вершине, не являющейся предшественником). Наш алгоритм (листинг 5.25) отделяет от дерева обхода по одной сильной компоненте за шаг и присваивает всем ее вершинам номер данной компоненты.

Листинг 5.25. Алгоритм разложения графа на сильно связные компоненты

```
strong_components(graph *g)
{
   int i;     /* Счетчик */
   for (i=1; i<=(g->nvertices); i++) {
      low[i] = i;
      scc[i] = -1;
   }
   components_found = 0;
   init_stack(&active);
   initialize_search(&g);
   for (i=1; i<=(g->nvertices); i++)
      if (discovered[i] == FALSE) {
         dfs(g,i);
      }
}
```

Переменная `low[v]` представляет самую старшую известную вершину, находящуюся в той же самой сильно связной компоненте, что и вершина *v*. Эта вершина не обязательно является родителем вершины *v*, но благодаря поперечным ребрам может находиться на одном уровне с ней. Поперечные ребра, которые идут к вершинам из *предыдущих* сильно связных компонент графа, нам не нужны, т. к. от них нет пути назад к вершине *v*, но в других случаях поперечные ребра подлежат обработке. Прямые ребра не влияют на достижимость по ребрам дерева обхода в глубину и поэтому их можно не принимать во внимание (листинг 5.26).

Листинг 5.26. Процедура обработки ребер

```
int low[MAXV+1];  /* Самая старшая вершина наверняка находится в
                                        компоненте вершины v */
int scc[MAXV+1];  /* Номер сильной компоненты для каждой вершины */
process_edge(int x, int y)
{
  int class; /* Класс ребра */
  class = edge_classification(x,y);
  if (class == BACK) {
    if (entry_time[y] < entry_time[ low[x] ] )
      low[x] = y;
  }
```

```
  if (class == CROSS) {
    if (scc[y] == -1)  /* Компонента еще не присвоена */
      if (entry_time[y] < entry_time[ low[x] ] )
        low[x] = y;
  }
}
```

Новая сильно связная компонента считается обнаруженной, когда самой нижней вершиной, достижимой из вершины *v*, является эта же вершина *v*. В таком случае эта компонента снимается со стека. В противном случае, мы объявляем родителя самым старшим достижимым предшественником и выполняется возврат (листинг 5.27).

Листинг 5.27. Процедура обработки вершин

```
process_vertex_early(int v)
{
   push(&active,v);
}
process_vertex_late(int v)
{
   if (low[v] == v) {   /* Ребро (parent[v],v) отрезает
                           сильно связную компоненту */
      pop_component(v);
   }
   if (entry_time[low[v]] < entry_time[low[parent[v]]])
      low[parent[v]] = low[v];
}
pop_component(int v)
{
   int t;                  /* Заполнитель вершины */
   components_found = components_found + 1;
   scc[v] = components_found;
   while ((t = pop(&active)) != v)
      scc[t] = components_found;
   }
}
```

Замечания к главе

Рассмотренный в главе материал по обходу графов является расширенной версией материала из главы 9 книги [SR03]. Самое лучшее описание библиотеки для работы с графами Combinatorica можно найти в старом [Ski90] и новом [PS03] издании книги, посвященной работе с этой библиотекой. Доступное введение в теорию социальных сетей можно найти в книгах [Bar03] и [Wat04].

5.11. Упражнения

Алгоритмы для эмуляции графов

1. [3] Даны графы G_1 и G_2 (рис. 5.16).

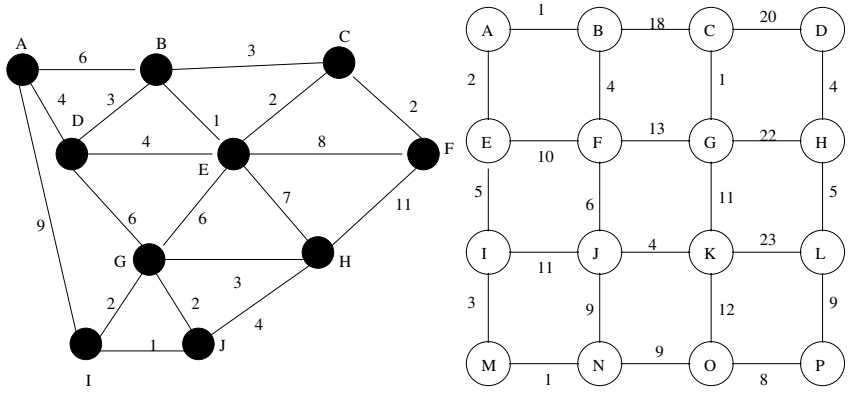


Рис. 5.16. Примеры графов

а) Укажите порядок вершин при обходе в ширину, начинающемся с вершины *A*. В случае конфликтов рассматривайте вершины в алфавитном порядке (т. е. *A* предшествует *Z*).

б) Укажите порядок вершин в обходе в глубину, начинающемся с вершины *A*. В случае конфликтов рассматривайте вершины в алфавитном порядке (т. е. *A* предшествует *Z*).

2. [3] Выполните топологическую сортировку на графе *G*, представленном на рис. 5.17.

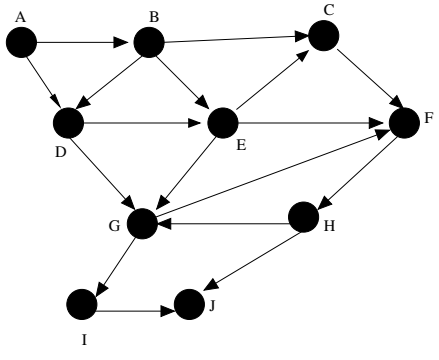


Рис. 5.17. Пример графа

Обход графов

3. [3] Докажите методом индукции, что между любой парой вершин дерева существует однозначный путь.

4. [3] Докажите, что при обходе в ширину неориентированного графа *G* каждое ребро может быть либо древесным, либо поперечным, где *x* не является ни предшественником, ни потомком вершины *y* в поперечном ребре (*x*, *y*).

5. [3] Разработайте алгоритм с линейным временем исполнения для вычисления хроматического числа графов, где степень каждой вершины не выше второй. Должны такие графы быть двудольными?

6. [5] В обходах в ширину и глубину неоткрытая вершина помечается как *открытая* при первом ее посещении и как *обработанная* после того, как она была полностью исследована. В любой момент времени в открытом состоянии могут находиться несколько вершин одновременно.

   а) Опишите граф из n вершин с заданной вершиной v, у которого при обходе в ширину, начинающемся в вершине v, одновременно $\Theta(n)$ вершин находятся в *открытом* состоянии.

   б) Опишите граф из n вершин с заданной вершиной v, у которого при обходе в глубину, начинающемся в вершине v, одновременно $\Theta(n)$ вершин находятся в *открытом* состоянии.

   в) Опишите граф из n вершин с заданной вершиной v, у которого в некоторой точке обхода в глубину, начинающегося в вершине v, одновременно $\Theta(n)$ вершин находятся в *неоткрытом* состоянии, в то время как $\Theta(n)$ вершин находятся в *обработанном* состоянии. (Следует иметь в виду, что при этом могут также существовать *открытые* вершины.)

7. [4] Возможно ли восстановить двоичное дерево, зная его обход в ширину и симметричный обход? Если возможно, то опишите алгоритм для этого. Если невозможно, то предоставьте контрпример. Решите задачу для случаев обхода в ширину и глубину.

8. [3] Предоставьте правильный эффективный алгоритм для преобразования одной структуры данных неориентированного графа G в другую. Для каждого алгоритма предоставьте значение временной сложности, полагая, что граф состоит из n вершин и m ребер.

   а) Преобразовать матрицу смежности в списки смежности.

   б) Преобразовать список смежности в матрицу инцидентности. Матрица инцидентности M содержит строку для каждой вершины и столбец для каждого ребра. Если вершина i является частью ребра j, то $M[i, j] = 1$, в противном случае $M[i, j] = 0$.

   в) Преобразовать матрицу инцидентности в списки смежности.

9. [3] Дано арифметическое выражение в виде дерева. Каждый лист дерева является целом числом, а каждый внутренний узел представляет одну из стандартных арифметических операций (+,–,*,/). Например, представление выражения 2 + 3*4 + (3*4)/5 в виде дерева показано на рис. 5.18, *а*. Разработайте алгоритм с временем исполнения $O(n)$ для вычисления такого выражения, где n — количество узлов в дереве.

10. [5] Арифметическое выражение представлено в виде бесконтурного орграфа, из которого удалены общие вложенные выражения. Каждый лист является целым числом, а каждый внутренний узел представляет одну из стандартных арифметических операций (+,–,*,/). Например, представление выражения 2 + 3*4 + (3*4)/5 в виде бесконтурного орграфа показано на рис. 5.18, *б*. Разработайте алгоритм для вычисления такого выражения за время $O(n + m)$, где n — количество вершин бесконтурного орграфа, а m — количество ребер. (Подсказка: для достижения требуемой эффективности модифицируйте алгоритм для случая с деревом.)

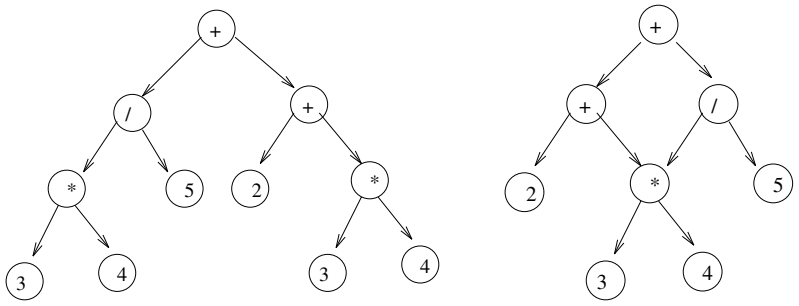


Рис. 5.18. Представление выражения 2 + 3*4 + (3*4)/5 в виде дерева *(а)* и бесконтурного орграфа *(б)*

11. [8] В *разделе 5.4* описывается алгоритм для эффективного создания двойственного графа триангуляции, который не гарантирует линейного времени исполнения. Для данной задачи разработайте алгоритм с линейным временем исполнения в наихудшем случае.

Разработка алгоритмов

12. [5] *Квадратом* (square) орграфа $G = (V, E)$ называется граф $G^2 = (V, E^2)$, при условии, что $(u, w) \in E^2$ тогда и только тогда, когда существует такая вершина $v \in V$, для которой $(u, v) \in E$ и $(v, w) \in E$, т. е. от вершины u к вершине w существует путь, состоящий ровно из двух ребер. Предоставьте эффективные алгоритмы для создания списков и матриц смежности для таких графов.

13. [5] *Вершинным покрытием* (vertex cover) графа $G = (V, E)$ называется такое подмножество вершин $V' \in V$, для которого каждое ребро в E инцидентно, по крайней мере, одной вершине в V'.

    а) Разработайте эффективный алгоритм поиска вершинного покрытия минимального размера, если G является деревом.

    б) Пусть $G = (V, E)$ будет такой граф, в котором вес каждой вершины равен степени ее ребра. Разработайте эффективный алгоритм поиска вершинного покрытия графа минимального веса.

    в) Пусть $G = (V, E)$ будет деревом с вершинами с произвольным весом. Разработайте эффективный алгоритм поиска вершинного покрытия графа минимального веса.

14. [3] *Вершинным покрытием* графа $G = (V, E)$ называется такое подмножество вершин $V' \in V$, в котором каждое ребро в E содержит, по крайней мере, одну вершину из V'. Если мы удалим все листья из любого дерева обхода в глубину графа G, то будут ли оставшиеся вершины составлять вершинное покрытие графа G? Если да, предоставьте доказательство; если нет, приведите контрпример.

15. [5] *Вершинным покрытием* графа $G = (V, E)$ называется такое подмножество вершин $V' \in V$, в котором каждое ребро в E содержит, по крайней мере, одну вершину из V'. *Независимым множеством* (independent set) графа $G = (V, E)$ называется подмножество вершин $V' \in V$, такое, что E не содержит ни одного ребра, обе вершины которого являлись бы членами V'.

Независимым вершинным покрытием (independent vertex cover) называется такое подмножество вершин, которое является как независимым множеством, так и вершинным покрытием графа G. Разработайте эффективный алгоритм для проверки, содержит ли граф G независимое вершинное покрытие. К какой классической задаче на графах сводится данная задача?

16. [5] *Независимым множеством* (independent set) неориентированного графа $G = (V, E)$ называется такое подмножество вершин U, для которого E не содержит ни одного ребра, обе вершины которого являлись бы членами U.

    а) Разработайте эффективный алгоритм поиска независимого множества максимального размера, если G является деревом.

    б) Пусть $G = (V, E)$ будет деревом, каждая вершина которого имеет вес, равный степени данной вершины. Разработайте эффективный алгоритм поиска независимого множества максимального размера дерева G.

    в) Пусть $G = (V, E)$ будет деревом с вершинами с произвольно присвоенным весом. Разработайте эффективный алгоритм поиска независимого множества максимального размера дерева G.

17. [5] Нужно определить, содержит ли данный неориентированный граф $G = (V, E)$ треугольник, т. е. цикл длиной 3.

    а) Разработайте алгоритм с временем исполнения $O(|V|^3)$ поиска треугольника в графе, если таковой имеется.

    б) Усовершенствуйте ваш алгоритм для исполнения за время $O(|V|\cdot|E|)$. Можно полагать, что $|V| \le |E|$.

    Обратите внимание на то, что эти пределы позволяют оценить время для преобразования представления графа G из матрицы смежности в список смежности (или в противоположном направлении).

18. [5] Имеется набор фильмов M_1, M_2, ..., M_k и набор клиентов, каждый из которых указывает два фильма, которые он бы хотел посмотреть в ближайшие выходные. Фильмы показываются по вечерам в субботу и воскресенье. Одновременно могут демонстрироваться несколько фильмов.

    Вам нужно решить, какие фильмы следует показывать в субботу, а какие в воскресенье, таким образом, чтобы каждый клиент смог посмотреть указанные им фильмы. Возможно ли составить расписание, по какому каждый фильм показывается только один раз? Разработайте эффективный алгоритм поиска такого расписания.

19. [5] Диаметр дерева $T = (V, E)$ задается формулой: $\max_{u,v \in V} \delta(u, v)$, где $\delta(u, v)$ — количество ребер в пути от вершины u к вершине v. Опишите эффективный алгоритм для вычисления диаметра дерева, докажите его правильность и проанализируйте время его исполнения.

20. [5] Имеется неориентированный граф G из n вершин и m ребер и целое число k. Разработайте алгоритм с временем исполнения $O(m + n)$ для поиска максимального *порожденного подграфа* (induced subgraph) H графа G, каждая вершина которого имеет степень $\ge k$, или докажите, что такого графа не существует. Порожденным подграфом $F = (U, R)$ графа $G = (V, E)$ называется подмножество U вершин V графа G и все ребра R графа G, для которых обе вершины каждого ребра являются членами U.

21. [6] Даны две вершины v и w в орграфе $G = (V, E)$. Разработайте алгоритм с линейным временем исполнения для поиска разных кратчайших путей (не обязательно не имеющих общих вершин) между v и w. Примечание: все ребра графа G невзвешенные.

22. [6] Разработайте алгоритм с линейным временем исполнения для удаления всех вершин второй степени из графа путем замены ребер (u, v) и (v, w) ребром (u, w). Также нужно удалить множественные копии ребер, оставив только одно ребро. Обратите внимание, что удаление копий ребра может создать новую вершину второй степени, которую нужно будет удалить, а удаление вершины второй степени может создать кратные ребра, которые также нужно будет удалить.

Ориентированные графы

23. [5] Перед вами стоит задача выстроить n непослушных детей друг за другом, причем у вас имеется список из m утверждений типа "i ненавидит j". Если i ненавидит j, то будет разумно не ставить i позади j, т. к. i может что-нибудь швырнуть в j.

    а) Разработайте алгоритм с временем исполнения $O(m + n)$ для выстраивания детей с соблюдением перечисленных в списке условий. Если такое упорядочивание невозможно, алгоритм должен сообщить об этом.

    б) Допустим, что вам нужно выстроить детей в несколько рядов таким образом, что если i ненавидит j, то i должен стоять в каком-нибудь ряду впереди j. Разработайте алгоритм для определения минимального количества рядов, если это возможно.

24. [3] Определите максимальное количество компонент, на которое можно уменьшить количество слабо связных компонент ориентированного графа, добавив в него одно ориентированное ребро. Что можно сказать по поводу количества сильно связных компонент?

25. [5] *Ориентированным деревом* (arborescence) орграфа G называется такое корневое дерево, что существует ориентированный путь от корня ко всем другим вершинам графа. Разработайте правильный эффективный алгоритм для тестирования, содержит ли граф G ориентированное дерево. Укажите временную сложность алгоритма.

26. [5] *Истоком* орграфа $G = (V, E)$ называется такая вершина v, из которой все другие вершины графа G достижимы ориентированным путем.

    а) Разработайте алгоритм с временем исполнения $O(n + m)$ (где $n = |V|$ и $m = |E|$) для проверки того, является ли данная вершина v истоком для графа G.

    б) Разработайте алгоритм с временем исполнения $O(n + m)$ (где $n = |V|$ и $m = |E|$) для проверки того, содержит ли граф G исток.

27. [9] *Турниром* (tournament) называется полный орграф, т. е. такой граф $G = (V, E)$, в котором для всех $u, v \in V$ только или (u, v) или (v, u) принадлежит множеству E. Покажите, что каждый турнир содержит Гамильтонов путь, т. е. путь, который проходит через каждую вершину только один раз. Разработайте алгоритм поиска этого пути.

Шарниры графа

28. [5] Шарниром графа G называется вершина, удаление которой разъединяет граф. Дан граф G из n вершин и m ребер. Разработайте алгоритм с временем исполнения $O(n + m)$ для поиска вершины в графе G, не являющейся шарниром, т. е. вершины, чье удаление не разъединяет граф.

29. [5] По условиям предыдущей задачи разработайте алгоритм с временем исполнения $O(n + m)$ для поиска такой последовательности удалений n вершин, в которой ни одно удаление не разъединяет граф. (Подсказка: рассмотрите возможность обхода в глубину и в ширину.)

30. [3] Дан связный неориентированный граф G. Ребро e, удаление которого разъединяет граф, называется *мостом*. Должен ли каждый мост e быть ребром в дереве обхода в глубину графа G? Если да, то предоставьте доказательство этого; если нет, то предоставьте соответствующий контрпример.

Задачи, предлагаемые на собеседовании

31. [5] Какие структуры данных используются при обходе в глубину и при обходе в ширину?

32. [4] Напишите функцию, которая обходит дерево двоичного поиска и возвращает i-й по порядку узел.

Задачи по программированию

Эти задачи доступны на сайтах **http://www.programming-challenges.com** и **http://uva.onlinejudge.org**.

1. Bicoloring. 110901/10004.
2. Playing with Wheels. 110902/10067.
3. The Tourist Guide. 110903/10099.
4. Edit Step Ladders. 110905/10029.
5. Tower of Cubes. 110906/10051.

Алгоритмы для работы со взвешенными графами

Рассмотренные в *главе 5* структуры данных и алгоритмы обхода предоставляют базовые конструктивные блоки для выполнения любых вычислений на графах. Но все эти алгоритмы применялись для *невзвешенных графов* (unweighted graphs), т. е. графов, в которых все ребра имеют одинаковый вес.

Но для *взвешенных графов* (weighted graph) существует отдельная область задач. В частности, ребрам графов дорожных сетей присваиваются такие числовые значения, как длина, скорость движения или пропускная способность данного отрезка дороги. Определение кратчайшего пути в таких графах оказывается более сложной задачей, чем обход в ширину в невзвешенных графах, но открывает путь к обширному диапазону приложений.

Рассматриваемые в *главе 5* структуры данных поддерживают графы со взвешенными ребрами неявно, но сейчас мы сделаем эту возможность явной. Для этого мы сначала определим структуру списка смежности, состоящую из массива связных списков, в которых ребра, исходящие из вершины x, указываются в списке `edges[x]` (листинг 6.1).

Листинг 6.1. Определение структуры списка смежности

```
typedef struct {
   edgenode *edges[MAXV+1];   /* Информация о смежности */
   int degree[MAXV+1];        /* Степень каждой вершины */
   int nvertices;             /* Количество вершин в графе */
   int nedges;                /* Количество ребер в графе */
   int directed;              /* Граф ориентированный? */
} graph;
```

Каждая переменная `edgenode` представляет собой запись из трех полей (листинг 6.2). Первое поле описывает вторую конечную точку ребра (`y`), второе (`weight`) предоставляет возможность присваивать ребру вес, а третье (`next`) указывает на следующее ребро в списке.

Листинг 6.2. Структура переменной edgenode

```
typedef struct {
   int y;                     /* Информация о смежности */
   int weight;                /* Вес ребра, если есть */
   struct edgenode *next;     /* Следующее ребро в списке */
} edgenode;
```

Далее мы рассмотрим несколько сложных алгоритмов, использующих эту структуру данных, включая алгоритмы поиска минимальных остовных деревьев, кратчайших

маршрутов и максимальных потоков. То обстоятельство, что для этих задач оптимизации существуют эффективные решения, привлекает к ним наше особое внимание. Вспомните, что для первой рассмотренной нами задачи для взвешенных графов (задачи коммивояжера) детерминистического алгоритма решения не существует.

6.1. Минимальные остовные деревья

Остовным деревом (spanning tree) графа $G = (V, E)$ называется подмножество ребер множества E, которые создают дерево, содержащее все вершины V. В случае взвешенных графов особый интерес представляют минимальные остовные деревья, т. е. остовные деревья с минимальной суммой весов ребер.

Минимальные остовные деревья позволяют решить задачу, в которой требуется соединить множество точек (представляющих города, дома, перекрестки и другие объекты) наименьшим объемом дорожного полотна, проводов, труб и т. п. Любое дерево — это, в сущности, наименьший (по количеству ребер) возможный связный граф, в то время как минимальное остовное дерево является наименьшим связным графом по весу ребер. В геометрических задачах набор точек $p_1, \ldots, p_n$ определяет полный граф, в котором ребру (v_i, v_j) присваивается вес, равный расстоянию между точками p_i и p_j. На рис. 6.1 показан пример геометрического минимального остовного дерева.

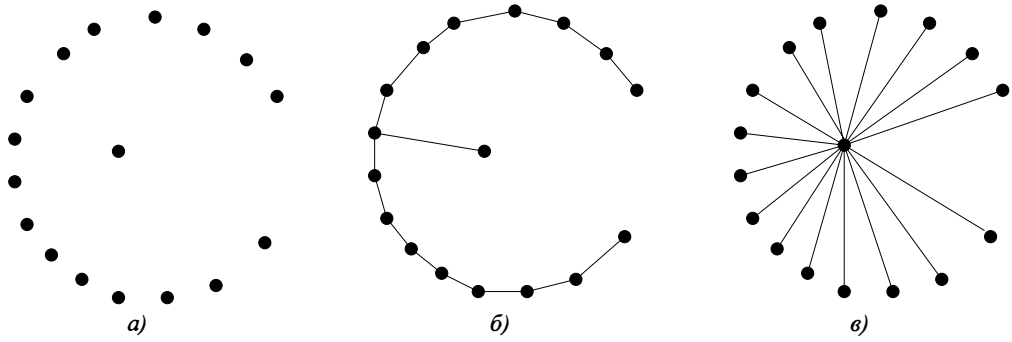


Рис. 6.1. Набор точек *(а)*, его минимальное остовное дерево *(б)* и кратчайший маршрут от центра дерева *(в)*

Дополнительную информацию о минимальных остовных деревьях см. в *разделе 15.3*.

Минимальное остовное дерево имеет наименьшую протяженность изо всех возможных остовных деревьев. Но граф может содержать несколько минимальных остовных деревьев. Более того, все остовные деревья невзвешенного графа (или графа со всеми ребрами одинакового веса) являются минимальными остовными деревьями, т. к. каждое из них содержит ровно $n - 1$ ребер. Такое остовное дерево можно найти с помощью алгоритма обхода в глубину или ширину. Найти минимальное остовное дерево во взвешенном графе гораздо труднее. Тем не менее, в последующих разделах рассматриваются два разных алгоритма для решения этой задачи, каждый из которых демонстрирует пригодность определенных эвристических "жадных" алгоритмов.

6.1.1. Алгоритм Прима

Алгоритм Прима для построения минимального остовного дерева начинает обход с одной вершины и создает дерево, добавляя по одному ребру, пока не будут включены все вершины.

"Жадные" алгоритмы принимают решение по следующему шагу, выбирая наилучшее локальное решение изо всех возможных вариантов, не принимая во внимание глобальную структуру. Так как мы ищем дерево с минимальным весом, то естественно предложить "жадный" алгоритм, последовательно выбирающий ребра с наименьшим весом, которые увеличат количество вершин дерева. Псевдокод алгоритма для построения минимального остовного дерева приведен в листинге 6.3.

Листинг 6.3. Алгоритм Прима

```
Prim-MST(G)
  Выбираем произвольную вершину s, с которой надо начинать
                                   построение дерева
    While (остаются вершины, не включенные в дерево)
     Выбираем ребро минимального веса между деревом и вершиной вне дерева
     Добавляем выбранное ребро и вершину в дерево Tprim
```

Очевидно, что алгоритм Прима создает остовное дерево, т. к. добавление ребра между вершиной в дереве и вершиной вне дерева не может создать цикл. Но почему именно это остовное дерево должно иметь наименьший вес изо всех остовных деревьев? Мы видели достаточно примеров других "жадных" эвристических алгоритмов, которые не дают оптимального общего решения. Поэтому нужно быть особенно осторожным, чтобы доказать любое такое утверждение.

В данном случае мы применим метод доказательства от противного. Допустим, что существует граф G, для которого алгоритм Прима не возвращает минимальное остовное дерево. Так как мы создаем дерево инкрементальным способом, то это означает, что существует момент, в который было принято неправильное решение. Перед добавлением ребра (x, y) дерево T_{prim} состояло из набора ребер, являющихся поддеревом определенного минимального остовного дерева T_{min}, но остовное дерево, полученное добавлением к этому поддереву ребра (x, y), больше не является минимальным (рис. 6.2, *а*).

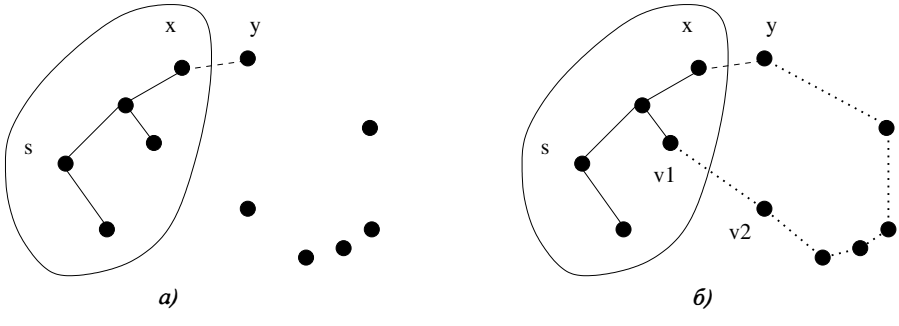


Рис. 6.2. Выдает ли алгоритм Прима неправильное решение? Нет, т. к. $d(v_1, v_2) \geq d(x, y)$

Но как это могло случиться? Как можно видеть на рис. 6.2, *б*, в остовном дереве T_{min} должен быть путь p от вершины x к вершине y. Этот путь должен содержать ребро (v_1, v_2), в котором вершина v_1 находится в дереве T_{prim}, а вершина v_2 — нет. Вес этого ребра должен быть, по крайней мере, равен весу ребра (x, y), иначе алгоритм Прима выбрал бы это ребро до ребра (x, y), когда у него была такая возможность. Удалив из дерева T_{min} ребро (v_1, v_2) и вставив вместо него ребро (x, y), мы получим остовное дерево с весом не большим, чем прежнее дерево. Это означает, что выбор алгоритмом Прима ребра (x, y) не был ошибочным. Таким образом, создаваемое алгоритмом Прима остовное дерево должно быть минимальным.

Реализация

Алгоритм Прима создает минимальное остовное дерево поэтапно, начиная с указанной вершины. На каждом этапе мы вставляем в остовное дерево одну новую вершину. "Жадного" алгоритма достаточно для гарантии правильности выбора: чтобы соединить вершины в дереве с вершиной вне дерева он всегда выбирает ребро с наименьшим весом. Самой простой реализацией этого алгоритма было бы представлять каждую вершину булевой переменной, указывающей, находится ли уже эта вершина в дереве (массив `intree` в листинге 6.4), а потом просмотреть все ребра на каждом этапе, чтобы найти ребро с минимальным весом и единственной вершиной `intree`. В листинге 6.4 приведена реализация алгоритма Прима.

Листинг 6.4. Реализация алгоритма Прима

```
prim(graph *g, int start)
{
   int  i;                /* Счетчик */
   edgenode *p;           /* Временный указатель */
   bool intree[MAXV+1];   /* Вершина уже в дереве? */
   int distance[MAXV+1]; /* Стоимость добавления к дереву */
   int v;                 /* Текущая вершина для обработки */
   int w;                 /* Кандидат на следующую вершину */
   int weight;            /* Вес ребра */
   int dist;              /* Наилучшее текущее расстояние от начала */
   for (i=1; i<=g->nvertices; i++) {
      intree[i] = FALSE;
      distance[i] = MAXINT;
      parent[i] = -1;
   }
   distance[start] = 0;
   v = start;
   while (intree[v] == FALSE) {
      intree[v] = TRUE;
      p = g->edges[v];
      while (p != NULL) {
         w = p->y;
         weight = p->weight;
         if ((distance[w] > weight) && (intree[w] == FALSE)) {
            distance[w] = weight;
```

```
            parent[w] = v;
         }
         p = p->next;
      }
      v = 1;
      dist = MAXINT;
      for (i=1; i<=g->nvertices; i++)
         if ((intree[i] == FALSE) && (dist > distance[i])) {
            dist = distance[i];
            v = i;
         }
   }
}
```

В процедуре поддерживается информация о весе каждого ребра, связывающего с деревом каждую находящуюся вне дерева вершину. Из множества этих ребер к дереву добавляется ребро с наименьшим весом. После каждой вставки ребра необходимо обновлять стоимость (в виде весов ребер) просмотра вершин, не входящих в дерево. Но так как единственным изменением в дереве является последняя добавленная вершина, то все возможные обновления весов будут определяться исходящими ребрами этой вершины.

Анализ

Итак, алгоритм Прима является правильным, но насколько он эффективен? Это будет зависеть от используемых для его реализации структур данных. В псевдокоде алгоритм Прима исполняет n циклов, просматривая все m ребер в каждом цикле, что дает нам алгоритм с временной сложностью, равной $O(mn)$.

Но в нашей реализации проверка всех ребер в каждом цикле не выполняется. Мы рассматриваем не более n известных ребер с наименьшим весом, представленных в массиве `parent`, и не более n ребер, исходящих из последней добавленной вершины v для обновления этого массива. Благодаря наличию булева флага для каждой вершины, указывающего, находится ли эта вершина в дереве, мы можем проверить, связывает ли текущее ребро дерево с вершиной вне дерева, за постоянное время.

Общее время исполнения алгоритма Прима равно $O(n^2)$, что является хорошей иллюстрацией того, как удачная структура данных способствует ускорению алгоритмов. Более того, применение более сложной структуры данных в виде очереди с приоритетами позволяет осуществить реализацию алгоритма Прима с временной сложностью, равной $O(m + n\lg n)$, поскольку удается быстрее найти ребро с минимальным весом для расширения дерева в каждом цикле.

Само минимальное остовное дерево или его вес можно реконструировать двумя разными способами. Самым простым способом было бы дополнить код в листинге 6.4 операторами для вывода обнаруженных ребер или общего веса всех выбранных ребер. В качестве альтернативы можно закодировать топологию дерева в массиве `parent`, чтобы она совместно с первоначальным графом предоставляла всю информацию о минимальном остовном дереве.

6.1.2. Алгоритм Крускала

Алгоритм Крускала является альтернативным подходом к построению минимальных остовных деревьев, который оказывается более эффективным на разреженных графах. Подобно алгоритму Прима, алгоритм Крускала является "жадным", но, в отличие от него, он не создает дерево, начиная с определенной вершины.

Алгоритм Крускала наращивает связные компоненты вершин, создавая в конечном счете минимальное остовное дерево. Первоначально каждая вершина является отдельной компонентой будущего дерева. Алгоритм последовательно ищет ребро для добавления в расширяющийся лес путем поиска самого легкого ребра среди всех ребер, соединяющих два дерева в лесу. При этом выполняется проверка, не находятся ли обе конечные точки ребра-кандидата в одной и той же связной компоненте. В случае положительного результата проверки такое ребро отбрасывается, т. к. добавление его создало бы цикл в будущем дереве. Если же конечные точки находятся в разных компонентах, то ребро принимается и соединяет две компоненты в одну. Так как каждая связная компонента всегда является деревом, то выполнять явную проверку на циклы нет необходимости.

В листинге 6.5 приводится псевдокод алгоритма Крускала.

Листинг 6.5. Алгоритм Крускала

```
Kruskal-MST(G)
   Помещаем ребра в очередь с приоритетами, упорядоченную по весу
   count = 0
   while (count < n - 1) do
      рассматриваем следующее ребро (v, w)
      if (component (v) ≠ component(w))
         добавляем в дерево Tkruskal
         объединяем component(v) и component(w)
```

Так как этот алгоритм вставляет $n - 1$ ребер, не порождая циклы, он, очевидно, создает остовное дерево для любого связного графа. Но откуда мы знаем, что это *минимальное* остовное дерево? Будем доказывать от противного. Допустим, что оно не является таковым. Так же, как и в доказательстве правильности алгоритма Прима, допустим, что существует какой-то граф, для которого он возвращает неправильный ответ (рис. 6.3).

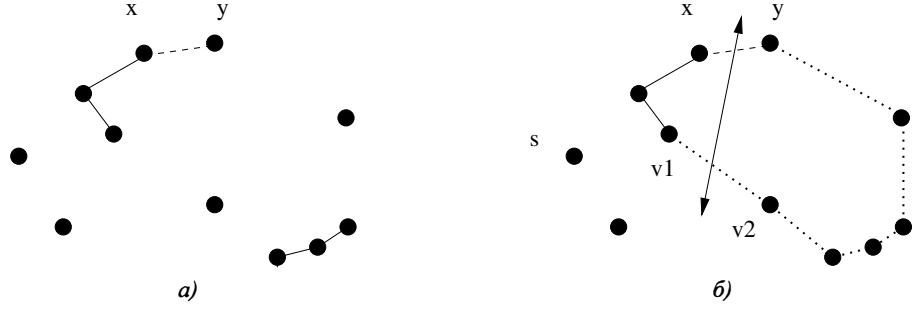


Рис. 6.3. Выдает ли алгоритм Крускала неправильное решение? Нет, т. к. $d(v_1, v_2) \geq d(x, y)$

В частности, должно существовать ребро (x, y), первоначальная вставка которого в дерево $T_{kruskal}$ не дает ему быть минимальным остовным деревом T_{min}. Вставка такого ребра (x, y) в дерево T_{min} создаст цикл с путем от вершины x к вершине y. Так как вершины x и y были в разных компонентах во время вставки ребра (x, y), то, по крайней мере, одно ребро (скажем, ребро (v_1, v_2)) в этом пути было бы проверено алгоритмом Крускала позже, чем ребро (x, y). Но это означает, что вес $w(v_1, v_2) \geq w(x, y)$, поэтому обмен местами этих двух ребер дает нам дерево с весом, самое большее, T_{min}. Поэтому выбор ребра (x, y) не мог быть ошибочным, из чего следует правильность алгоритма.

Какова временная сложность алгоритма Крускала? Упорядочивание m-го количества ребер занимает время $O(m \lg m)$. Цикл `for` выполняет m итераций, в каждой из которых проверяется связь двух деревьев и ребро. При наиболее очевидном подходе это можно реализовать посредством поиска в ширину или глубину в разреженном графе, имеющем, самое большее, m ребер и n вершин, что дает нам алгоритм с временем исполнения $O(mn)$.

Но если удастся выполнять проверку компонент быстрее, чем за время $O(n)$, то можно получить более эффективную реализацию алгоритма. В действительности, рассматриваемая в следующем разделе сложная структура данных позволяет выполнять такую проверку за время $O(\lg n)$. С использованием этой структуры алгоритм Крускала исполняется за время $O(m \lg m)$, что быстрее, чем время исполнения алгоритма Прима для разреженных графов. В очередной раз обращаю ваше внимание на эффект, оказываемый правильной структурой данных на реализацию простых алгоритмов.

Реализация

Реализация алгоритма Крускала показана в листинге 6.6.

Листинг 6.6. Реализация алгоритма Крускала

```
kruskal(graph *g)
{
   int i;                        /* Счетчик */
   set_union s;                  /* Структура данных set_union */
   edge_pair e[MAXV+1];          /* Структура данных массива ребер */
   bool weight_compare();
   set_union_init(&s, g->nvertices);
   to_edge_array(g, e);        /* Сортируем ребра по возрастающему весу */
   qsort(&e, g->nedges, sizeof(edge_pair),weight_compare);
   for (i=0; i<(g->nedges); i++) {
      if (!same_component(s,e[i].x,e[i].y)) {
         printf("edge (%d,%d) inMST\n",e[i].x,e[i].y);
         union_sets(&s,e[i].x,e[i].y);
      }
   }
}
```

На рис. 6.4 показаны граф G *(а)* и минимальные остовные деревья для него, создаваемые алгоритмами Прима *(б)* и Крускала *(в)*. Числами возле ребер указан порядок их вставки; приоритет вставки равнозначных ребер определяется произвольным путем.

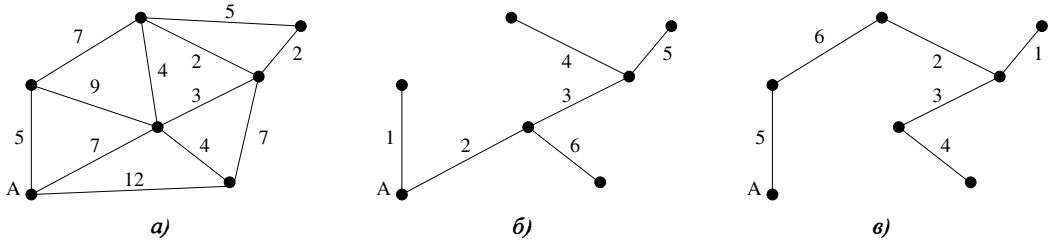


Рис. 6.4. Граф G и минимальные остовные деревья, создаваемые алгоритмами Прима и Крускала

6.1.3. Поиск-объединение

Разбиением множества (set partition) называется разбиение элементов некоего универсального множества (например, целых чисел от 1 до n) на несколько непересекающихся (disjointed) подмножеств, таким образом, каждый элемент исходного множества должен быть ровно в одном из получившихся подмножеств. Разбиения множества возникают естественным образом в таких задачах на графах, как связные компоненты (каждая вершина находится ровно в одной компоненте связности) и раскраска графа (человек может быть женского или мужского пола, но мы исключаем варианты отсутствия пола или принадлежности к обоим полам). Алгоритмы для генерирования разбиений множеств и связных объектов представлены в *разделе 14.6*.

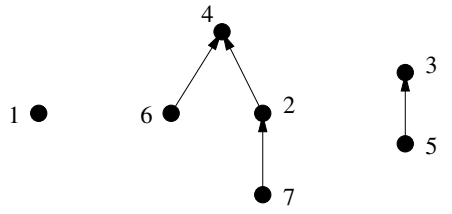
Связные компоненты графа можно представить в виде разбиения множества. Для эффективного исполнения алгоритма Крускала нам нужна структура данных, обеспечивающая эффективную поддержку таких операций:

- *одна компонента* (v_1, v_2) — выполняет проверку, находятся ли вершины v_1 и v_2 в одной и той же компоненте связности текущего графа;
- *объединить компоненты* (C_1, C_2) — объединяет данную пару связных компонент в одну по предоставленному ребру между ними.

Каждая из двух очевидных структур-кандидатов для поддержки этих операций в действительности обеспечивает эффективную поддержку только одной из них. В частности, явно пометив каждый элемент номером его компоненты, мы сможем выполнять проверку "одна компонента" за постоянное время, но обновление номеров компонент после слияния будет занимать линейное время. С другой стороны, операцию слияния компонент можно рассматривать как вставку ребра в граф, но тогда нам нужно выполнить полный обход графа, чтобы найти связные компоненты в случае необходимости. Решением является применение специальной структуры данных (рис. 6.5), где каждое подмножество представляется в виде "обратного" дерева с указателями от узлов к их родителям.

Каждый узел дерева содержит элемент множества, а *имя* множеству присваивается по корневому ключу. По причинам, которые станут понятными далее, мы для каждой вершины v отслеживаем также информацию о количестве элементов поддерева, имеющего корень в этой вершине.

В листинге 6.7 приводится определение структуры данных `set_union`.



1	2	3	4	5	6	7
0	4	0	0	3	4	2

б)

Рис. 6.5. Пример структуры данных: в виде деревьев *(а)* и в виде массива *(б)*

Листинг 6.7. Определение структуры данных `set_union`

```
typedef struct {
   int p[SET_SIZE+1];      /* Родительский элемент */
   int size [SET_SIZE+1]; /* Количество элементов в поддереве i */
   int n;                  /* Количество элементов в множестве */
} set_union;
```

Реализуем необходимые нам операции над компонентами посредством двух более простых операций — `find` и `union`:

- `find(i)` — находит корень дерева, содержащего элемент *i*, переходя по указателям на родителей до тех пор, пока это возможно. Возвращает метку корня;
- `union(i,j)` — связывает корень одного из деревьев (скажем, дерева, содержащего элемент *i*) с корнем дерева, содержащего другой элемент (скажем, элемент *j*); таким образом, операция `find(i)` теперь эквивалентна операции `find(j)`.

Мы стремимся минимизировать время исполнения *любой* последовательности операций `union` и `find`. Так как древесные структуры могут быть сильно разбалансированными, то нам необходимо ограничить высоту наших деревьев. Самым очевидным способом контроля высоты будет принятие правильного решения, какой из двух корней компонент должен стать корнем объединенной компоненты после каждого выполнения операции `union`.

Минимизировать высоту дерева можно, сделав меньшее дерево поддеревом большего. Почему это так? Потому что высота всех узлов в корневом поддереве остается одинаковой, в то время как высота всех узлов, вставленных в это дерево, увеличивается на единицу. Таким образом, объединение с меньшим деревом позволяет оставить без изменений высоту большего дерева.

Реализация

Реализация операций `union` и `find` приводится в листинге 6.8.

Листинг 6.7. Реализация операций `union` и `find`

```
set_union_init(set_union *s, int n)
{
   int i;                          /* Счетчик */
   for (i=1; i<=n; i++) {
      s->p[i] =i;
```

```
      s->size[i] = 1;
   }
   s->n = n;
}
int find(set_union *s, int x)
{
   if (s->p[x] == x)
      return(x);
   else
      return(find(s, s->p[x]) );
}
int union_sets(set_union *s, int s1, int s2)
{
   int r1, r2;               /* Корни подмножеств */
   r1 = find(s, s1);
   r2 = find(s, s2);
   if (r1 == r2) return;     /* Уже находится в этом множестве */
   if (s->size[r1] >= s->size[r2]) {
      s->size[r1] = s->size[r1] + s->size[r2];
      s->p[ r2 ] = r1;
   }
   else {
      s->size[r2] = s->size[r1] + s->size[r2];
      s->p[ r1 ] = r2;
   }
}
bool same_component(set_union *s, int s1, int s2)
{
   return ( find(s, s1) == find(s, s2) );
}
```

Анализ

При каждом исполнении операции `union` дерево с меньшим количеством узлов становится потомком. Но каким высоким может быть такое дерево в зависимости от количества узлов в нем? Рассмотрим, каким может быть наименьшее возможное дерево с высотой h. Высота деревьев с одним узлом равна 1. Наименьшее дерево высотой 2 имеет два узла, полученных в результате объединения двух одноузловых деревьев. Когда изменяется высота? При объединении двух одноузловых деревьев этого не происходит, т. к. эти деревья просто становятся потомками корневого дерева высотой 2. И только при объединении двух деревьев высотой 2 мы получаем дерево высотой 3, имеющее четыре узла.

Уловили закономерность? Высота дерева увеличивается на единицу при удваивании количества узлов. Сколько раз мы можем удваивать количество узлов, пока не исчерпаем все n узлов? Нам удастся сделать максимум $\lg_2 n$ удваиваний. Таким образом, мы можем выполнять как операцию объединения `union`, так и операцию поиска `find` за время $O(\log n)$, что достаточно хорошо для алгоритма Крускала. В действительности, поиск-объединение может выполняться даже быстрее (см. *раздел 12.5*).

6.1.4. Разновидности остовных деревьев

Алгоритм поиска минимального остовного дерева обладает несколькими интересными свойствами, полезными для решения родственных задач, в частности, поиска таких разновидностей остовных деревьев, как:

- *максимальные остовные деревья.* Допустим, телефонная компания наняла подрядчика для прокладки телефонного кабеля между домами. Компания платит подрядчику с учетом протяженности установленного кабеля. К сожалению, подрядчик оказался недобросовестным, и пытается получить наибольший доход от этой работы. В терминах теории графов для этого ему нужно создать наиболее тяжелое остовное дерево. *Максимальное остовное дерево* любого графа можно найти, просто поменяв у всех ребер знак веса с плюса на минус и применив алгоритм Прима. Максимальное по модулю дерево в графе с отрицательным весом ребер будет максимальным остовным деревом в первоначальном графе.

  Большинство алгоритмов для графов не поддается адаптации под отрицательные числа с такой легкостью. Более того, алгоритмы поиска кратчайшего пути некорректно работают с отрицательными числами и не возвращают самый длинный путь при использовании этого подхода;

- *остовные деревья с минимальным произведением весов ребер.* Допустим, нам нужно найти остовное дерево с минимальным произведением весов ребер, полагая, что вес всех ребер положительный. Так как $\lg(a \cdot b) = \lg(a) + \lg(b)$, то минимальное остовное дерево графа, в котором веса ребер были заменены их логарифмами, даст нам остовное дерево с минимальным произведением весов ребер первоначального графа;

- *минимальное остовное дерево.* Иногда требуется найти остовное дерево, в котором наибольший вес ребра минимален среди всех таких деревьев. Любое минимальное остовное дерево обладает этим свойством. Доказательство этого следует непосредственно из факта правильности алгоритма Крускала.

  Минимальные остовные деревья имеют интересные области применения, когда вес ребра представляет стоимость, пропускную способность и т. д. Менее эффективным, но концептуально более простым способом решения таких задач может быть удаление всех "тяжелых" ребер графа с проверкой полученного графа на связность. Эту проверку можно выполнить простым обходом в ширину или глубину.

Если все m ребер графа имеют разный вес, то такой граф имеет единственное минимальное остовное дерево. В противном случае возвращаемое алгоритмом Прима или Крускала дерево определяется способом, который применяется в алгоритме для выбора одного из нескольких ребер с одинаковым весом.

Существует две важных разновидности минимальных остовных деревьев, которые *не* поддаются решению этими способами.

- *Дерево Штейнера.* Допустим, нам нужно проложить проводку (ребра графа) между домами (вершины графа), но при выполнении этой задачи мы можем создавать дополнительные промежуточные пункты (вершины) в качестве общих соединений. Дерево, полученное в результате решения этой задачи, называется минимальным деревом Штейнера и рассматривается в *разделе 16.10*.

- *Остовное дерево с минимальной степенью вершин.* Предположим, нам нужно найти минимальное остовное дерево, у которого наибольшая степень вершины минимальна. Таким деревом будет простой путь с $n-2$ вершинами степени 2 и двумя конечными вершинами степени 1. Путь, который проходит через каждую вершину только один раз, называется *гамильтоновым путем*, и рассматривается в *разделе 16.5.*

6.2. История из жизни. И все на свете только сети

Однажды мне сообщили, что небольшая компания по тестированию печатных плат нуждается в консультации по алгоритмам. Вскоре я оказался в непримечательном здании за одним столом с президентом компании Integri-Test и ведущим техническим специалистом.

— Мы являемся ведущей компанией, поставляющей роботизированные устройства для тестирования печатных плат. Наши клиенты предъявляют очень высокие требования к надежности выпускаемых ими печатных плат. В частности, перед тем как устанавливать компоненты на печатную плату, они проверяют, что на ней нет разрывов в дорожках. Это означает, что им нужно проверить, что каждая пара точек на плате, которые должны быть соединенными, в действительности соединены.

— И как вы выполняете это тестирование? — спросил я.

— Мы используем два роботизированных манипулятора, оснащенные электрическими щупами. Для проверки соединения между двумя точками манипуляторы подключают щупы к этим обеим точкам. Если дорожка между точками исправна, то щупы замыкают цепь. А для сетей мы фиксируем один из манипуляторов в одной точке, а вторым последовательно проверяем все остальные точки сети.

— Погодите! — вскричал я. — Что такое сеть на печатной плате?

— Печатная плата содержит несколько наборов точек, соединенных между собой посредством металлизированных дорожек (рис. 6.6, *а*). Вот эти наборы точек и соединяющие их дорожки мы и называем сетями. Иногда сеть состоит из двух точек, т. е. из одного изолированного провода, а иногда содержит 100-200 точек, как в случае с проводом питания или заземления.

— Понятно. Значит, у вас есть список всех соединений между парами точек на печатной плате и вы хотите проверить целостность этих соединяющих дорожек.

Он отрицательно покачал головой. — Не совсем так. Вход для нашей программы тестирования состоит только из контактных точек сети (рис. 6.6, *б*). Мы не знаем расположения отрезков соединяющей дорожки, но это и не требуется. Все, что нам нужно, — это проверить, что точки сети соединены вместе. Для этого один щуп приставляется к самой левой точке сети, а второй перемещается по всем остальным точкам, проверяя, что все они соединены с первой точкой. В таком случае они должны быть соединены друг с другом.

— Хорошо. Значит, правый манипулятор должен посетить все остальные точки сети. А каким образом вы определяете последовательность этих посещений?

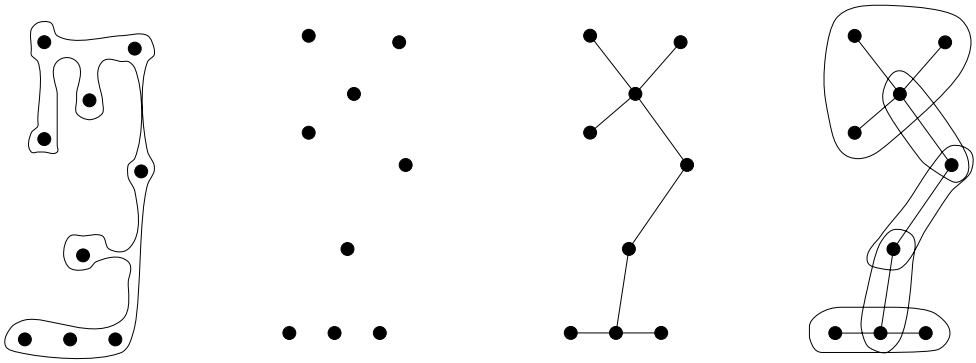


Рис. 6.6. Пример сети печатной платы: металлизированная соединяющая дорожка *(а)*, контактные точки *(б)*, минимальное остовное дерево контактных точек *(в)*, контактные точки, разбитые на кластеры *(г)*

На этот вопрос ответил технический специалист. Мы сортируем точки слева направо и обходим их в этом порядке. Это хороший способ?

— Вы что-нибудь знаете о задаче коммивояжера? — спросил я.

Он был инженер-электрик, а не программист. — Нет. А что это?

— Это название задачи, которую вы пытаетесь решить. Нужно посетить набор точек в порядке, который минимизирует время обхода. Алгоритмы для решения этой задачи всесторонне изучены. Для небольших сетей можно найти оптимальный маршрут методом полного перебора. А для больших сетей очень хорошее приближение оптимального маршрута можно найти посредством эвристических алгоритмов. — Для более подробной информации о решении задачи коммивояжера я бы показал им *раздел 16.4* этой книги, если бы она была у меня под рукой.

Президент компании что-то записал в своем блокноте, а потом нахмурился. — Хорошо. Допустим, вы сможете лучше упорядочить точки. Но проблема заключается не в этом. Когда мы наблюдаем за работой нашего робота, то иногда видим, что правый манипулятор доходит до самого правого края платы для данной сети, в то время как левый манипулятор стоит неподвижно. Мне кажется, что было бы полезно разбивать сети на меньшие части, чтобы сбалансировать нагрузку на манипуляторы.

Я уселся поудобнее и стал обдумывать задачу. Оба манипулятора (левый и правый) решают взаимосвязанные задачи коммивояжера. Левый манипулятор перемещается от одной к другой самой левой точке каждой сети, в то время как правый посещает все другие точки каждой сети. Разбив каждую сеть на несколько меньших сетей, мы можем избежать ситуации, когда правый манипулятор проходит путь через всю плату. Кроме того, разбивка всей сети на части увеличит количество точек в задаче коммивояжера для левого манипулятора, вследствие чего каждое его перемещение также будет небольшим.

— Вы правы. Разбивка больших сетей на несколько меньших должна повысить эффективность работы манипуляторов. Сети должны быть небольшого размера, как по количеству точек, так и по физической площади. Но нам необходимо иметь гарантию в том,

что если мы проверим связность каждой малой сети, то также проверим связность большой сети, членами которой они являются. Одной общей точки для двух малых сетей будет достаточно, чтобы доказать, что состоящая из этих сетей большая сеть также связная, т. к. ток может протекать между любой парой точек.

Теперь нам нужно было решить задачу разбиения каждой сети на несколько меньших сетей, что является задачей кластеризации, или группирования. Для группирования часто применяются минимальные остовные деревья (см. *раздел 15.3*). Это и было решением нашей конкретной задачи кластеризации. В частности, мы могли найти минимальное остовное дерево точек сети и разбить его на несколько кластеров, разрывая любое слишком длинное ребро остовного дерева (рис. 6.6, *в*). Как можно видеть на рис. 6.6, *г*, каждый кластер точек будет иметь ровно одну общую точку с другим кластером, что обеспечивает связность, т. к. мы охватываем все ребра остовного дерева. Форма кластеров следует из расположения точек в сети. Если точки расположены на плате вдоль линии, то минимальным остовным деревом будет маршрут, а кластерами будут пары точек. А если точки расположены плотной группой, то этот кластер будет хорошей площадкой для игры в классики правым манипулятором.

Я объяснил моим собеседникам идею создания минимального остовного дерева графа. Президент выслушал, что-то опять записал в своем блокноте и снова нахмурился.

— Мне нравится ваша идея кластеризации. Но минимальные остовные деревья определяются на графах, а у нас есть только точки. Где мы возьмем веса для ребер?

— О, мы можем рассматривать их как полный граф, в котором соединена каждая пара точек. А вес ребра определяется как расстояние между двумя точками. Или не определяется...?

Я опять начал размышлять. Стоимость ребра должна отражать время перемещения манипулятора между двумя точками. В то время как расстояние связано с временем перемещения, это не обязательно взаимозаменяемые понятия.

— У меня вопрос относительно вашего робота. Скорость перемещения манипулятора по горизонтали такая же, как и по вертикали?

Они подумали немного. — Да, такая же. Мы используем одинаковые двигатели для перемещения манипулятора по горизонтали и вертикали. Так как оба двигателя работают независимо друг от друга, то манипулятор можно перемещать одновременно по горизонтали и по вертикали.

— То есть, перемещение на один фут влево и один фут вверх занимает точно такое же время, как перемещение на один фут только влево? Это означает, что вес каждого ребра должен соответствовать не евклидову расстоянию между двумя точками, а наибольшему расстоянию между ними либо по горизонтали, либо по вертикали. Это называется метрикой L_∞, но мы можем зафиксировать ее, изменяя вес ребер в графе. Происходит ли еще что-нибудь необычное с вашими роботами? — спросил я.

— Роботу нужно некоторое время, чтобы развить рабочую скорость. Также, наверное, нужно принять во внимание ускорение и замедление движения манипулятора.

— Абсолютно верно. Чем точнее мы смоделируем перемещение манипулятора между двумя точками, тем лучшим будет наше решение. Но теперь у нас есть очень четкая формулировка решения. Давайте закодируем его и посмотрим, насколько оно хорошо.

Они явно сомневались, что этот подход принесет каку -нибудь пользу, но согласились подумать. Несколько недель спустя они позвонили мне и сообщили, что новый алгоритм сокращает дистанцию маршрута примерно на 30% по сравнению с их первоначальным подходом за счет небольшого увеличения предпроцессорных вычислений. Но так как их тестовое оборудование стоило 200 000 долларов, а персональный компьютер — всего лишь 2 000 долларов, то это был замечательный компромисс. Кроме того, это было особенно выгодным, т. к. в случае тестирования партии одинаковых плат предварительную обработку нужно было выполнять только один раз.

Ключевой идеей, приведшей к успешному решению, было моделирование задачи посредством классических алгоритмов для решения задач на графах. Я понял, что передо мной задача коммивояжера, как только разговор зашел о минимизации протяженности перемещений манипулятора. Когда я выяснил, что для проверки связности точек они неявно использовали звездообразное остовное дерево, было естественным задать вопрос, не улучшило бы производительность минимальное остовное дерево. Эта идея проложила путь к идее кластеризации, т. е. разбиению каждой сети на несколько меньших сетей. Наконец, внимательный подход к разработке метрик расстояний, чтобы точно смоделировать издержки самого манипулятора, позволил нам внедрить в решение такие сложные свойства, как ускорение и замедление манипулятора, не изменяя фундаментальной модели графа или структуры алгоритма.

Подведение итогов

Большинство приложений с использованием графов можно свести к стандартным задачам на графах, к которым можно применить хорошо известные алгоритмы. Это такие задачи, как построение минимальных остовных деревьев, кратчайших путей и прочие задачи, представленные в каталоге задач этой книги.

6.3. Поиск кратчайшего пути

Путь (path) — это последовательность ребер, соединяющих две вершины. Так как кинорежиссер Мел Брукс (Mel Brooks) приходится двоюродным братом мужу сестры моего отца, то в графе дружеских отношений между мной и ним есть путь (рис. 6.7), хотя мы никогда лично не встречались.

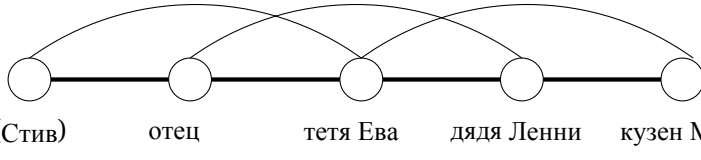


Рис. 6.7. Мел Брукс приходится двоюродным братом мужу сестры моего отца

Но если бы я хотел произвести на читателей впечатление, заявляя о близком знакомстве с кузеном Мелом, то лучше бы было говорить, что мой дядя Ленни и он росли вместе. Через дядю Ленни длина пути дружеских отношений между мной и кузеном Мелом составляет 2 звена, а длина пути кровных и родственных связей составляет 4 звена. Наличие нескольких путей свидетельствует о важности определения *кратчайшего пути* между двумя узлами, даже в приложениях, не имеющих отношения к транспортировке.

В невзвешенном графе кратчайший путь от вершины *s* к вершине *t* можно найти посредством обхода в ширину с исходной точкой в вершине *s*. Дерево обхода в ширину предоставляет путь, состоящий из минимального количества ребер. Это кратчайший путь, если все ребра имеют одинаковый вес.

Но во взвешенных графах обход в ширину *не* предоставляет кратчайшего пути. В данном типе графа кратчайший путь может содержать большее количество ребер, точно так же, как и кратчайший (во временном измерении) путь от дома к офису может проходить по нескольким отрезкам второстепенных дорог, вместо одного или двух прямых отрезков главной дороги, как показано на рис. 6.8.

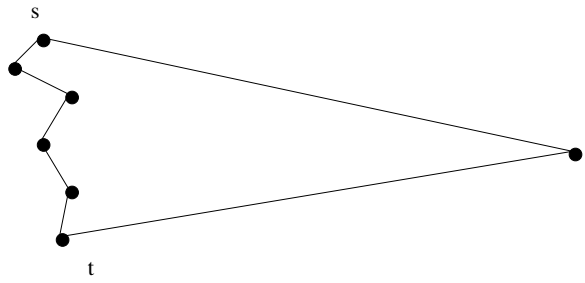


Рис. 6.8. Кратчайший путь от точки *s* к точке *t* может проходить через несколько промежуточных точек

В этом разделе мы рассмотрим два разных алгоритма поиска кратчайшего пути во взвешенных графах.

6.3.1. Алгоритм Дейкстры

Алгоритм Дейкстры является предпочтительным методом поиска кратчайших путей в графах со взвешенными ребрами и/или вершинами. Алгоритм находит кратчайший путь от заданной начальной вершины *s* ко всем другим вершинам графа, включая требуемую конечную вершину *t*.

Допустим, что кратчайший путь от вершины *s* к вершине *t* графа *G* проходит через определенную промежуточную вершину *x*. Очевидно, что этот путь должен содержать кратчайший путь от вершины *s* к вершине *x*, в качестве префикса, ибо в противном случае можно было бы сократить путь *s*–*t*, используя более короткий префиксный путь *s*–*x*. Таким образом, прежде чем найти кратчайший путь от начальной вершины *s* к конечной вершине *t*, нам нужно найти кратчайший путь от начальной вершины *s* к промежуточной вершине *x*.

Алгоритм Дейкстры работает поэтапно, находя на каждом этапе кратчайший путь от вершины *s* к некой *новой* вершине. Говоря конкретно, вершина *x* такова, что сумма $dist(s, v_i) + w(v_i, x)$ минимальна для всех необработанных $1 \leq i \leq n$, где $w(i, j)$ — длина ребра между вершинами *i* и *j*, а $dist(i, j)$ — длина кратчайшего пути между ними.

Здесь напрашивается стратегия, аналогичная динамическому программированию. Кратчайший путь от вершины *s* к самой себе является тривиальным, при условии отсутствия ребер с отрицательным весом, поэтому $dist(s, s) = 0$. Если (s, y) является са-

мым легким ребром, входящим в вершину s, то это означает, что $dist(s, y) = w(s, y)$. Определив кратчайший путь к вершине x, мы проверяем все исходящие из нее ребра, чтобы узнать, не существует ли лучшего пути из начальной вершины s к какой-либо неизвестной вершине через вершину x. Псевдокод этого алгоритма представлен в листинге 6.9.

Листинг 6.9. Алгоритм Декстры

```
ShortestPath-Dijkstra(G, s, t)
known = {s}
for i = 1 to n, dist[i] = ∞
for each edge (s, v), dist[v] = w(s, v)
last = s
while (last ≠ t)
   select v_next, неизвестная вершина, минимизирующая dist[v]
   for each edge (v_next, x), dist[x] = min[dist[x],
                              dist[v_next] + w(v_next, x) ]
   last = v_next
   known = known ∪ {v_next}
```

Основная идея алгоритма Дейкстры очень похожа на основную идею алгоритма Прима. В каждом цикле мы добавляем одну вершину к дереву вершин, для которых мы *знаем* кратчайший путь из вершины s. Так же, как и в алгоритме Прима, мы сохраняем информацию о наилучшем пути на данное время для всех вершин вне дерева и вставляем их в дерево в порядке возрастания веса.

Разница между алгоритмом Дейкстры и алгоритмом Прима состоит в том, как они оценивают привлекательность каждой вершины вне дерева. В задаче поиска минимального остовного дерева нас интересовал только вес следующего возможного ребра дерева. А при поиске кратчайшего пути нам нужна также информация о ближайшей к вершине s вершине вне дерева, т. е., кроме веса нового ребра, мы хотим знать еще *и расстояние* от вершины s к смежной с ней вершине дерева.

Реализация

Псевдокод алгоритма Дейкстры скрывает его сходство с алгоритмом Прима. В действительности, разница между этими двумя алгоритмами очень незначительная. В листинге 6.10 приводится реализация алгоритма Дейкстры, которая, по сути, является реализацией алгоритма Прима, в которой изменены всего лишь три строчки кода (отмечены в листинге комментарием) и имя функции.

Листинг 6.10. Реализация алгоритма Декстры

```
dijkstra(graph *g, int start)    /* Для Прима — prim(g, start) */
{
   int i;                    /* Счетчик */
   edgenode *p;              /* Временный указатель */
   bool intree[MAXV+1];      /* Вершина уже в дереве? */
   int distance[MAXV+1];     /* Расстояние вершины от начала */
   int v;                    /* Текущая вершина для обработки */
```

```
   int w;                    /* Кандидат на следующую вершину */
   int weight;               /* Вес ребра */
   int dist;                 /* Наилучшее текущее расстояние от начала */
   for (i=1; i<=g->nvertices; i++) {
      intree[i] = FALSE;
      distance[i] = MAXINT;
      parent[i] = -1;
   }
   distance[start] = 0;
   v = start;
   while (intree[v] == FALSE) {
      intree[v] = TRUE;
      p = g->edges[v];
      while (p != NULL) {
         w = p->y;
         weight = p->weight;
   /* ИЗМЕНЕНИЕ */   if (distance[w] > (distance[v]+weight))
   /* ИЗМЕНЕНИЕ */      distance[w] = distance[v]+weight;
   /* ИЗМЕНЕНИЕ */      parent[w] = v;
                    }
                    p = p->next;
         }
         v = 1;
         dist = MAXINT;
         for (i=1; i<=g->nvertices; i++)
            if ((intree[i] == FALSE) && (dist > distance[i])) {
               dist = distance[i];
               v = i;
            }
   }
}
```

Этот алгоритм находит не только кратчайший путь от вершины s к вершине t, но и кратчайший путь от вершины s ко всем другим вершинам, что определяет минимальное остовное дерево с корнем в вершине s. Для неориентированных графов это будет дерево обхода в ширину, но, в общем, алгоритм предоставляет кратчайший путь от вершины s ко всем остальным вершинам.

Анализ

Какова временная сложность алгоритма Дейкстры? В том виде, как он реализован здесь, временная сложность алгоритма равна $O(n^2)$. Это такое же время исполнения, как и для алгоритма Прима, что не удивительно — ведь за исключением дополнительного условия, это точно такой же алгоритм, как и алгоритм Прима.

Длина кратчайшего пути от начальной вершины `start` к заданной вершине t равна точно значению `distance[t]`. Каким образом мы можем найти сам путь с помощью алгоритма Дейкстры? Точно так же, как в процедуре `find_path()` (см. листинг 5.13), мы следуем обратным указателям на родительские узлы от вершины t, пока мы не дойдем до начальной вершины (или пока не получим –1, если такого пути не существует).

Алгоритм Дейкстры работает правильно только на графах, в которых нет ребер с отрицательным весом. Дело в том, что при построении пути может встретиться ребро с отрицательным весом настолько большим по модулю, что оно полностью изменит оптимальный путь от вершины s к какой-то другой вершине, которая уже включена в дерево. Образно говоря, самым выгодным путем к соседу по лестничной клетке может оказаться путь через банк на другом конце города, если этот банк выдает за каждое посещение достаточно большое вознаграждение, делающее такой маршрут выгодным.

В большинстве приложений ребра с отрицательным весом не используются, что делает это ограничение на пригодность алгоритма Дейкстры чисто теоретическим. Рассматриваемый далее алгоритм Флойда также работает правильно только в том случае, если отсутствуют циклы с отрицательной стоимостью, которые заметно искажают структуру кратчайшего пути. Если банк в нашем примере не ограничивает вознаграждение одноразовой выплатой, то выгода от его бесконечного посещения заставит вас *навсегда* отказаться от идеи навестить соседа.

Остановка для размышлений. Кратчайший путь с учетом веса вершин

ЗАДАЧА. Допустим, что нам нужно разработать эффективный алгоритм для поиска самого дешевого пути в графе со взвешенными вершинами, а не со взвешенными ребрами. То есть, стоимостью пути от вершины x к вершине y является сумма весов всех вершин в пути.

Решение. Естественным решением будет адаптация алгоритма для графов со взвешенными ребрами (например, алгоритма Дейкстры) для этой ситуации. Очевидно, что такой подход возможен. Мы просто заменим все упоминания веса ребра на вес его конечной вершины. Эту информацию можно хранить в массиве и обращаться к ней по мере надобности.

Но лично я предпочитаю вместо этого модифицировать граф таким образом, чтобы алгоритм Дейкстры дал желаемый результат. Для этого мы установим в качестве веса ориентированного ребра (i, j) вес вершины j. Применение алгоритма Дейкстры на таком графе дает требуемый результат.

Эту технику можно распространить на другие задачи, например, такие, в которых вес имеется как у ребер, так и у вершин. ■

6.3.2. Кратчайшие пути между всеми парами вершин

Допустим, что нам нужно найти "центральную" вершину графа, т. е. вершину с кратчайшими путями ко всем остальным вершинам. Практическим примером такой задачи может быть выбор места для открытия офиса. Или, допустим, вам нужно узнать диаметр графа, т. е. максимальное кратчайшее расстояние между всеми парами вершин. Практическим примером данной задачи может быть определение максимального временного интервала для доставки письма или сетевого пакета. В этих случаях нужно найти кратчайшие пути между всеми парами вершин графа.

Одним из возможных решений этой задачи будет последовательное применение алгоритма Дейкстры для каждой из возможных n начальных вершин. Но лучше будет использовать алгоритм Флойда-Варшалла, чтобы создать матрицу расстояний размером $n \times n$ из исходной матрицы весов графа.

Алгоритм Флойда лучше всего использовать на матрицах смежности. В этом нет ничего необычного, т. к. нам требуется сохранить все n попарных расстояний в любом случае. В листинге 6.11 приводится определение структуры для хранения требуемой информации.

Листинг 6.11. Определение типа `adjacency_matrix`

```
typedef struct {
   int weight[MAXV+1][MAXV+1];      /* Информация о смежности/весе */
   int nvertices;                   /* Количество вершин в графе */
} adjacency_matrix;
```

Под тип `adjacency_matrix` выделяется память для матрицы максимально большого размера. Кроме того, в нем предусмотрено поле для хранения информации о количестве вершин в графе.

Критическим вопросом в реализации матрицы смежности является способ обозначения отсутствующих в графе ребер. Обычно для обозначения присутствующих ребер невзвешенных графов используется 1, а отсутствующих — 0. Если эти числа обозначают вес ребер, то такое обозначение дает неверный результат, т. к. отсутствующие ребра рассматриваются как "бесплатный" путь между вершинами. Вместо этого отсутствующие ребра нужно инициализировать значением `MAXINT`. Таким образом мы можем и проверять наличие ребра, и автоматически игнорировать его при поиске кратчайшего пути, т. к. для этого будут рассматриваться только действительные ребра, при условии, что значение `MAXINT` меньше, чем диаметр графа.

Кратчайший путь между двумя вершинами графа можно определить несколькими способами. Алгоритм Флойда-Варшалла начинает работу с присвоения вершинам графа номеров от 1 до n. Эти номера используются не для маркировки вершин, а для их упорядочивания. Определим $W[i,j]^k$ как длину кратчайшего пути от вершины i к вершине j, в котором используются в качестве промежуточных вершин только вершины, пронумерованные 1, 2, ..., k.

Что это означает? Когда $k = 0$, то промежуточные вершины недопустимы, поэтому единственными разрешенными путями являются первоначальные ребра графа. Таким образом, матрица кратчайших путей для всех пар вершин состоит из первоначальной матрицы смежности. Будет выполнено n циклов, где в k-м цикле разрешается использовать только первые k вершин в качестве возможных промежуточных вершин на пути между каждой парой вершин x и y.

В каждом последующем цикле добавляется одна новая возможная промежуточная вершина, что расширяет набор возможных кратчайших путей. Использование k-й вершины в качестве конечной помогает только в том случае, если существует кратчайший путь, который проходит через эту вершину, поэтому

$$W[i,j]^k = \min(W[i,j]^{k-1}, W[i,k]^{k-1} + W[k,j]^{k-1}).$$

Правильность этого выражения не совсем очевидна с первого взгляда, поэтому я рекомендую хорошо разобраться в нем, чтобы убедиться в этом. Но, как видно из листинге 6.12, реализация алгоритма достаточно проста.

Листинг 6.12. Реализация алгоритма Флода-Варшалла

```
floyd(adjacency_matrix *g)
{
int i,j;            /* Счетчики измерений */
int k;              /* Счетчик промежуточных вершин */
int through_k;      /* Длина пути через вершину k */
for (k=11; k<=g->nvertices; k++)
   for (i=1; i<=g->nvertices; i++)
      for (j=1; j<=g->nvertices; j++) {
         through_k = g->weight[i][k]+g->weight[k] [j];
         if (through_k < g->weight[i][j])
            g->weight[i]{j] = through_k;
      }
}
```

Время исполнения алгоритма Флойда-Варшалла равно $O(n^3)$, что ничем не лучше, чем время исполнения n вызовов алгоритма Дейкстры. Но циклы этого алгоритма такие сжатые, и сама программа такая короткая, что на практике он оказывается более эффективным. Алгоритм Флойда-Варшалла примечателен тем, что это один из немногих алгоритмов работы с графами, которые работают лучше на матрицах смежности, чем на списках смежности.

По результату работы алгоритма Флойда-Варшалла в его текущей форме нельзя воссоздать фактический кратчайший путь между любой парой вершин. Но эти пути можно получить, если мы сохраним матрицу родителей P для нашего выбора последней промежуточной вершины, используемую для каждой пары вершин (x, y). Пусть это будет вершина k. Кратчайшим путем от вершины x к вершине y будет конкатенация кратчайшего пути от вершины x к вершине k и кратчайшего пути от вершины k к вершине y, которую можно воссоздать рекурсивным способом по матрице P. Но обратите внимание, что в большинстве задач поиска кратчайшего пути между всеми парами точек требуется только матрица расстояний. Для таких приложений и предназначен алгоритм Флойда-Варшалла.

6.3.3. Транзитивное замыкание

Алгоритм Флойда-Варшалла имеет еще одну важную область применения — вычисление *транзитивного замыкания* (transitive closure). При анализе орграфа часто требуется знать, какие вершины достижимы из данного узла.

В качестве примера рассмотрим граф шантажиста, в котором наличие ориентированного ребра (i, j) означает, что лицо i обладает достаточным компроматом на лицо j и может заставить его сделать все, что угодно. Для участия в специальном проекте вы хотите нанять одного из этих n людей.

Простым решением было бы нанять шантажиста, представленного вершиной с наибольшим весом, но лучшим выбором будет шантажист с компроматом на наибольшее число людей, т. е., вершина со связями с наибольшим количеством других вершин. У Стива может быть мощный компромат, но только на Майкла, но если Майкл имеет компромат на всех прочих, тогда именно он будет лучшим выбором для нашего проекта.

Вершины, достижимые из любой другой вершины, можно найти посредством обхода в ширину или глубину. Но все варианты можно вычислить с помощью алгоритма кратчайшего пути между всеми парами вершин. Если после исполнения алгоритма Флойда-Варшалла длина кратчайшего пути между вершиной *i* и вершиной *j* равняется `MAXINT`, то можно быть уверенным, что между этими вершинами прямого пути не существует. Любые две вершины, у которых вес ребра меньше `MAXINT`, достижимы, как и в смысле теории графов, так и в смысле описанного проекта.

Транзитивное замыкание рассматривается более подробно в *разделе 15.5*.

6.4. История из жизни. Печатаем с помощью номеронабирателя

В составе группы специалистов я был на экскурсии в компании Periphonics, которая в то время была лидером в области создания телефонных систем речевого взаимодействия. Это более интеллектуальные системы, чем системы типа "Нажмите кнопку 1 для перехода в меню", "Нажмите кнопку 2, если вы не нажали 1", которые так раздражают нас в повседневной жизни. Гид произносил стандартный текст о продукции компании, когда один из членов группы спросил: "Почему бы вам не использовать для ввода данных систему распознавания голоса? Это было бы намного проще, чем вводить текст с помощью номеронабирателя".

Реакция гида была отработана. — Нашим клиентам предоставляется опция включения возможности распознавания голоса в наш продукт, но очень немногие из них делают это. Системы распознавания слитной речи для общего класса пользователей недостаточно точны в большинстве приложений. Наши клиенты предпочитают создавать системы на основе ввода текста с клавиатуры телефона.

— Предпочитают вводить текст с помощью номеронабирателя? Не смешите меня! — раздался голос из задних рядов группы. — Я *ненавижу* такой способ ввода текста. Всегда, когда я звоню в свою маклерскую контору, чтобы узнать биржевой курс, какая-то машина велит мне ввести трехбуквенный код. Что еще хуже, чтобы указать, какую из трех букв, показанных на кнопке, ввести, нужно нажать две кнопки. Я нажимаю кнопку 2, а машина мне говорит: "Нажмите 1 для A, Нажмите 2 для B, Нажмите 3 для C". Это ужасно, если вас интересует мое мнение.

— Возможно, не нужно нажимать две кнопки, чтобы ввести одну букву, — подключился я к разговору. — Может быть, система сумеет вычислить требуемую букву по контексту.

— Когда вводишь трехбуквенный биржевой код акции, контекста маловато.

— Конечно, но если вводить предложения, то контекста будет предостаточно. Спорю, что мы смогли бы правильно восстановить текст, вводимый с панели телефона по одной кнопке на букву.

Сотрудник Periphonics бросил на меня равнодушный взгляд и продолжил экскурсию. Но мне эта идея запала в голову, и когда я вернулся в свой офис, я решил попытаться воплотить ее в жизнь.

При вводе с номеронабирателя не все буквы можно ввести одинаковым способом. Более того, не все буквы можно даже ввести, т. к. на стандартном американском телефоне буквы Q и Z не обозначены. Поэтому мы условились, что буквы Q, Z и пробел будут на кнопке звездочки (*). Для распознания вводимого с номеронабирателя текста можно было бы воспользоваться частотой использования букв алфавита. Например, если нажата кнопка 3, то существует большая вероятность, что имеется в виду буква E, а не находящиеся на этой же кнопке буквы D или F. Таким образом, учитывая частоту использования каждой буквы в окне из трех букв (триграмме), можно было попытаться предсказать вводимый текст. Вот что получилось, когда я испытал этот метод вводом Геттисбергского послания:

> enurraore ane reten yeasr ain our ectherr arotght eosti on ugis aootinent a oey oation aoncdivee in licesty ane eedicatee un uhe rrorosition uiat all oen are arectee e ual
>
> ony ye are enichde in a irect aitil yar uestini yhethes uiat oatioo or aoy oation ro aoncdivee ane ro eedicatee aan loni eneure ye are oet on a irect aattlediele oe uiat yar ye iate aone un eedicate a rostion oe uiat eiele ar a einal restini rlace eor uiore yin iere iate uhdis lives uiat uhe oation ogght live it is aluniethes eittini ane rrores uiat ye rioule en ugir
>
> att in a laries reore ye aan oou eedicate ye aan oou aoorearate ye aan oou ialloy ugis iroune the arate oen litini ane eeae yin rustgilee iere iate aoorearatee it ear aante our roor rowes un ade or eeuraat the yople yill little oote oor loni renences yiat ye ray iere att it aan oetes eosiet yiat uhfy eie iere it is eor ur uhe litini rathes un ae eedicatee iere un uhe undiniside yopl yhici uhfy yin entght iere iate uiur ear ro onaky aetancde it is rathes eor ur un ae iere eedicatee un uhe irect uarl rencinini adeore ur uiat eron uhere ioooree eeae ye uale inarearee eeuotion uo tiat aaure eor yhici uhfy iere iate uhe lart eull oearure oe eeuotioo tiat ye iere iggily rerolue uiat uhere eeae riall oou iate eide io

Как можно видеть, метод триграмм оказался довольно хорошим для шифрования текста, но совсем непригодным для его воспроизведения. Причина этому очевидная — алгоритм не знает абсолютно ничего об английских словах. Если применить его совместно со словарем, то мы, возможно, получим какой-то результат. Но в словаре может быть несколько разных слов для одной и той же последовательности кнопок номеронабирателя. В качестве крайнего примера последовательность 22737 может означать одиннадцать разных английских слов, включая *cases*, *cares*, *cards*, *capes*, *caper* и *bases*. В нашей следующей попытке для последовательностей кнопок с несколькими возможными словами мы выводили буквы, в правильности которых не было сомнений, а остальные угадывали методом триграмм. Наградой за это был следующий вывод:

> eourscore and seven yearr ain our eatherr brought forth on this continent azoey nation conceivee in liberty and dedicatee uo uhe proposition that all men are createe equal
>
> ony ye are engagee in azipeat civil yar uestioi whether that nation or aoy nation ro conceivee and ro dedicatee aan long endure ye are oet on azipeat battlefield oe that yar ye iate aone uo dedicate a rostion oe that field ar a final perthni place for those yin here iate their lives that uhe nation oight live it is altogether fittinizane proper that ye should en this
>
> aut in a larges sense ye aan oou dedicate ye aan oou consecrate ye aan oou hallow this ground the arate men litioi and deae yin strugglee here iate consecratee it ear above our roor power uo ade or detract the world will little oote oor long remember what ye ray here aut it aan meter forget what uhfy die here it is for ur uhe litioi rather uo ae dedicatee here uo uhe toeioisgee york which uhfy yin fought here iate thus ear ro mocky advancee it is rather for ur uo ae here dedicatee uo uhe great task renagogoi adfore ur that from there honoree deae ye uale increasee devotion uo that aause for which uhfy here iate uhe last eull measure oe devotion that ye here highky resolve that

there deae shall oou iate fide io vain that this nation under ioe shall iate azoey birth oe freedom and that ioternmenu oe uhe people ay uhe people for uhe people shall oou perish from uhe earth

Кто-либо, изучающий американскую историю, возможно и распознал бы этот текст, но для всех других он был бессмысленным. Нам нужно было найти какой-то способ различать слова, набираемые одинаковой последовательностью кнопок. Можно было бы попробовать учитывать относительную частоту использования каждого слова, но здесь также было бы слишком много ошибок.

На этом этапе я подключил к проекту Гаральда Pay (Harald Rau), который оказался прекрасным помощником. Прежде всего, он был смышленый и упорный аспирант. Кроме этого, т. к. его родным языком был немецкий, он верил всему, что я говорил ему об английской грамматике.

Гаральд создал алгоритм реконструкции слов по нажатым клавишам (рис. 6.9).

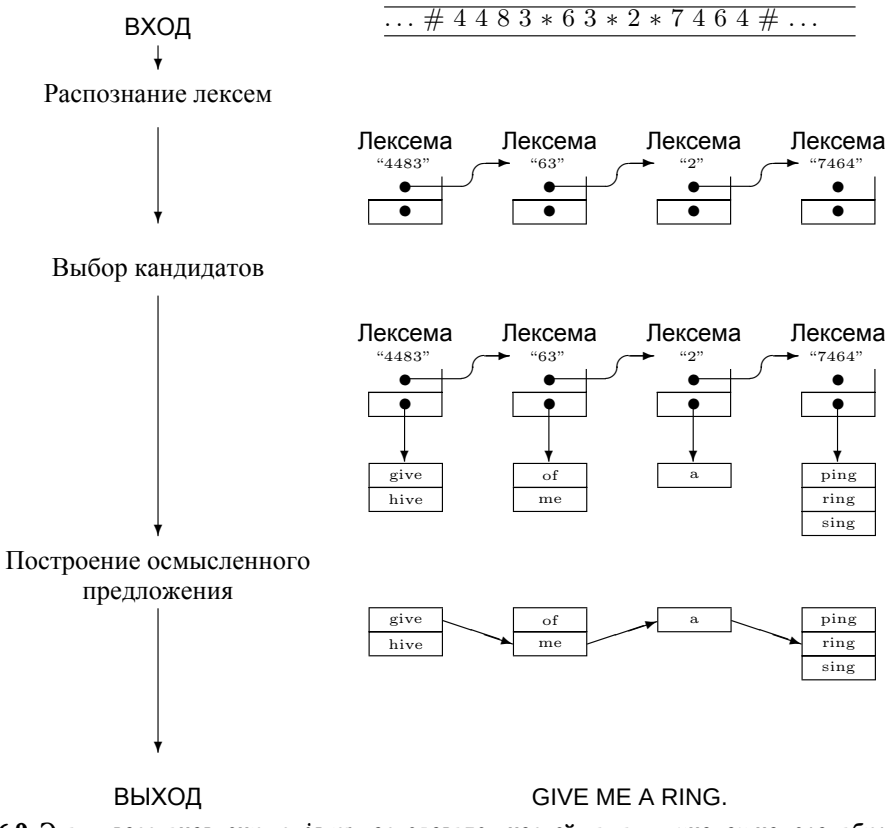


Рис. 6.9. Этапы восстановления слов из последовательностей нажатых кнопок номеронабирателя

Программа обрабатывала предложение, идентифицируя все слова, совпадающие с каждой последовательностью введенного кода. Принципиальную трудность представляло встраивание в нее грамматических условий.

— Хорошая информация о грамматике и о частоте использования слов имеется в большой базе данных, называющейся Brown Corpus. Она содержит тысячи типичных

английских предложений, проанализированных по частям речи. Но как мы можем учесть всю эту информацию в нашей программе? — спросил Гаральд.

— Давай будем рассматривать это, как задачу на графах, — предложил я.

— *Задачу на графах?* Где же здесь графы?

— Предложение можно рассматривать как последовательность лексем, каждая из которых представляет слово в предложении. Для каждой лексемы имеется соответствующий список слов, из которого нужно выбрать правильное. Как это сделать? Каждое возможное предложение, т. е. комбинацию слов, можно рассматривать как путь в графе, вершины которого — полный набор возможных слов. Каждое возможное *i*-е слово будет соединено ребром с каждым возможным (*i* + l)-м словом. Самым дешевым путем по этому графу будет наилучшая интерпретация предложения.

— Но все пути выглядят одинаково; они имеют одинаковое количество ребер. Погоди, теперь я вижу! Чтобы сделать пути разными, нам нужно присвоить ребрам вес.

— Совершенно верно! Стоимость ребра будет отражать вероятность нашего выбора соединяемой им пары слов. Эту стоимость можно определять по тому, как часто данная пара слов встречается в базе данных. Или вес можно присваивать с учетом частей речи. Возможно, существительные реже соседствуют с другими существительными, чем с глаголами.

— Отслеживать статистику пар слов будет трудно, поскольку их так много! Но мы знаем частоту использования каждого слова. Как мы можем учесть этот фактор в нашей программе?

— Можно сделать так, что цена за проход через определенную вершину будет зависеть от частоты употребления данного слова. В таком случае самый короткий путь через граф будет самым лучшим предложением.

— Но как нам определить относительную важность каждого из этих факторов?

— Сначала попробуй реализовать то, что тебе кажется естественным, а потом можно экспериментировать.

Алгоритм поиска кратчайшего пути, разработанный Гаральдом по этим принципам, показан на рис. 6.10.

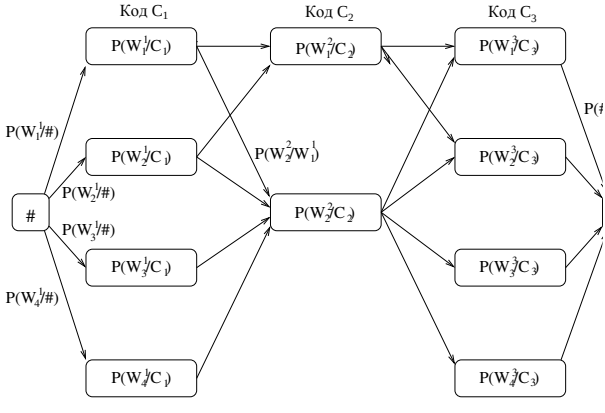


Рис. 6.10. Путь с наименьшей стоимостью представляет самую лучшую интерпретацию предложения

Программа со встроенными грамматическими и статистическими условиями работала прекрасно. Оцените ее воспроизведение ввода Геттисбергского послания с номеронабирателя телефона теперь:

> ENGAGED IN A GREAT CIVIL WAR TESTING WHETHER THAT NATION OR ANY NATION SO CONCEIVED AND SO DEDICATED CAN LONG ENDURE. WE ARE MET ON A GREAT BATTLEFIELD OF THAT **WAS**. WE HAVE COME TO DEDICATE A PORTION OF THAT FIELD AS A FINAL **SERVING** PLACE FOR THOSE WHO HERE **HAVE** THEIR LIVES THAT THE NATION MIGHT LIVE. IT IS ALTOGETHER FITTING AND PROPER THAT WE SHOULD DO THIS. BUT IN A LARGER SENSE WE CAN NOT DEDICATE WE CAN NOT CONSECRATE WE CAN NOT HALLOW THIS GROUND. THE BRAVE MEN LIVING AND DEAD WHO STRUGGLED HERE HAVE CONSECRATED IT FAR ABOVE OUR POOR POWER TO ADD OR DETRACT. THE WORLD WILL LITTLE NOTE NOR LONG REMEMBER WHAT WE SAY HERE BUT IT CAN NEVER FORGET WHAT THEY DID HERE. IT IS FOR US THE LIVING RATHER TO BE DEDICATED HERE TO THE UNFINISHED WORK WHICH THEY WHO FOUGHT HERE HAVE THUS FAR SO NOBLY ADVANCED. IT IS RATHER FOR US TO BE HERE DEDICATED TO THE GREAT TASK REMAINING BEFORE US THAT FROM THESE HONORED DEAD WE TAKE INCREASED DEVOTION TO THAT CAUSE FOR WHICH THEY HERE **HAVE** THE LAST FULL MEASURE OF DEVOTION THAT WE HERE HIGHLY RESOLVE THAT THESE DEAD SHALL NOT HAVE DIED IN VAIN THAT THIS NATION UNDER GOD SHALL HAVE A NEW BIRTH OF FREEDOM AND THAT GOVERNMENT OF THE PEOPLE BY THE PEOPLE FOR THE PEOPLE SHALL NOT PERISH FROM THE EARTH.

В то время как в выводе имеется несколько ошибок, результат, несомненно, является достаточно хорошим для многих приложений. В компании Periphonics определенно были такого мнения, т. к. они лицензировали нашу программу для использования в своих продуктах. В табл. 6.1 можно видеть, что мы смогли правильно воссоздать свыше 99% символов из почти мегабайта, занимаемого речами президента Клинтона. Так что, если бы он отправлял их в виде SMS-сообщений, вводя текст с номеронабирателя, то мы определенно были бы в состоянии понять их.

Таблица 6.1. Реконструкция нескольких текстов, вводимых с помощью кнопочного номеронабирателя телефона

Текст	Количество символов	Количество правильных символов	Количество правильных непробельных символов	Количество правильных слов	Время обработки символа
Речи Клинтона	1 073 593	99,04%	98,86%	97,67%	0,97 мс
Роман "Herland"	278 670	98,24%	97,89%	97,02%	0,97 мс
Роман "Моби Дик"	1 123 581	96,85%	96,25%	94,75%	1,14 мс
Библия	3 961 684	96,20%	95,39%	95,39%	1,33 мс
Пьесы Шекспира	4 558 202	95,20%	94,21%	92,86%	0,99 мс

Скорость воссоздания символов достаточно высокая, более того, быстрее, чем их ввод с клавиатуры номеронабирателя.

Условия для многих задач распознавания закономерностей можно сформулировать естественным образом в виде задач поиска кратчайшего пути в графе. Для решения

этих задач хорошо подходит алгоритм Витерби, который широко применяется в системах распознавания речи и рукописи. Алгоритм Витерби, по сути, решает задачу поиска кратчайшего пути в бесконтурном орграфе. Таким образом, часто бывает полезно попытаться сформулировать поставленную задачу в терминах теории графов.

6.5. Потоки в сетях и паросочетание в двудольных графах

Графы со взвешенными ребрами можно рассматривать как трубопроводную сеть, в которой вес ребра (i, j) определяет пропускную способность каждого отрезка трубопровода. В свою очередь, пропускную способность можно рассматривать как функцию поперечного сечения трубы. Широкая труба может пропускать 10 единиц потока за определенное время, в то время как более узкая труба — только 5 единиц за то же самое время. В *задаче о потоках в сетях* требуется определить максимальный объем потока, который можно пропустить между вершинами s и t взвешенного графа G, соблюдая ограничения на максимальную пропускную способность каждого ребра-трубы.

6.5.1. Паросочетание в двудольном графе

В то время как задача о потоках в сетях представляет самостоятельный интерес, ее основная ценность состоит в решении других важных задач на графах. Классическим примером является паросочетание в двудольном графе. Паросочетанием графа $G = (V, E)$ называется такое подмножество ребер $E' \subset E$, в котором никакие два ребра не имеют общую вершину. При этом вершины группируются попарно таким образом, что каждая вершина принадлежит не более, чем одной такой паре. На рис. 6.11 показаны двудольный граф с максимальным паросочетанием (выделено жирными линиями) и соответствующий экземпляр потоков в сети с максимальным потоком s–t.

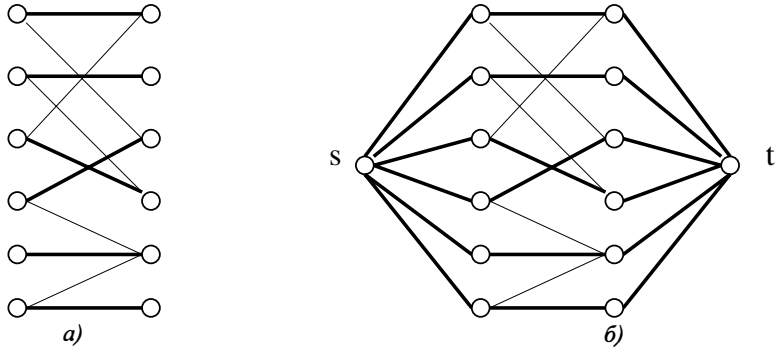


Рис. 6.11. Двудольный граф с максимальным паросочетанием *(а)* и экземпляр потоков в сети *(б)*

Граф G называется двудольным, если его вершины можно разделить на два множества, L и R, таким образом, что все ребра графа имеют одну вершину в множестве L, а другую — в множестве R. Многие естественные графы являются двудольными. Например, некоторые вершины могут представлять задания, требующие исполнения, а остальные

вершины — людей, которые могут выполнить эти задания. Наличие ребра (j, p) означает, что задание j может быть выполнено человеком p. Или же часть вершин может представлять юношей, а другая девушек, а ребра — совместимые гетеросексуальные пары. Паросочетания в этих графах подаются естественному толкованию как рабочее задание или женитьба. Такие паросочетания подробно обсуждаются в *разделе 15.6*.

Максимальное парообразование в двудольном графе можно с легкостью найти, используя потоки в сети. Для этого создаем вершину-*исток* s, которая соединена со всеми другими вершинами в подмножестве L взвешенными ребрами, каждое весом 1. Потом создаем вершину-*сток* t, которая соединена со всеми другими вершинами в подмножестве R взвешенными ребрами, каждое весом 1. Наконец, присваиваем каждому ребру двудольного графа G вес 1. Теперь максимальный возможный поток из вершины s в вершину t определяет максимальное паросочетание в графе G. Всегда можно найти поток такого же объема, как и паросочетание, используя только ребра паросочетания и соединенные ими истоки и стоки. Кроме этого, поток большего объема невозможен. Ведь не можем же мы пропускать через какую-либо вершину больше, чем одну единицу потока.

6.5.2. Вычисление потоков в сети

Традиционные алгоритмы работы с потоками в сети основаны на идее *увеличивающих путей*, которая состоит в многократном повторении процесса поиска пути с положительной пропускной способностью от вершины s к вершине t и добавления его к потоку. Можно доказать, что поток в сети является оптимальным тогда и только тогда, когда в ней не имеется увеличивающего пути. Так как каждое добавление пути увеличивает поток, то в итоге должен быть найден глобальный максимум.

Ключевой структурой является *граф остаточного потока* (residual flow graph) $R(G, f)$, где G — граф ввода, а f — текущий поток через G. Ориентированный взвешенный граф $R(G, f)$ содержит те же самые вершины, что и граф G. Для каждого ребра (i, j) в графе G, которое обладает пропускной способностью $c(i, j)$ и имеет поток $f(i, j)$, граф $R(G, f)$ может содержать следующие два ребра:

- ребро (i, j) с весом $c(i, j) - f(i, j)$, если $c(i, j) - f(i, j) > 0$;
- ребро (j, i) с весом $f(i, j)$, если $f(i, j) > 0$.

Наличие ребра (i, j) в графе остаточного потока указывает на то, что от вершины i к вершине j можно направить положительный поток, точный объем которого определяется весом ребра. Путь в графе остаточного потока от вершины s к вершине t подразумевает, что от первой вершины ко второй можно пропустить дополнительный поток, объем которого определяется минимальным весом ребра на этом пути. Для иллюстрации этой идеи на рис. 6.12 изображены максимальный поток s–t в графе G и связный граф остаточного потока $R(G)$. Минимальный разрез потока s–t обозначен пунктирной линией возле вершины t.

Максимальный поток s–t в графе G равен 7. Этот поток определяется двумя путями в графе остаточного потока $R(G)$, направленными от вершины t к вершине s и пропускной способностью 2 + 5. Эти потоки полностью используют пропускную способность двух ребер, входящих в вершину t, вследствие чего не остается путей, способных что-

либо добавить. Таким образом, поток является оптимальным. Множество ребер, удаление которых отделяет вершину *s* от вершины *t* (как два ребра, входящие в вершину *t* на рис. 6.12), называется (*s–t*)-*разрезом* (cut). Очевидно, что никакой поток из вершины *s* в вершину *t* не может превзойти вес такого минимального разреза. С другой стороны, всегда возможен поток, равный минимальному разрезу.

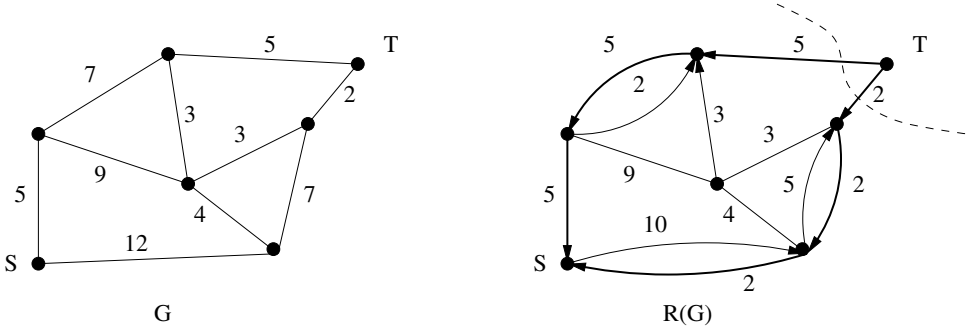


Рис. 6.12. Максимальный поток *s–t* в графе *G* и связный граф остаточного потока *R*(*G*)

Подведение итогов

Максимальный поток из *s* в *t* всегда равен минимальному весу (*s–t*)-разреза. Таким образом, алгоритмы для работы с потоками можно применять для решения общих задач связности ребер и вершин графов.

Реализация

В этой книге мы не можем представить теорию потоков сети в полном объеме. Но мы сможем, по крайней мере, показать, как определять увеличивающие пути и вычислять оптимальный поток. Для каждого ребра в графе остаточного потока необходимо отслеживать как текущий объем потока, проходящего через него, так и его *остаточную* пропускную способность. Соответственно, необходимо модифицировать структуру ребра, чтобы учесть дополнительные поля (листинг 6.13).

Листинг 6.13. Модифицированная структура ребра

```
typedef struct {
   int v;                  /* Соседняя вершина */
   int capacity;           /* Пропускная способность ребра */
   int flow;               /* Поток через ребро */
   int residual;           /* Остаточная пропускная способность ребра */
   struct edgenode *next; /* следующее ребро в списке */
} edgenode;
```

Используя обход в ширину, мы выполняем поиск путей от истока к стоку, которые увеличивают общий поток, и добавляем обнаруженные пути, увеличивая общий поток. Когда все увеличивающие пути обнаружены и добавлены в поток, процедура завершается, возвращая оптимальный поток (листинг 6.14).

Листинг 6.14. Процедура поиска оптимального потока

```
netflow(flow_graph *g, int source, int sink)
{
   int volume;                    /* Вес увеличивающего пути*/
   add_residual_edges(g);
   initialize_search(g);
   bfs(g,source);
   volume = path_volume(g, source, sink, parent);
   while (volume > 0) {
      augment_path(g, source, sink, parent, volume);
      initialize_search(g);
      bfs(g,source);
      volume = path_volume(g, source, sink, parent);
   }
}
```

Увеличивающий путь от истока к стоку повышает объем потока; такой путь можно найти посредством обхода в ширину соответствующего графа. Рассматриваются только такие ребра, которые имеют остаточную пропускную способность, или, иными словами, положительный остаточный поток. При обходе в ширину насыщенные и ненасыщенные ребра различаются с помощью процедуры, возвращающей булево значение (листинг 6.15).

Листинг 6.15. Процедура для различения насыщенных и ненасыщенных ребер

```
bool valid_edge(edgenode *e)
{
   if (e->residual > 0) return (TRUE);
   else return(FALSE);
}
```

При увеличении пути максимально возможный объем потока перемещается из ребер с остаточной пропускной способностью в положительный поток. Этот объем ограничен ребром с наименьшей пропускной способностью, точно так же, как и объем пропускаемой по водопроводу воды ограничен наиболее узкой трубой. Соответствующий код показан в листинге 6.16.

Листинг 6.16. Добавление увеличивающих путей в поток

```
int path_volume(flow_graph *g, int start, int end, int parents[])
{
   edgenode *e;                        /* Рассматриваемое ребро */
   edgenode *find_edge();
   if (parents[end] == -1) return(0);
   e = find_edge(g,parents[end],end);
   if (start == parents[end])
      return(e->residual);
   else
      return( min(path_volume(g,start,parents[end],parents),
         e->residual) );
}
```

```
edgenode *find_edge(flow_graph *g, int x, int y)
{
   edgenode *p;  /* Временный указатель */
   p = g->edges[x];
   while (p != NULL) {
      if (p->v == y) return(p);
      p = p->next;
   }
   return(NULL);
}
```

Направление дополнительного объема потока по ориентированному ребру (i, j) уменьшает остаточную пропускную способность этого ребра, но *увеличивает* пропускную способность ребра (j, i). Таким образом, действие увеличения пути требует модифицирования как прямых, так и обратных ребер для каждого звена пути. Соответствующий код показан в листинге 6.17.

Листинг 6.17. Модификация ребер

```
augment_path(flow_graph *g, int start, int end,
             int parents[], int volume)
{
   edgenode *e;              /* Рассматриваемое ребро */
   edgenode *find_edge();
   if (start == end) return;
   e = find_edge(g, parents[end],end);
   e->flow += volume;
   e->residual -= volume;
   e = find_edge(g, end, parents[end]);
   e->residual += volume;
   augment_path(g, start, parents[end], parents, volume);
}
```

Для инициализации графа потока требуется создать ориентированные ребра (i, j) и (j, i) для каждого ребра сети $e = (i, j)$. Все начальные потоки инициализируются в 0. Первоначальный остаточный поток ребра (i, j) устанавливается равным пропускной способности ребра сети e, а первоначальный остаточный поток ребра (j, i) устанавливается равным нулю.

Анализ

Алгоритм поиска увеличивающих путей в конечном счете приходит к оптимальному решению. Но каждый новый увеличивающий путь может добавлять только незначительный объем к общему потоку, поэтому, теоретически, время схождения алгоритма может быть сколь угодно большим.

Но Эдмондс (Edmonds) и Карп (Karp) в книге [EK72] доказали, что постоянно выбирая *кратчайший* невзвешенный увеличивающий путь, можно гарантировать, что добавление $O(n^3)$ увеличивающих путей будет достаточным для получения оптимального потока. В действительности, наша реализация поиска оптимального потока и является

алгоритмом Эдмондса-Карпа, т. к. для поиска следующего увеличивающего пути используется обход в ширину, начинающийся с истока.

6.6. Разрабатывайте не алгоритмы, а графы

Правильное моделирование является ключом к эффективному использованию алгоритмов для работы с графами. Мы определили несколько свойств графов и разработали алгоритмы для их вычисления. В каталоге задач представлено около двух десятков разных задач по графам. Эти классические задачи по графам предоставляют основу для моделирования большинства приложений.

Но секрет успешного решения задач на графах заключается в умении разрабатывать не алгоритмы для работы с графами, а сами графы. Мы уже видели несколько примеров этой идеи. В частности:

- *максимальное* остовное дерево можно найти, изменив веса ребер исходного графа G на отрицательные и применив на получившемся графе G' алгоритм для построения *минимального* остовного дерева. Остовное дерево графа G' с максимальным по модулю общим весом будет максимальным остовным деревом графа G;
- для решения задачи паросочетания в двудольном графе мы создали специальный граф потоков в сети, в котором максимальный поток соответствует паросочетанию максимальной мощности.

Рассматриваемые далее примеры дополнительно демонстрируют мощь правильного моделирования. Каждый из этих примеров возник в реальном приложении и каждый можно смоделировать в виде задачи на графах. Некоторые примеры довольно сложны, но они иллюстрируют универсальность графов в представлении взаимоотношений. Прочитав задачу, не спешите заглядывать на страницу с решением, а попытайтесь создать для нее соответствующее графовое представление.

Остановка для размышлений. Нить Ариадны

ЗАДАЧА. Требуется алгоритм для разработки естественных маршрутов, по которым персонажи видеоигр могут проходить через помещения, наполненные разными препятствиями.

Решение. Предположительно, искомый маршрут должен соответствовать маршруту, который бы выбрало разумное существо. А поскольку разумные существа склонны к лени и оптимальным действиям, они выберут самый короткий маршрут. Соответственно, данную задачу нужно моделировать как задачу поиска кратчайшего пути.

Но что у нас будет служить графом? Один из подходов может заключаться в наложении решетки на комнату. Для каждого узла решетки, свободного от предметов, создаем вершину. Между любыми двумя близлежащими вершинами обязательно найдется ребро, взвешенное пропорционально расстоянию между ними. Хотя для поиска кратчайшего пути существуют прямые геометрические методы (см. *раздел 15.4*), данную задачу легче смоделировать дискретно в виде графа. ■

Остановка для размышлений. Упорядочивание последовательности

ЗАДАЧА. Проект секвенирования ДНК имеет на выходе экспериментальные данные, состоящие из небольших фрагментов. Для каждого фрагмента f известно, что некоторые фрагменты обязательно расположены слева от него, а другие — справа. Нужно найти непротиворечивое упорядочивание фрагментов слева направо, которое удовлетворяет всем требованиям.

Решение. Создаем ориентированный граф, в котором каждый фрагмент представлен вершиной. Вставляем ориентированное ребро $(\overrightarrow{l,f})$ от любого фрагмента l, который должен быть слева от фрагмента f, и ориентированное ребро $(\overrightarrow{f,r})$ от любого фрагмента r, который должен быть справа от фрагмента f. Нам нужно упорядочить вершины таким образом, чтобы все ребра были направлены слева направо. Это будет *топологическая сортировка* получившегося бесконтурного орграфа. Граф должен быть бесконтурным, т. к. при наличии контуров (т. е. замкнутых маршрутов) непротиворечивое упорядочивание невозможно. ■

Остановка для размышлений. Размещение прямоугольников по корзинам

ЗАДАЧА. Произвольное множество прямоугольников в плоскости нужно распределить по минимальному количеству корзин таким образом, чтобы ни одно из подмножеств прямоугольников в любой корзине не пересекалось с другим. Иными словами, в одной и той же корзине не может быть двух перекрывающихся прямоугольников.

Решение. Создаем граф, в котором каждая вершина представляет прямоугольник, а перекрывающиеся прямоугольники соединяются ребром. Каждая корзина соответствует *независимому множеству* прямоугольников, поэтому ни один из них не накладывается на другой. *Раскраской вершин* графа называется разбиение вершин на независимые множества, поэтому для нашей задачи требуется минимизировать количество цветов. ■

Остановка для размышлений. Конфликт имен файлов

ЗАДАЧА. При переносе кода из операционной системы UNIX в DOS нужно сократить имена нескольких сотен файлов до, самое большее, 8 символов. Просто использовать первые восемь символов имени файла нельзя, т. к. *имя_файла1* и *имя_файла2* будут преобразованы в одно и то же имя. Как рационально сократить имена файлов и при этом избежать конфликтов между получившимися именами?

Решение. Создайте двудольный граф, в котором вершины соответствуют каждому первоначальному имени файла f_i для $1 \le i \le n$, а также набор приемлемых сокращений для каждого имени f_{i1}, ..., f_{ik}. Соедините ребром каждое первоначальное имя и его сокращенный вариант. Теперь нужно найти набор из n ребер, не имеющих общих вершин, соотнеся, таким образом, первоначальное имя файла с приемлемым индивидуализированным сокращением. Задача *паросочетаний в двудольном графе*, рассматриваемая в *разделе 15.6*, как раз является типом задачи поиска независимого множества ребер в графе. ■

Остановка для размышлений. Разделение текста

ЗАДАЧА. Найти способ для разделения строчек текста в разрабатываемой системе распознавания текста. Хотя между печатными строчками текста имеется свободный промежуток, но по разным причинам, таким как помехи или перекос страницы, этот интервал трудно выделить. Каким образом можно решить это задачу?

Решение. Примем следующее определение графа. Каждый пиксел изображения представляется вершиной графа, а соседние пикселы соединяются ребром. Вес этого ребра должен быть пропорционален степени черного цвета в пикселах. В этом графе интервал между двумя печатными строчками будет путем, направленным от левого края страницы к правому. Нам нужно найти относительно прямой путь, максимально избегающий черных точек. Предполагается, что кратчайший путь в графе пикселов будет с большой вероятностью правильным разделителем. ■

Подведение итогов

Разработать действительно новый алгоритм работы с графами очень нелегко, поэтому не тратьте на это время. Вместо этого для моделирования стоящей перед вами задачи старайтесь разработать графы, которые позволяют применить классические алгоритмы.

Замечания к главе

Представление задачи в форме потоков в сети является эффективным алгоритмическим инструментом, но для того, чтобы понять, можно ли решить данную задачу с помощью этого инструмента, требуется опыт. Для более подробного изучения данной темы рекомендуется ознакомиться с книгами [CC97] и [AMO93].

Метод увеличивающих путей описан Фордом (Ford) и Фулкерсоном (Fulkerson) в книге [FF62]. Эдмондс (Edmonds) и Карп (Karp) в книге [EK72] доказали, что постоянно выбирая кратчайший невзвешенный увеличивающий путь, можно гарантировать, что добавление $O(n^3)$ увеличивающих путей будет достаточным для получения оптимального потока.

Система распознавания текста, вводимого с помощью кнопочного номеронабирателя телефона, которая рассматривалась ранее в этой главе, описывается более подробно в книге [RS96].

6.7. Упражнения

Алгоритмы для эмуляции графов

1. [3] Для графов из задачи 1 *главы 5*:

   а) Нарисуйте остовный лес, получаемый после каждой итерации основного цикла алгоритма Крускала.

   б) Нарисуйте остовный лес, получаемый после каждой итерации основного цикла алгоритма Прима.

   в) Найдите остовное дерево с кратчайшим путем и с корнем в вершине A.

   г) Вычислите максимальный поток от вершины A к вершине H.

Минимальные остовные деревья

2. [3] Будет ли путь между двумя вершинами в минимальном остовном дереве обязательно самим коротким путем между этими двумя вершинами в полном графе? Если да, предоставьте доказательство; если нет, приведите контрпример.

3. [3] Допустим, что все ребра графа имеют разный вес, т. е. нет ни одной пары ребер с одинаковым весом. Будет ли путь между двумя вершинами в минимальном остовном дереве обязательно самым коротким путем между этими двумя вершинами в полном графе? Если да, предоставьте доказательство; если нет, приведите контрпример.

4. [3] Могут ли алгоритмы Прима и Крускала выдавать разные минимальные остовные деревья? Аргументируйте свой ответ.

5. [3] Могут ли алгоритмы Прима и Крускала работать с графами, содержащими ребра с отрицательным весом? Аргументируйте свой ответ.

6. [5] Дано минимальное остовное дерево T графа G (содержащего n вершин и m ребер). К этому графу мы добавим новое ребро $e = (u, v)$ весом w. Разработайте эффективный алгоритм для построения минимального остовного дерева графа $G + e$. Чтобы решение было засчитано полностью, алгоритм должен иметь время исполнения $O(n)$.

7. [5] а) Дано минимальное остовное дерево T взвешенного графа G. Создайте новый граф G', увеличив вес каждого ребра графа G на k. Создают ли ребра минимального остовного дерева T графа G минимальное остовное дерево графа G'? Если да, предоставьте доказательство; если нет, приведите контрпример.

   б) Пусть $P = \{s, \ldots, t\}$ описывает кратчайший взвешенный путь между вершинами s и t взвешенного графа G. Создайте новый граф G', увеличив вес каждого ребра графа G на k. Описывает ли P кратчайший путь от вершины s к вершине t в графе G'? Если да, предоставьте доказательство; если нет, приведите контрпример.

8. [5] Разработайте и выполните анализ алгоритма, который во взвешенном графе G находит наименьшее изменение в стоимости ребра, не являющегося ребром минимального остовного дерева, вследствие которого изменится минимальное остовное дерево графа. Алгоритм должен быть корректным и иметь полиноминальное время исполнения.

9. [4] Рассмотрим задачу поиска взвешенного связного подмножества ребер T с минимальным весом во взвешенном связном графе G. Вес T состоит из суммы весов всех его ребер.

   а) Почему эта задача не является задачей поиска минимального остовного дерева? Подсказка: вспомните о ребрах с отрицательным весом.

   б) Разработайте эффективный алгоритм поиска связного подмножества T с минимальным весом.

10. [4] Дан неориентированный граф $G = (V, E)$. Множество ребер $F \subseteq E$ называется *разрывающим множеством ребер* (feedback-edge set), если каждый контур в графе G имеет, по крайней мере, одно ребро в F.

    а) Дан невзвешенный граф G. Разработайте эффективный алгоритм поиска разрывающего множества ребер минимального размера.

    б) Дан взвешенный неориентированный граф G с ребрами положительного веса. Разработайте эффективный алгоритм поиска разрывающего множества ребер минимального веса.

11. [5] Модифицируйте алгоритм Прима, чтобы он исполнялся за время $O(n\log k)$ для графа, у которого есть только k разных весов ребер.

Поиск-объединение

12. [5] Разработайте эффективную структуру данных для выполнения таких операций на взвешенных орграфах:

    а) слияние двух указанных компонентов;

    б) поиск компонента, содержащего указанную вершину v;

    в) возвращение минимального ребра из указанного компонента.

13. [5] Разработайте структуру данных, которая может поддерживать выполнение последовательности из m операций *объединение* и *поиск* на универсальном множестве из n элементов, такой, что вначале выполняются все операции объединения, а за ними — все операции поиска. Последовательность операций должна выполняться за время $O(m + n)$.

Поиск кратчайшего пути

14. [3] В задаче *о кратчайшем пути в пункт назначения* (single-destination shortest path) требуется найти кратчайший путь из каждой вершины графа в указанную вершину. Разработайте эффективный алгоритм для решения этой задачи.

15. [3] Дано: неориентированный взвешенный граф $G = (V, E)$ и его остовное дерево T с кратчайшим путем и с корнем в вершине v. Если увеличить вес всех ребер графа G на k, останется ли T кратчайшим остовным деревом с корнем в вершине v?

16. [3] а) Приведите пример взвешенного связного графа $G = (V, E)$ и вершины v, для которых минимальное остовное дерево и кратчайшее остовное дерево с корнем в вершине v являются одинаковыми.

    б) Приведите пример взвешенного связного ориентированного графа $G = (V, E)$ и вершины v, для которых минимальное остовное дерево существенно отличается от минимального остовного дерева с корнем в вершине v.

    в) Могут ли эти два типа остовных деревьев быть полностью непересекающимися?

17. [3] Докажите, что следующие утверждения верны, в противном случае приведите соответствующий контрпример.

    а) Будет ли путь между двумя вершинами в минимальном остовном дереве неориентированного графа кратчайшим (имеющим минимальный вес) путем?

    б) Допустим, что граф имеет единственное минимальное остовное дерево. Будет ли путь между двумя вершинами в минимальном остовном дереве неориентированного графа кратчайшим (имеющим минимальный вес)?

18. [5] В некоторых задачах на графах вес может присваиваться не ребрам (или не только ребрам), а вершинам. Пусть C_v означает вес вершины v, а $C(x, y)$ — вес ребра (x, y). Требуется найти самый дешевый путь между вершинами a и b графа G. Стоимость пути определяется как сумма весов ребер и вершин, через которые проходит путь.

    а) Все ребра графа имеют нулевой вес, а отсутствующие ребра обозначаются бесконечным весом (∞). Вес каждой вершины $C_v = 1$ ($1 \le v \le n$). Разработайте *эффективный* ал-

горитм поиска самого дешевого пути от вершины a к вершине b. Укажите временную сложность этого алгоритма.

б) Вершины имеют разный вес (но обязательно положительный), а вес ребер остается прежним. Разработайте эффективный алгоритм поиска самого дешевого пути от вершины a к вершине b. Укажите временную сложность этого алгоритма.

в) Вершины и ребра имеют разный вес (но обязательно положительный). Разработайте эффективный алгоритм поиска самого дешевого пути от вершины a к вершине b. Укажите временную сложность этого алгоритма.

19. [5] Дан ориентированный взвешенный граф G с n вершинами и m ребрами, в котором все вершины имеют положительный вес. *Контуром* (directed cycle) называется ориентированный путь, который начинается и заканчивается в одной и той же вершине и содержит, по крайней мере, одно ребро. Предоставьте алгоритм с временем исполнения $O(n^3)$ для поиска контура в графе с минимальным общим весом. Чтобы решение было зачтено частично, время исполнения алгоритма может быть равным $O(n^2m)$.

20. [5] Можно ли модифицировать алгоритм Дейкстры для решения задачи поиска самого длинного пути из заданного пункта выхода (single-source *longest* path), изменив *minimum* на *maximum*? Если да, предоставьте доказательство; если нет, приведите контрпример.

21. [5] Дан взвешенный бесконтурный орграф $G = (V, E)$, в котором, возможно, имеются ребра с отрицательным весом. Разработайте алгоритм с линейным временем исполнения для решения задачи поиска кратчайшего пути из заданной начальной вершины v.

22. [5] Дан взвешенный ориентированный граф $G = (V, E)$, в котором все веса положительные. Пусть v и w — вершины графа G, k — целое число ($k \leq |V|$). Разработайте алгоритм поиска кратчайшего пути от вершины v к вершине w, содержащего ровно k ребер. Путь не обязательно должен быть простым.

23. [5] *Арбитражем* называется использование разницы в курсах обмена валют для получения прибыли. Например, в течение короткого периода времени за 1 доллар США можно купить 0,75 фунта стерлингов, за 1 фунт стерлингов — 2 австралийских доллара, а за 1 австралийский доллар — 0,50 доллара США. То есть осуществление такой сделки принесет прибыль, равную $0.75 \times 2 \times 0.7 = 1.05$ доллара США или 5%. Имеется n валют c_1, ..., c_n и таблица курсов валют R размером $n \times n$, в которой указывается, что за одну единицу валюты c_i можно купить $R[i, j]$ единиц валюты c_j. Разработайте и выполните анализ алгоритма для определения максимального значения

$$R[c_1, c_{i1}] \cdot R[c_{i1}, c_{i2}] \cdot \cdots R[c_{ik+1}, c_{ik}] \cdot R[c_{ik}, c_1]$$

Подсказка: ищите кратчайший путь между всеми парами вершин.

Потоки в сети и паросочетание

24. [3] *Паросочетанием* в графе называется набор непересекающихся ребер, т. е. ребер, которые не имеют общих вершин. Разработайте алгоритм поиска максимального паросочетания в дереве.

25. [5] *Реберным покрытием* (edge cover) неориентированного графа $G = (V, E)$ называется набор ребер, для которого в каждую вершину графа входит, по крайней мере, одно ребро из этого набора. Разработайте эффективный алгоритм на основе паросочетаний для поиска реберного покрытия максимального размера графа G.

Задачи по программированию

Эти задачи доступны на сайтах **http://www.programming-challenges.com** и **http://uva.onlinejudge.org**.

1. Freckles. 111001/10034.
2. Necklace. 111002/10054.
3. Railroads. 111004/10039.
4. Tourist Guide. 111006/10199.
5. The Grand Dinner. 111007/10249.

ГЛАВА 7

Комбинаторный поиск и эвристические методы

Для многих задач можно найти решение с помощью методов исчерпывающего перебора, хотя временная сложность таких методов может оказаться астрономической. А в других ситуациях время, потраченное на поиск оптимального решения, обязательно окупится в дальнейшем. Правильность работы электрической схемы можно доказать, проверив все возможные входные контакты и удостоверившись в правильности выходного сигнала. Но такая проверка не всегда стоит затраченных на это усилий.

Частота тактового генератора современных компьютеров достигает нескольких гигагерц, что означает, что они могут исполнять несколько миллиардов базовых инструкций в секунду. Так как осуществление какой-либо представляющей интерес операции более высокого уровня требует нескольких сот базовых инструкций, то можно ожидать, что за секунду мы можем просмотреть несколько миллионов элементов.

Но важно иметь представление, насколько велик один миллион. Один миллион перестановок означает все возможные упорядочивания приблизительно 10 или 11 объектов, но не больше. Один миллион подмножеств означает все возможные комбинации около 20 элементов, но не более. Решение задач существенно большего размера требует тщательного сокращения пространства поиска, чтобы обработке подвергались только действительно имеющие важность элементы.

В этой главе представляется метод перебора с возвратом, применяющийся для перечисления всех возможных решений комбинаторной алгоритмической задачи. Также приводится иллюстрация хитроумных методов отсечения тупиковых решений для ускорения работы реальных поисковых приложений. Кроме этого, для решения задач, слишком больших для решения методом полного перебора всех комбинаций, рассматриваются эвристические методы, такие как имитация отжига. Такие эвристические методы являются важными инструментами в наборе любого практикующего алгориста.

7.1. Перебор с возвратом

Перебор с возвратом позволяет систематически исследовать все возможные конфигурации области поиска. Эти конфигурации могут представлять все возможные расположения объектов, т. е. перестановки, или все возможные наборы объектов, т. е. подмножества. В других ситуациях может потребоваться выполнить перечисление всех деревьев графа, всех путей между двумя вершинами или всех возможных способов группирования вершин по цветам.

Общий момент в этих задачах — то, что каждую возможную конфигурацию нужно сгенерировать ровно один раз. Запрет как на повторение, так и на пропуск конфигураций означает, что нам нужно определить четкий порядок их генерирования. Мы будем моделировать наше решение в виде вектора $a = (a_1, a_2, ..., a_n)$, в котором каждый эле-

мент a_i выбирается из конечного упорядоченного множества S_i. Такой вектор может представлять конфигурацию, в которой a_i содержит i-й элемент перестановки. Или же этот вектор может представлять заданное подмножество S, где a_i истинно тогда и только тогда, когда i-й элемент содержится в S. Этот же вектор может также представлять последовательность ходов в игре или путь в графе, где a_i содержит i-й элемент последовательности.

На каждом этапе алгоритма перебора с возвратом мы пытаемся расширить данное частичное решение $a = (a_1, a_2, ..., a_k)$, добавляя следующий элемент в конец последовательности. После расширения последовательности нам нужно проверить, не содержит ли она полного решения, и если содержит, то можем ли мы вывести или вычислить его. В противном случае нам нужно выяснить, существует ли возможность расширения частичного решения к полному решению.

При переборе с возвратом создается дерево, в котором каждая вершина представляет частичное решение. Если узел y создан в результате перехода от узла x, то эти узлы соединяются ребром. Такое дерево частичных решений предоставляет альтернативный взгляд на перебор с возвратом, т. к. процесс создания решений в точности соответствует процессу обхода в глубину дерева перебора с возвратом. Рассматривая перебор с возвратом как обход в глубину неявного графа, мы создаем естественную рекурсивную реализацию базового алгоритма, псевдокод которого показан в листинге 7.1.

Листинг 7.1. Перебор с возвратом

```
Backtrack-DFS(A, k)
   if A = (a1, a2, ..., ak) является решением, выводим его
   else
      k = k + 1
      compute Sk
      while Sk ≠ Ø do
         ak = an element in Sk
         Sk = Sk — ak
         Backtrack-DFS(A,k)
```

Хотя для перечисления решений можно было бы также применить и обход в ширину, обход в глубину намного предпочтительнее, т. к. он занимает меньше места. Текущее состояние поиска полностью представляется путем от корня к текущему узлу обхода в глубину. Требуемое для этого место пропорционально высоте дерева. А при обходе в ширину в очереди сохраняются все узлы текущего уровня, для чего нужно место, пропорциональное ширине дерева поиска. Для большинства представляющих интерес задач ширина дерева возрастает экспоненциально по отношению к его высоте.

Реализация

Код алгоритма перебора с возвратом показан в листинге 7.2.

Листинг 7.2. Реализация алгоритма перебора с возвратом

```
bool finished = FALSE;        /* Найдены все решения? */
backtrack(int a[], int k, data input)
```

```
{
   int c[MAXCANDIDATES];  /* Кандидаты для следующей позиции */
   int ncandidates;       /* Количество кандидатов на следующую позицию*/
   int i;                 /* Счетчик */
   if (is_a_solution(a, k, input))
      process_solution(a, k, input);
   else {
      k = k+1;
      construct_candidates(a, k, input, c, &ncandidates);
      for (i=0; i<ncandidates; i++) {
         a[k] = c[i];
         make _move(a, k, input);
         backtrack(a, k, input);
         unmake_move(a, k,input);
         if (finished) return;       /* Досрочное завершение*/
      }
   }
}
```

Метод перебора с возвратом обеспечивает правильность результата, перечисляя все возможные комбинации, а эффективность обеспечивается тем, что никакое состояние не исследуется более одного раза.

Изучите, как рекурсия позволяет создать легкую и элегантную реализацию алгоритма перебора с возвратом. Так как при каждом рекурсивном вызове создается новый массив кандидатов `c`, то подмножества еще не рассмотренных кандидатов на расширение решения в каждой позиции не будут пересекаться друг с другом.

Алгоритм содержит пять процедур, специфичных для конкретных приложений:

- `is_a_solution(a,k,input)`. Эта булева функция проверяет, составляют ли первые k элементов вектора `a` полное решение данной задачи. Последний аргумент, `input`, позволяет передавать в процедуру общую информацию. Например, с его помощью можно указать значение n, представляющее заданный размер решения. Эта информация может быть полезной при создании перестановок или подмножеств из n элементов, но при создании объектов переменного размера, таких как последовательности ходов игры, можно передавать другие данные, более соответствующие ситуации;
- `construct_candidates(a,k,input,c,ncandidates)`. Эта процедура записывает в массив `c` полный набор возможных кандидатов на k-ю позицию вектора `a`, при заданном содержимом первых $k-1$ позиций. Количество кандидатов, содержащихся в этом массиве, заносится в переменную `ncandidates`. Так же, как в предыдущей функции, аргумент `input` можно использовать для передачи в процедуру вспомогательной информации;
- `process_solution(a,k,input)`. Эта процедура выводит, вычисляет или иным образом обрабатывает полное решение после его создания;
- `make_move(a,k,input)` и `unmake_move(a,k,input)`. Эти процедуры позволяют модифицировать структуру данных в ответ на последнее перемещение, а также очистить структуру данных, если мы решим отменить это перемещение. При необходимости эту структуру можно было воссоздавать с нуля на основе вектора решений `a`, но

этот подход не является эффективным, когда с каждым перемещением связаны изменения, которые можно с легкостью отменить.

Эти процедуры функционируют в виде заглушек (т. е. ничего не делают) в вызовах процедуры `backtrack()` во всех примерах этого раздела, но применяются в программе решения головоломок судоку в *разделе 7.3*.

Для внепланового завершения программы используется глобальный флаг `finished`, который можно установить в любой прикладной процедуре.

Чтобы по-настоящему понять принцип работы алгоритма перебора с возвратом, нужно разобраться, как можно создавать такие объекты, как перестановки и подмножества, определяя правильное пространство состояний. Несколько примеров пространств состояний рассматривается в последующих подразделах.

7.1.1. Генерирование всех подмножеств

Критическим вопросом при разработке пространств состояний для представления комбинаторных объектов является количество объектов, которые нужно представить. Сколько существует подмножеств множества из n элементов, например, множества целых чисел $\{1, ..., n\}$? Для $n = 1$ существует два таких подмножества — $\{\}$ и $\{1\}$. Для $n = 2$ существует четыре подмножества, а для $n = 2$ — восемь. Как видим, количество подмножеств удваивается с каждым новым элементом множества; таким образом, для множества из n элементов существует 2^n подмножеств. Каждое подмножество описывается содержащимися в нем элементами. Чтобы сгенерировать все 2^n подмножеств, мы создаем массив (вектор) из n ячеек, в котором булево значение a_i указывает, содержит ли данное подмножество i-й элемент. В схеме нашего общего алгоритма перебора с возвратом $S_k = (true, false)$, в то время как значение a является решением при $k = n$. Теперь мы можем сгенерировать все подмножества, используя простые реализации процедур `is_a_solution()`, `construct_candidates()` и `process_solution()`, показанные в листинге 7.3.

Листинг 7.3. Реализация базовых процедур процедуры `backtrack()`

```
is_a_solution(int a[], int k, int n)
{
   return (k == n);                        /* k == n? */
}
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
   c[0] = TRUE;
   c[1] = FALSE;
   *ncandidates = 2;
}
process_solution(int a[], int k)
{
   int i;                                  /* Счетчик*/
   printf("{");
   for (i=1; i<=k; i++)
   if (a[i] == TRUE) printf(" %d",i);
  printf(" }\n");
}
```

Самой сложной из этих трех процедур оказывается процедура для вывода каждого подмножества после его создания!

Наконец, при вызове процедуры `backtrack` ей нужно передать соответствующие аргументы. Конкретно, это означает предоставление указателя на пустой вектор решения, установление k в ноль для обозначения того, что вектор пустой, и указание количества элементов в универсальном множестве (листинг 7.4).

Листинг 7.4. Вызов процедуры `backtrack()` для генерирования подмножеств

```
generate_subsets(int n)
{
   int a[NMAX];      /* Вектор решений */
   backtrack(a,0,n);
}
```

В каком порядке будут генерироваться подмножества множества {1, 2, 3}? Это зависит от порядка перемещений, выполняемых в процедуре `construct_candidates`. Так как *true* всегда идет перед *false*, то сначала будут сгенерированы все подмножества для *true*, а пустое подмножество, состоящее из всех *false*, генерируется последним: {123}, {12}, {13}, {1}, {23}, {2}, {3}, {}.

Изучите внимательно этот пример и убедитесь, что вы понимаете, как работает процедура перебора с возвратом. Задача генерирования подмножеств рассматривается более подробно в *разделе 14.5*.

7.1.2. Генерирование всех перестановок

Обязательным предварительным условием генерирования перестановок множества элементов {1, ..., n} является подсчет их количества. Для первого элемента перестановки имеется n вариантов. Вторым элементом может быть любой из оставшихся $n-1$ элементов, т. к. в перестановках повторение элементов не допускается. Последовательное применение этой процедуры дает нам $n! = \prod_{i=1}^{n} i$ разных перестановок.

Этот способ подсчета количества перестановок подсказывает подходящую структуру для их представления. В частности, создаем массив (вектор) a из n ячеек. Набором кандидатов на i-е место будет набор элементов, которые не вошли в $(i-1)$ элементов частичного решения, что соответствует первым $i-1$ элементам перестановки.

В схеме общего алгоритма перебора с возвратом S_k = {1, ..., n} – a, причем значение a является решением, когда $k = n$. Соответствующая процедура `construct_candidates()` представлена в листинге 7.5.

Листинг 7.5. Процедура `construct_candidates()` для генерирования всех перестановок

```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
   int i;                  /* Счетчик */
   bool in_perm[NMAX];     /* Какие элементы в перестановке? */
```

```
   for (i=1; i<NMAX; i++) in_perm[i] = FALSE;
   for (i=0; i<k; i++) in_perm[ a[i] ] = TRUE;
   *ncandidates = 0;
   for (i=1; i<=n; i++)
      if (in_perm[i] == FALSE) {
         c[ *ncandidates] = i;
         *ncandidates = *ncandidates + 1;
      }
}
```

Узнать, является ли i кандидатом на k-е место в перестановке, можно путем перебора всех элементов $k-1$ массива a, чтобы убедиться в отсутствии совпадений. Но гораздо лучший способ отслеживания элементов, находящихся в частичном решении, — организовать структуру данных в форме вектора разрядов (см. *раздел 12.5*), что позволит нам выполнять проверку за постоянное время.

Для завершения программы генерирования перестановок необходимо определить используемые в ней процедуры `process_solution` и `is_a_solution`, а также передать необходимые параметры вызываемой процедуре `backtrack`. Как определение процедур, так и передача параметров выполняется, по сути, так же, как и для программы генерирования подмножеств (листинг 7.6).

Листинг 7.6. Процедуры генерирования перестановок

```
process_solution(int a[], int k)
{
   int i;            /* Счетчик */
   for (i=1; i<=k; i++) printf(" %d",a[i]);
   printf("\n");
}
is_a_solution(int a[], int k, int n)
{
   return (k == n);
}
generate_permutations(int n)
{
   int a[NMAX];       /* Вектор решений */
   backtrack(a,0,n);
}
```

В результате упорядочения кандидатов эти перестановки генерируются в отсортированном порядке, т. е. 123, 132, 213, 231, 312 и 321. Задача генерирования перестановок рассматривается более подробно в *разделе 14.4*.

7.1.3. Генерирование всех путей в графе

Задача перечисления всех простых путей от вершины s к вершине t графа является более сложной, чем перечисление подмножеств или перестановок множества элементов. Не существует явной формулы для определения количества решений в зависимости от количества вершин и ребер, т. к. количество путей зависит от структуры графа.

Начальной точкой любого пути от вершины s к вершине t всегда является вершина s, т. е. вершина s является единственным кандидатом на первое место и $S_1 = \{s\}$. Возможными кандидатами на второе место в пути являются такие вершины v, для которых ребро (s, v) находится в графе, т. к. допустимый путь от вершины к вершине идет по ребрам между ними. Вообще говоря, множество S_{k+1} состоит из набора вершин, смежных с вершиной a_k, которые не были использованы в частичном решении A. Определение соответствующей процедуры `construct_candidates()` приводится в листинге 7.7.

Листинг 7.7. Процедура `construct_candidates()` для перечисления всех путей в графе

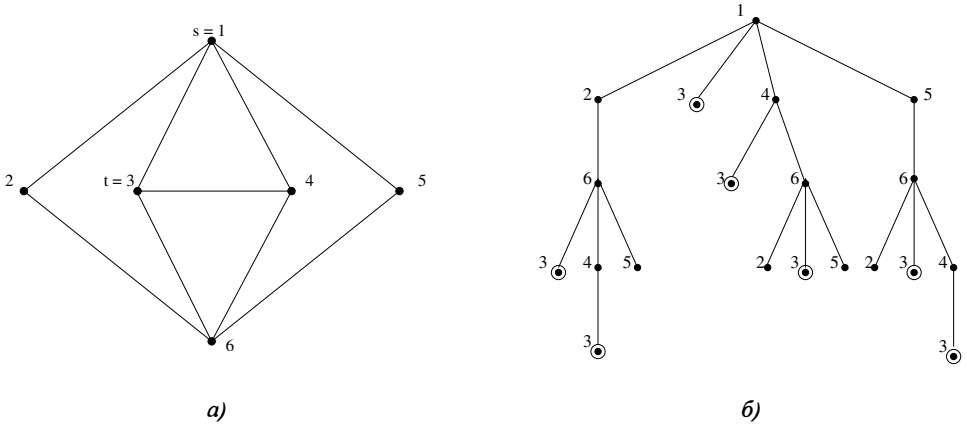
```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
   int i;                  /* Счетчики */
   bool in_sol[NMAX];      /* Что уже находится в решении? */
   edgenode *p;            /* Временный указатель */
   int last;               /* Последняя вершина в текущем пути */
   for (i=1; i<NMAX; i++) in_sol[i] = FALSE;
   for (i=1; i<k; i++) in_sol[ a[i] ] = TRUE;
   if (k==1) {             /* Всегда начинаем с вершины 1 */
      c[0] = 1;
      *ncandidates = 1;
   }
   else  {
      *ncandidates = 0;
      last = a[k-1] ;
      p = g.edges[last];
      while (p != NULL) {
         if (!in_sol[ p->y ]) {
            c[*ncandidates] = p->y;
            *ncandidates = *ncandidates + 1;

         p = p->next;
      }
   }
}
```

Условием правильного пути является $a_k = t$. Процедуры для определения решения и его обработки приводятся в листинге 7.8.

Листинг 7.8. Процедуры для определения решения и его обработки

```
is_a_solution(int a[], int k, int t)
{
   return (a[k] == t);
}
process_solution(int a[], int k)
{
   solution_count ++; /* Подсчитываем все пути от s к t */
}
```

Вектор решений A должен иметь достаточно элементов для представления всех n вершин, хотя, скорее всего, для большинства путей они не понадобятся. На рис. 7.1 показано дерево поиска с перечислением всех путей между определенной парой вершин графа.



а) *б)*

Рис. 7.1. Граф *(а)* и дерево поиска со всеми путями между вершинами s и t *(б)*

7.2. Отсечение вариантов поиска

Метод перебора с возвратом обеспечивает правильность результата, перечисляя все возможные комбинации. Перечисление всех $n!$ перестановок n вершин графа и выбор наилучшей из них дает нам правильный алгоритм для определения оптимального маршрута коммивояжера. Для каждой перестановки мы можем видеть, в действительности ли граф G содержит все ребра, указываемые в маршруте, и если содержит, то веса всех ребер суммируются.

Но предварительное генерирование всех перестановок для их дальнейшего исследования будет расточительным расходованием ресурсов. Допустим, что наш путь начинается в вершине v_1, но ребро (v_1, v_2) не является частью графа G. Тогда рассмотрение следующих $(n-2)!$ перестановок, сгенерированных, начиная с ребра (v_1, v_2), будет бессмысленной тратой времени. Разумнее отказаться от поиска после (v_1, v_2) и продолжить с (v_1, v_3). Ограничение набора следующих элементов таким образом, чтобы остались лишь перемещения, допустимые для текущей частичной конфигурации, позволяет значительно понизить сложность поиска.

Отсечением (pruning) называется метод прекращения поиска решения, как только установлено, что данное частичное решение невозможно расширить до полного. Задача коммивояжера состоит в определении самого дешевого маршрута, который проходит через все вершины. Допустим, что мы нашли маршрут t со стоимостью C_t. Потом в процессе продолжения поиска мы получаем частичное решение с суммой стоимости вершин $C_A > C_t$. Нужно ли продолжать исследование этого узла? Нет, так как любой маршрут $a_1, ..., a_k$ с этим префиксом будет стоить больше, чем маршрут t, и поэтому наверняка не является оптимальным. Отсечение таких бесперспективных частичных маршрутов на раннем этапе может существенно улучшить время исполнения.

Другим средством уменьшения вариантов возможных решений комбинаторного поиска является применение симметрии. Отсечение частичных решений, идентичных рассмотренным ранее, требует умения распознавать симметричные области поиска. Возьмем, например, состояние поиска кратчайшего пути после того, как были рассмотрены все частичные решения, начиная с v_1. Есть ли смысл продолжать поиск с частичными решениями, начинающимися с v_2? Нет. Любой маршрут, начинающийся и заканчивающийся в вершине v_2, можно рассматривать как смещенный маршрут, начинающийся и заканчивающийся в вершине v_1, т. к. эти маршруты являются циклами. Таким образом, для n вершин существует только $(n-1)!$ разных маршрутов, а не $n!$. Ограничивая первый элемент маршрута вершиной v_1, мы получаем экономию времени порядка n, не пропуская при этом никаких представляющих интерес решений. Такие закономерности могут быть далеко не очевидными, но, когда они обнаружены, их можно эксплуатировать.

Подведение итогов

Комбинаторный поиск можно использовать совместно с методами отсечения для решения задач оптимизации небольшого размера. Смысл понятия "небольшой" зависит от конкретной задачи, но обычно размер составляет $15 \le n \le 50$ элементов.

7.3. Судоку

Головоломки судоку очень популярны во всем мире. Многие газеты печатают их в своих дневных выпусках, выпущены целые сборники этих головоломок. О популярности судоку можно судить по тому факту, что авиакомпания British Airways издала приказ, запрещающий бортпроводникам решать их во время взлета и посадки. Я даже заметил, что во время моих лекций по алгоритмам в задних рядах аудитории довольно многие занимаются решением этих головоломок.

Что же представляет собой судоку? В наиболее распространенной форме это квадрат размером 9 × 9 клеточек, некоторые из которых содержат цифры от 1 до 9, а остальные пустые. Решение головоломки состоит в заполнении пустых клеточек таким образом, чтобы каждая строка, каждый столбец и каждый малый квадрат размером 3 × 3 клеточки, содержали все цифры от 1 до 9, без повторений и без пропусков. Пример головоломки судоку и ее решение показаны на рис. 7.2.

	3 5	1 2
	6	7
7		3
	4	8
1		
	1 2	
8		4
5		6

а)

6 7 3	8 9 4	5 1 2
9 1 2	7 3 5	4 8 6
8 4 5	6 1 2	9 7 3
7 9 8	2 6 1	3 5 4
5 2 6	4 7 3	8 9 1
1 3 4	5 8 9	2 6 7
4 6 9	1 2 8	7 3 5
2 8 7	3 5 6	1 4 9
3 5 1	9 4 7	6 2 8

б)

Рис. 7.2. Головоломка судоку *(а)* и ее решение *(б)*

Судоку хорошо поддается решению методом перебора с возвратом. Мы используем головоломку на рис. 7.2 для лучшей иллюстрации этого алгоритмического метода. Пространством состояний будут пустые клеточки, каждая из которых будет в конечном итоге заполнена какой-либо цифрой. Кандидатами на заполнение пустой клеточки (i, j) являются целые числа от 1 до 9, которых еще нет в строке i, столбце j и в малом квадрате, содержащим клеточку (i, j). Возврат осуществляется, когда больше нет кандидатов на заполнение клеточки.

Вектор решения `a`, поддерживаемый процедурой `backtrack`, может содержать в каждой ячейке только одно целое число. Этого достаточно для хранения содержимого клеточки (числа 1–9), но не для хранения ее координат. Поэтому для хранения позиций ходов мы используем отдельный массив `boardtype`. Основные структуры данных для поддержки нашего решения определены в листинге 7.9.

Листинг 7.9. Определение основных структур данных

```
#define DIMENSION 9                 /* Доска размером 9*9 */
#define NCELLS DIMENSION*DIMENSION /* На доске 9*9 имеется 81 клеточка */
typedef struct {
   int x, y;                        /* Координаты x и y клеточки */
} point;
typedef struct {
   int m[DIMENSION+1][DIMENSION+1]; /* Матрица содержимого доски */
   int freecount;          /* Количество оставшихся пустых клеточек */
   point move [NCELLS+1]; /* Как были заполнены клеточки */
} boardtype;
```

На очередном шаге игры мы должны сначала выбрать открытую клеточку, которую хотим заполнить следующей (процедура `next_square`), затем определить числа, являющиеся кандидатами на заполнение этой клеточки (процедура `possible_values`). Эти процедуры (листинг 7.10), по сути, ведут учет ходов, хотя некоторые детали их работы могут сильно повлиять на производительность.

Листинг 7.10. Генерирование кандидатов на заполнение клеточки

```
construct_candidates(int a[], int k, boardtype *board,
                     int c[],int *ncandidates)
{
   int x,y;                    /* Позиция следующего хода */
   int i;                      /* Счетчик */
   bool possible[DIMENSION+1]; /* Какие числа можно использовать
                                  для данной клеточки */
   next_square(&x,&y,board); /* Какую клеточку заполнять следующей */
   board->move[k].x = x;     /* Сохранение выбора следующей клеточки */
   board->move[k].y = y;
   *ncandidates = 0;
   if ((x<0) && (y<0)) return;   /* Ошибка: нет допустимых ходов */
   possible_values(x,y,board,possible);
   for (i=0; i<=DIMENSION; i++)
```

```
      if (possible[i] == TRUE) {
         c[*ncandidates] = i;
         *ncandidates = *ncandidates + 1;
      }
}
```

Структуры данных игровой доски необходимо обновлять, чтобы отображать заполнение клеточки значением кандидата, а также очищать заполненные клеточки в случае необходимости возврата с данной позиции. Для выполнения этих обновлений используются процедуры `make_move` и `unmake_move` (листинг 7.11), которые вызываются непосредственно из процедуры `backtrack`.

Листинг 7.11. Процедуры make_move и unmake_move

```
make_move(int a[], int k, boardtype *board)
{
   fill_square(board->move[k].x,board->move[k].y,a[k],board);
}
unmake_move(int a[], int k, boardtype *board)
{
   free_square(board->move[k].x,board->move[k].y,board);
}
```

Одной из важных задач, выполняемых этими процедурами, является отслеживание количества остающихся на доске пустых клеточек. Решение найдено, когда на доске больше нет пустых клеточек (листинг 7.12).

Листинг 7.12. Процедура отслеживания пустых клеточек

```
is_a_solution(int a[], int k, boardtype *board)
{
   if (board->freecount == 0)
      return (TRUE);
   else
      return(FALSE);
}
```

Когда решение найдено, устанавливается глобальный флаг `finished`, что служит сигналом к прекращению поиска и выводу решения. Это можно делать, не опасаясь никаких последствий, т. к. классические головоломки судоку могут иметь только одно решение. Головоломки с расширенной интерпретацией могут иметь громадное количество решений. В самом деле, для пустой головоломки (т. е. без начальных цифр) существует 6 670 903 752 021 072 936 960 решений. Чтобы не просматривать все эти решения, мы и прекращаем поиск (листинг 7.13).

Листинг 7.13. Завершение поиска и обработка решения

```
process_solution(int a[], int k, boardtype *board)
{
   print_board(board);
   finished = TRUE;
}
```

Эта процедура завершает нашу программу, но остается необходимость в написании процедур для определения следующей клеточки для заполнения (`next_square`) и поиска кандидатов на заполнение этой клеточки (`possible_values`). Для выбора следующей клеточки для заполнения подходят два способа.

- *Выбор произвольной клеточки.* Выбираем первую попавшуюся пустую клеточку. Нам все равно, какую выбирать, поскольку нет очевидных оснований считать, что один эвристический метод окажется лучше другого.
- *Выбор клеточки с наименьшим количеством кандидатов.* При этом подходе мы проверяем количество оставшихся кандидатов на заполнение каждой пустой клеточки (i, j), т. е. количество цифр, которые еще не используются ни в строке i, ни в столбце j, ни в малом квадрате, содержащим клеточку (i, j). По результатам этих исследований мы выбираем клеточку с наименьшим количеством кандидатов на ее заполнение.

Хотя оба подхода дают правильные результаты, второй позволяет найти решение намного быстрее. Часто имеются пустые клеточки только с одним кандидатом, которые нельзя заполнить иной цифрой, кроме как этим единственным оставшимся кандидатом. Такие клеточки вполне можно заполнить на данном этапе, т. к. это поможет уменьшить количество возможных кандидатов для заполнения других пустых клеточек. Конечно же, в этом случае выбор каждой следующей клеточки для заполнения будет занимать больше времени, но если головоломка достаточно легкая, то нам, может быть, никогда не придется выполнять возврат.

Если для клеточки с наименьшим количеством кандидатов имеется два кандидата, то вероятность угадать правильный из них с первого раза равна 1/2, по сравнению с вероятностью 1/9 угадать правильного кандидата для клеточки без ограничений на количество кандидатов. Уменьшив среднее количество кандидатов для каждой клеточки, например, с трех до двух, мы получим громадное повышение производительности, т. к. это уменьшение дает прогрессивный выигрыш для каждой следующей клеточки. Например, если нам нужно заполнить 20 клеточек, то два кандидата на каждую клеточку дают 1 048 576 возможных вариантов заполнения. А уровень ветвления, равный 3, для каждой из 20 клеточек дает в 3 000 раз больше вариантов!

Выбрать возможных кандидатов для каждой клеточки можно двумя способами.

- *Локальный выбор.* Наш алгоритм перебора с возвратом выдаст правильный результат, если процедура генерирования кандидатов на заполнение клеточки (i, j), т. е. процедура `possible_values`, действует очевидным образом и предоставляет на выбор цифры от 1 до 9, которых еще нет в данной строке, столбце или малом квадрате.
- *Просмотр вперед.* Что будет, если для нашего текущего частичного решения существует какая-то *другая* пустая клеточка, для которой локальные критерии не оставляют кандидатов? В таком случае данное частичное решение невозможно довести до полного. Получается, что ситуация вокруг какой-то другой клеточки делает *действительное* количество кандидатов для клеточки (i, j) равным нулю!

  Со временем мы подойдем к этой другой клеточке, обнаружим, что для нее нет действительных кандидатов, и нам придется возвращаться назад. Но зачем вообще идти к этой клеточке, если все затраченные на это усилия будут напрасными? Будет

намного выгоднее выполнить возврат к текущей позиции и продолжить поиск в другом направлении[1].

Для успешного отсечения непродуктивных ветвей поиска требуется просмотр вперед, позволяющий обнаружить, что данный путь решения является тупиковым, и возвратиться из него на новый как можно раньше.

В табл. 7.1 показано количество вызовов процедуры определения решения `is_a_solution` для всех четырех комбинаций выбора следующей клеточки и возможных кандидатов для трех разных уровней сложности головоломки судоку.

- Головоломка низкого уровня сложности предназначена для решения человеком, а не компьютером. В действительности, моя программа решила ее без единого возврата, когда для следующей клеточки выбиралась клеточка с наименьшим количеством кандидатов.
- Головоломка средней сложности оказалась не под силу ни одному из вышедших в финал участников Мирового чемпионата по судоку в марте 2006 года. Но для программы потребовалось только несколько возвратов, чтобы решить эту головоломку.
- Головоломка высокого уровня сложности показана на рис. 7.2 и содержит только 17 заполненных клеточек. Это наименьшее известное количество заполненных клеточек на всех экземплярах задачи, которое дает только одно полное решение.

Таблица 7.1. Количество шагов для получения решения при разных стратегиях отсечения

Условие отсечения		Уровень сложности		
`next_square`	`possible_values`	Низкий	Средний	Высокий
произвольное значение	локальный выбор	1,904,832	863,305	программа не завершена
произвольное значение	будущий выбор	127	142	12,507,212
наименьшее количество кандидатов	локальный выбор	48	84	1,243,838
наименьшее количество кандидатов	будущий выбор	48	65	10,374

Что считается головоломкой высокого уровня сложности, зависит от применяемого для ее решения эвристического алгоритма. Несомненно, среди ваших знакомых есть люди, которым теория кажется труднее, чем практическое программирование, а есть и такие, кто думает иначе. Алгоритм A вполне может считать, что задача I_1 легче, чем задача I_2, в то время как алгоритм B видит трудность этих задач в обратном порядке.

[1] Этот подход с просмотром вперед мог бы естественным образом следовать из подхода выбора клеточки с наименьшим количеством. кандидатов, если бы было разрешено выбирать клеточки, для которых нет кандидатов. Но в моей реализации уже заполненные клеточки рассматривались, как не имеющие ходов, что ограничивает выбор следующей клеточки клеточками, по крайней мере, с одним кандидатом.

Какие выводы мы можем сделать из этих экспериментов? Предварительное исследование дальнейших вариантов решения с целью отсечения тупиковых является самым лучшим способом разрежения пространства поиска. Без применения этой операции мы никогда не решили бы головоломку самого высокого уровня, а более легкие головоломки решили в тысячи раз медленнее.

Разумный подход к выбору следующей клеточки имел аналогичный эффект, хотя технически мы просто изменяли порядок выполнения работы. Но обработка клеточек с наименьшим количеством кандидатов первыми равнозначна понижению исходящей степени каждого узла дерева, а каждая заполненная клеточка уменьшает количество возможных кандидатов для других клеточек.

При произвольном выборе следующей клеточки решение головоломки на рис. 7.2 заняло почти час. Несомненно, моя программа решила большинство других головоломок быстрее, но головоломки судоку предназначены для решения людьми за гораздо меньшее время. Выбор клеточки с наименьшим количеством кандидатов позволил сократить время поиска более чем в 1 200 раз.

Вот так проявляется вся мощь отсечения тупиковых вариантов поиска. Использование даже простых стратегий отсечения может значительно уменьшить время исполнения.

7.4. История из жизни. Покрытие шахматной доски

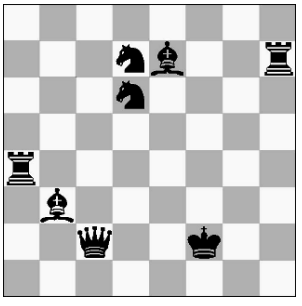
Каждый ученый мечтает о решении классической задачи — такой, которая оставалась нерешенной в течение нескольких столетий. Есть что-то романтическое в общении с ушедшими поколениями, участии в эволюционном развитии научной мысли и оказании помощи человечеству на его пути вверх по лестнице технического прогресса.

Задача может оставаться нерешенной в течение длительного времени по разным причинам. Возможно, она такая трудная, что для ее решения требуется уникальный мощный интеллект. Или же, возможно, еще не были разработаны идеи или методы, требуемые для решения данной задачи. Наконец, возможно, что задача никого не заинтересовала настолько, чтобы он всерьез занялся ею. Однажды я помог решить задачу, которая оставалась нерешенной свыше сотни лет.

Люди увлекаются игрой в шахматы в течение тысяч лет. Эффект комбинаторного взрыва был впервые зафиксирован в легенде, согласно которой изобретатель шахмат запросил у правителя в качестве награды за свое изобретение самую малость — одно зерно пшеницы на первое поле шахматной доски, два — на второе и т. д., т. е. вдвое больше на каждое следующее $(i + 1)$-е поле, чем на предыдущее i-е. Но когда правитель узнал, что ему придется раскошелиться на $\sum_{i=0}^{63} 2^i = 2^{64} - 1 =$ 18 446 744 073 709 551 61 зерен пшеницы, то его ликование по поводу приобретения такой замечательной игры по такой низкой цене сменилось негодованием по поводу алчности и коварства изобретателя.

Отрубив изобретателю голову, правитель был первым, кто применил метод отсечения в качестве меры контроля комбинаторного взрыва.

В 1849 г. немецкий гроссмейстер Йозеф Клинг (Josef Kling) поставил вопрос, возможно ли одновременно держать под ударом все 64 поля шахматной доски восемью основными фигурами — королем, ферзем, двумя конями, двумя ладьями и двумя слонами разного цвета. Фигуры не держат под ударом поле, на котором они находятся. Расстановки фигур, которые держат под ударом 63 поля, подобные показанным на рис. 7.3, были известны с далеких времен, но вопрос, являются ли такие расстановки наилучшими возможными, оставался открытым.



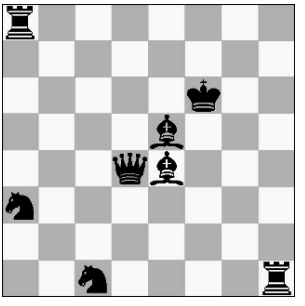


Рис. 7.3. Расстановки фигур, ставящие под удар 63 поля, но не все 64

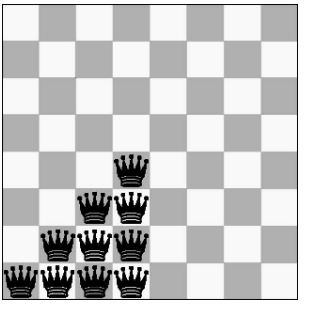


Рис. 7.4. Десять уникальных позиций для ферзя с учетом симметрии

Казалось, задача решается исчерпывающим комбинаторным перебором, но возможность решить ее таким способом зависела от размера пространства поиска.

Посмотрим, сколько существует способов расстановки на шахматной доске восьми основных шахматных фигур (короля, ферзя, двух ладей, двух слонов и двух коней). Очевидный предел таких расстановок равен $64!/(64 - 8)! = 178\,462\,987\,637\,760 \approx 10^{15}$. Однако было бы неразумно надеяться на перебор более чем 10^9 расстановок на обычном компьютере за приемлемое время.

Таким образом, чтобы решить задачу расстановок, необходимо выполнить отсечение значительного объема пространства поиска. После удаления ортогональных и диагональных симметричных вариантов для ферзя останется только десять возможных позиций (рис. 7.4).

После постановки ферзя для размещения пары ладей или коней остается $64 \cdot 63/2 = 2\,016$ положений, для короля — 64, и 32 для каждого из двух слонов. После такого отсечения остается $2\,663\,550\,812\,160 \approx 10^{13}$ разных расстановок, что все равно слишком много для исчерпывающего перебора.

Все эти расстановки можно было сгенерировать с помощью поиска с возвратом, но пространство поиска нуждалось в дополнительном значительном разрежении. Для этого нужно было найти способ, чтобы быстро доказать, что для определенной частичной расстановки не существует возможности ее завершения, чтобы поставить под удар все 64 поля. Предположим, что мы уже разместили на доске семь фигур, которые держат под ударом все поля за исключением десяти. Далее допустим, что осталось разместить короля. Существует ли в данной ситуации поле, с которого король может держать под

ударом оставшиеся девять полей? Ответ однозначно должен быть отрицательным, т. к. по правилам шахмат король может держать под ударом самое большее восемь полей. Таким образом, нет смысла проверять наличие решения при расположении короля на любом из оставшихся полей. Такая стратегия отсечения расстановок может дать большую экономию, но для ее реализации необходимо внимательно исследовать порядок постановки фигур на доску. Каждая фигура может держать под ударом определенное максимальное количество полей: ферзь — 27, король и конь — 8, ладья — 14, а слон — 13. Хорошей стратегией может быть постановка фигур на доску в порядке убывания их влияния. Таким образом, мы можем выполнять отсечение дальнейших расстановок, когда количество полей, не находящихся под боем, превышает сумму возможностей оставшихся непоставленных фигур. Эта сумма минимизируется постановкой фигур в убывающем порядке их возможностей держать поля под ударом.

Когда мы реализовали перебор с возвратом, используя эту стратегию отсечения, мы отсекли свыше 95% пространства поиска. После оптимизации метода генерирования постановок фигур наша программа могла исследовать 1 000 полей за секунду. Но и это было слишком медленно, т. к. $10^{12}/10^3 = 10^9$ секунд означало свыше 11 000 дней! Хотя мы могли бы настроить программу и ускорить время ее исполнения примерно на порядок, но в действительности нам было нужно найти способ отсечь еще больше тупиковых расстановок.

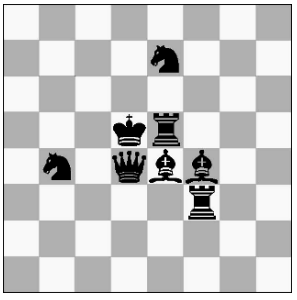
Чтобы отсечение было эффективным, нужно устранить большое количество расстановок одним приемом, а наши предыдущие попытки в этом направлении были слишком слабыми. А если поставить на доску не восемь фигур, а больше? Очевидно, что чем больше фигур поставлено на доску, тем больше вероятность, что они будут держать под ударом все 64 поля. Но если большее количество фигур не держит под ударом все 64 поля, то и любое из восьмифигурных подмножеств этого множества не способно на это. Такой подход позволяет устранить огромное количество расстановок, удалив всего лишь один узел.

Так что в последней версии нашей программы узлы дерева поиска представляли расстановки, которые могли содержать любое количество фигур и больше, чем одну фигуру на одном и том же поле. Для определенной расстановки мы различали *сильную* и *слабую* атаку поля. Сильная атака соответствует обычной атаке по шахматным правилам. Поле считается слабо атакованным, если оно сильно атаковано некоторым подмножеством имеющихся фигур, т. е. если блокировка одних фигур другими игнорируется. Как можно видеть на рис. 7.5, все 64 поля можно держать под слабой атакой восемью фигурами.

Наш алгоритм выполнял два прохода. В первом проходе перечислялись расстановки, в которых каждое поле находилось под слабой атакой, а во втором список разрежался, благодаря фильтрации расстановок с блокирующими фигурами. Расстановку со слабой атакой вычислить намного легче, т. к. в этом случае нет необходимости принимать во внимание блокирующие фигуры, а любое множество сильных атак является подмножеством множества слабых атак. Любую расстановку, содержащую поле под слабой атакой, можно было отсечь.

Эта версия программы была достаточно эффективной, чтобы на медленном IBM PC-RT 1988 года выпуска завершить поиск меньше чем за один день. Она не нашла ни

одной расстановки, удовлетворяющей первоначальным условиям задачи. Но с ее помощью мы смогли доказать, что возможно поставить под удар все поля шахматной доски, используя *семь* фигур, при условии, что ферзь и конь могут занимать одно и то же поле. На рис. 7.6 показано соответствующее расположение фигур, причем ферзь и конь, расположенные на одном поле, обозначены белым ферзем.



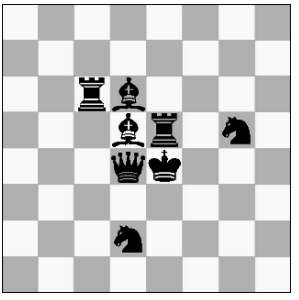


Рис. 7.5. Слабая атака всех 64 полей шахматной доски восемью основными фигурами

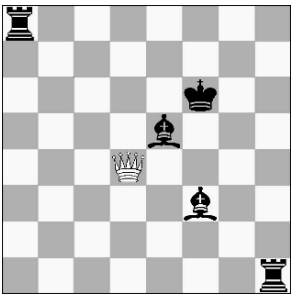


Рис. 7.6. Атака всех полей шахматной доски

Подведение итогов

Использование интеллектуальной стратегии отсечения тупиковых решений может позволить с удивительной легкостью решить комбинаторные задачи, на первый взгляд кажущиеся неразрешимыми. Выполненное должным образом отсечение окажет большее воздействие на время перебора, чем любой другой фактор.

7.5. Эвристические методы перебора

Метод перебора с возвратом позволяет нам найти наилучшее из возможных решений согласно данным условиям. Но любой алгоритм, исследующий все возможные конфигурации, обречен на провал на входных экземплярах большого размера. Эвристические методы предоставляют альтернативный подход к оптимизации трудных комбинаторных задач.

В этом разделе мы рассмотрим такие эвристические методы поиска. Основное внимание уделяется методу имитации отжига, который я считаю наиболее надежным для практического применения. Эвристические поисковые алгоритмы поначалу кажутся каким-то шаманством, но если их подвергнуть тщательному исследованию, то выяснится, что многое в их работе поддается вполне логическому объяснению.

Мы рассмотрим три разных эвристических метода поиска: *произвольную выборку* (random sampling), *градиентный спуск* (gradient descent) и *имитацию отжига* (simulated annealing). Для сравнения этих трех эвристических методов мы испытаем их на задаче коммивояжера. Все три метода имеют два общих компонента.

- *Представление пространства решений.* Это полное, но при этом краткое описание множества возможных решений задачи. Для задачи коммивояжера пространство решений содержит $(n-1)!$ элементов, т. е. все возможные циклические перестанов-

ки вершин. Для представления каждого элемента пространства решений нам требуется соответствующая структура данных. Для задачи коммивояжера решения-кандидаты можно представлять с помощью массива S, содержащего $n-1$ вершин, где ячейка S_i определяет $(i+1)$-ю вершину маршрута, начинающегося в вершине v_1.

- *Функция стоимости.* Поисковые алгоритмы должны иметь функцию *стоимости* или *оценки*, чтобы определять качество каждого элемента пространства решений. Наш эвристический алгоритм поиска определяет элемент с наилучшей возможной оценкой — наибольшим или наименьшим значением, в зависимости от природы задачи. В случае задачи коммивояжера функция стоимости для оценки данного кандидата решения S должна лишь сложить вместе все сопутствующие затраты, а именно вес всех ребер (S_i, S_{i+1}), где S_{n+i} соответствует v_i.

7.5.1. Произвольная выборка

Самым простым методом организации поиска в пространстве решений является произвольная выборка, которая также называется *методом Монте-Карло*. В этом случае последовательно создаются и оцениваются произвольные решения; процесс прекращается, как только будет найдено достаточно хорошее решение или когда нам надоест ждать его получения (что более вероятно). В качестве конечного решения выбирается самое лучшее решение изо всех исследованных в процессе выборки.

Для действительно произвольной выборки элементы необходимо выбирать из пространства решений равномерно произвольным образом. Это означает, что вероятность выбора любого элемента из пространства решений в качестве следующего кандидата должна быть одинаковой. Реализация такой выборки может быть сложнее, чем может казаться с первого взгляда. Алгоритмы для генерирования случайных перестановок, подмножеств, разбиений и графов рассматриваются в *разделах 14.4–14.7*. А в листинге 7.14 предоставлена процедура произвольного выбора решений.

Листинг 7.14. Процедура произвольного выбора решений

```
random_sampling(tsp_instance *t, int nsamples, tsp_solution *bestsol)
{
   tsp_solution s;      /* Текущее решение задачи коммивояжера */
   double best_cost;    /* Самая лучшая стоимость на данный момент */
   double cost_now;     /* Текущая стоимость */
   int i;               /* Счетчик */
   initialize_solution(t->n,&s);
   best_cost = solution_cost(&s,t);
   copy_solution(&s,bestsol);
   for (i=1; i<=nsamples; i++) {
      random_solution(&s);
      cost_now = solution_cost(&s,t);
      if (cost_now < best_cost) {
         best_cost = cost_now;
         copy_solution(&s,bestsol);
      }
   }
}
```

В каких случаях можно успешно использовать произвольную выборку?

- *Пространство решений содержит высокую долю приемлемых решений.* Найти травинку в стоге сена намного легче, чем иголку. Когда существует много приемлемых решений, то произвольная выборка должна быстро привести к одному из них.

  Одним из примеров успешного применения метода произвольной выборки является поиск простых чисел. Построение больших случайных простых чисел для ключей является важным аспектом криптографических систем, таких, как, например, RSA-кодирование. Приблизительно каждое ln*n*-е число является простым, поэтому, чтобы найти простое число длиной в несколько сотен цифр, требуется выполнить умеренное количество выборок.

- *Пространство решений не является однородным.* Произвольную выборку нужно применять в тех случаях, когда отсутствуют какие бы то ни было признаки приближения к решению. Допустим, что вам нужно выбрать среди своих друзей такого, у которого номер полиса социального страхования заканчивается на 00. Для решения этой задачи не существует другого метода, кроме как спросить произвольно выбранного товарища, каков номер его страховки.

  Возвратимся опять к задаче поиска больших простых чисел. Эти числа разбросаны среди других целых чисел без какой-либо системы. Использование произвольной выборки для их поиска будет не хуже любого другого метода.

Но подходит ли метод произвольной выборки для решения задачи коммивояжера? Нет. Самым лучшим решением задачи коммивояжера для 48 столиц континентальных штатов, которое я нашел, выполнив 1,5 миллиона произвольных перестановок, было 101 712,8 миль. Это больше чем в три раза превышает протяженность оптимального маршрута! Пространство решений этой задачи почти полностью состоит из посредственных и плохих решений, поэтому качество решений растет очень медленно с ростом количества выборок, поделенного на время поиска. Чтобы дать представление о разбросе решений между выборками, на рис. 7.7 показаны колебания результатов произвольно выбираемых решений (как правило, низкого качества) задачи коммивояжера для столиц американских штатов.

Подобно задаче коммивояжера, большинство встречающихся задач имеют сравнительно небольшое количество хороших решений, но высокую степень однородности пространства решений. Для эффективного решения таких задач требуются более мощные эвристические алгоритмы.

Остановка для размышлений. Выбор пары

Задача. Требуется выбрать две произвольные вершины графа и поменять их местами. Предложите эффективный алгоритм для генерирования произвольных элементов с равномерным распределением из $\binom{n}{2}$ неупорядоченных пар множества $\{1, ..., n\}$.

Решение. Рассмотрим следующую процедуру для генерирования случайных неупорядоченных пар чисел:

```
i   random_int(1, n-1);
j = random_int(i+1, n);
```

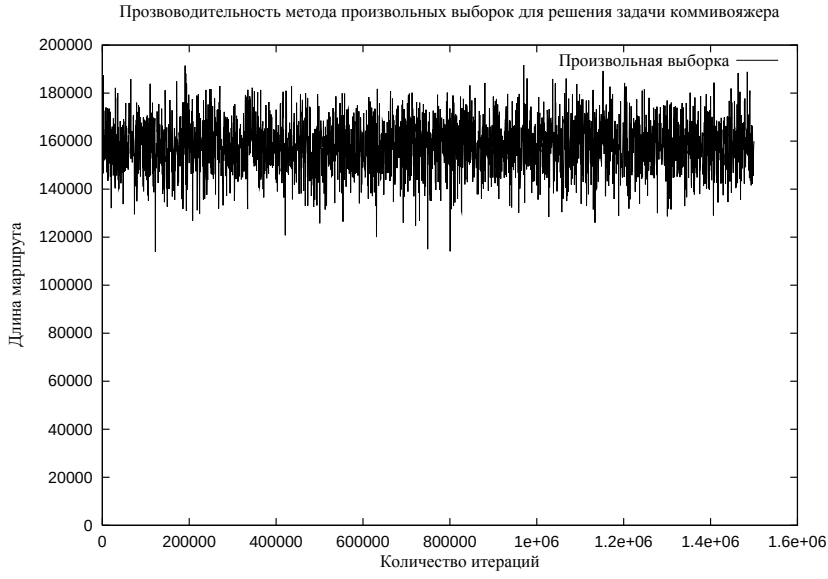


Рис. 7.7. Соотношения между временем поиска решений и их качеством для решения задачи коммивояжера методом произвольной выборки

Ясно, что эта процедура, на самом деле, генерирует неупорядоченные пары, т. к. $i < j$. Кроме этого, если допустить, что функция `random_int` генерирует целые числа в диапазоне своих двух аргументов однородно, также ясно, что можно сгенерировать все $\binom{n}{2}$ неупорядоченные пары.

Будет ли распределение этих пар однородным? Ответ на этот вопрос является отрицательным. Какова вероятность генерирования пары (1, 2)? Вероятность получения 1 равна $1/(n-1)$, а вероятность получения 2 также равна $1/(n-1)$, что дает нам вероятность получения пары $p(1, 2) = 1/(n-1)$. Но какова вероятность получения пары чисел $(n-1, n)$? Вероятность получения первого числа равна $1/n$, но для второго кандидата пары существует только один возможный вариант! Эта пара будет выпадать в n раз чаще, чем первая.

Проблема заключается в том, что количество пар, содержащих первое большое число, меньше, чем пар, содержащих первое малое число. Проблему можно было бы решить, вычислив, сколько именно неупорядоченных пар начинаются с числа i (ровно $n-i$), и соответствующим образом скорректировав вероятность. Тогда второе число пары можно было бы выбирать произвольным образом в диапазоне от $i+1$ до n с равномерным распределением.

Но вместо того, чтобы заниматься вычислениями, лучше воспользоваться тем обстоятельством, что генерирование n^2 упорядоченных пар является довольно простой задачей. Произвольно выбираем два целых числа независимо друг от друга. Игнорируя упорядоченность (т. е. превращая упорядоченную пару в неупорядоченную пару (x, y), где $x < y$), мы получаем вероятность генерирования каждой пары разных цифр, равную $2/n^2$. В случае генерирования пары (x, x) она отбрасывается и генерируется новая пара.

Алгоритм для генерирования равномерно распределенных произвольных пар целых чисел с ожидаемым постоянным временем исполнения приводится в листинге 7.15.

Листинг 7.15. Генерирование равномерно распределенных произвольных пар целых чисел

```
do {
   i = random_int(1,n);
   j = random_int(1,n);
   if (i > j) swap(&i,&j);
} while (i==j);
```

7.5.2. Локальный поиск

Допустим, что нам нужно найти эксперта по алгоритмам для решения определенной задачи. Мы *можем* позвонить по случайному телефонному номеру и спросить ответившего, не является ли он таким специалистом. В случае отрицательного ответа мы вешаем трубку и повторяем процесс, пока не найдем требуемого нам профессионала. Возможно, что после многократных звонков мы и найдем нужного человека, но было бы намного эффективнее спросить первого ответившего, нет ли среди его знакомых такого, который, возможно, знает эксперта по алгоритмам, и позвонить *ему* следующему.

Такая стратегия исследования прилегающего пространства вокруг каждого элемента пространства решений называется *локальным поиском*. Каждый элемент x в пространстве решений можно рассматривать, как вершину с исходящим ребром (x, y), направленным к каждому кандидату на решение y, являющемуся соседом x. Поиск выполняется из вершины x по направлению к наиболее перспективному кандидату вблизи этой вершины.

Мы не хотим явно создавать граф этого района для пространства решений большого размера. Представьте себе задачу коммивояжера, которая в таком графе будет иметь $(n-1)!$ вершин. Поэтому мы ищем решение посредством эвристического алгоритма, т. к. не надеемся найти решение в течение приемлемого времени исчерпывающим перебором всех вариантов.

На самом деле, мы хотим получить механизм перехода к следующему возможному решению, слегка модифицировав текущее. Типичные механизмы перехода включают обмен местами произвольной пары элементов или изменение (вставка или удаление) одного элемента решения.

Наиболее очевидным механизмом перехода для задачи коммивояжера был бы обмен местами произвольной пары вершин S_i и S_j текущего маршрута, как показано на рис. 7.8.

Такой обмен вершин изменяет до восьми ребер маршрута, удаляя ребра, смежные с S_i и S_j, и добавляя другие. В идеальном случае влияние этих инкрементальных изменений на возможность оценки качества решения можно вычислить также инкрементным образом, вследствие чего время исполнения функции оценки стоимости будет пропорционально размеру изменений (обычно постоянное), а не линейное по отношению к размеру решения.

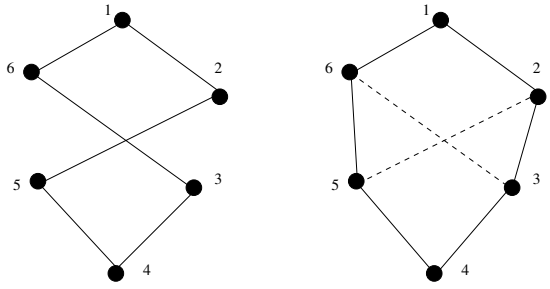


Рис. 7.8. Улучшение маршрута коммивояжера за счет обмена местами вершин 2 и 6

Эвристический алгоритм локального поиска начинает работу с произвольного элемента пространства решений и сканирует смежную с ним область, пытаясь найти подходящего кандидата на выполнение перехода. Для задачи коммивояжера таким кандидатом будет переход, который понижает стоимость маршрута. А для задачи другого типа, называемой *восхождением по выпуклой поверхности* (hill-climbing), мы пытаемся найти самую высшую (или самую низшую) точку, начав поиск в произвольной точке и выбирая любой промежуточный маршрут, ведущий в требуемом направлении. Эта процедура выбора промежуточного отрезка маршрута повторяется до тех пор, пока не дойдет до точки, в которой все исходящие направления не удовлетворяют нашим требованиям (листинг 7.16). Теперь вы "Царь горы" (или "Король канавы").

Листинг 7.16. Порцедура восхождения по выпуклой поверхности

```
hill_climbing(tsp_instance *t, tsp_solution *s)
{
   double cost;        /* Самая лучшая стоимость на данный момент */
   double delta;       /* Стоимость обмена */
   int i, j;           /* Счетчики */
   bool stuck;         /* Это решение лучшее? */
   double transition();
   initialize_solution(t->n, s);
   random_solution(s);
   cost = solution_cost(s, t);
   do {
      stuck = TRUE;
         for (i=1; i<t->n; i++)
         for (j=i+1; j<=t->n; j++) {
            delta = transition(s, t, i, j);
            if (delta < 0) {
               stuck = FALSE;
               cost   cost + delta;
            }
            else
               transition(s, t, j, i) ;
         }
   } while (!stuck);
}
```

Но, скорее всего, вы не "Царь горы". И вот почему. Допустим, что вы решили начать свое восхождение на гору, проснувшись утром в отеле на горнолыжном курорте. Ближайшей высшей точкой будет верхний этаж отеля, а потом его крыша. А дальше вам идти некуда. Чтобы взобраться на вершину горы, вам нужно сначала выйти на улицу, для чего придется спуститься на первый этаж отеля. Но это нарушает требование, чтобы каждый шаг повышал ваше местонахождение. Алгоритм восхождения по выпуклой поверхности и подобные эвристические алгоритмы, такие как жадный поиск или поиск методом градиентного спуска, позволяют быстро получить оптимальные локальные результаты, но часто не справляются с поиском наилучшего глобального решения.

В каких случаях локальный поиск приводит к успеху?

- *Пространство решений является высокооднородным.* Метод восхождения по выпуклой поверхности лучше всего работает с *выпуклым* пространством решений, т. е. пространством, содержащим ровно одно возвышение. Таким образом, независимо от местонахождения в пространстве решений точки начала поиска, у нас всегда имеется направление, в котором нужно продолжать поиск до тех пор, пока мы не дойдем до глобального максимума.

  Этим свойством обладают многие естественные задачи. В частности, можно считать, что двоичный поиск начинается в середине пространства решений. В этой точке существует только одно из двух возможных направлений, в котором нужно идти, чтобы приблизиться к целевому элементу. Симплексный алгоритм для линейного программирования (см. *раздел 13.6*) представляет собой не что иное, как восхождение по выпуклой поверхности правильного пространства решений, но, тем не менее, гарантирует получение оптимального решения для любой задачи линейного программирования.

- *Стоимость оценки изменения намного ниже стоимости глобальной оценки.* Стоимость оценки произвольного решения из n вершин для задачи коммивояжера равна $\Theta(n)$, т. к. нам нужно суммировать стоимость каждого ребра в циклической перестановке, описывающей маршрут. Но когда эта стоимость известна, то стоимость маршрута после обмена местами данной пары вершин можно определить за постоянное время.

  Если у нас имеется очень большое значение n и ограниченное время для поиска, то лучше потратить это время на выполнение нескольких инкрементальных оценок, чем на несколько произвольных выборок, даже если мы ищем иголку в стоге сена.

Основным недостатком локального поиска является то, что как только мы нашли локальный оптимум, то не остается больше никаких вариантов для поиска глобального решения. Конечно же, если у нас имеется время, мы можем начать новый поиск с другой произвольно выбранной точки, но в поисковом пространстве, содержащем много небольших возвышенностей, маловероятно найти оптимальное решение.

Насколько хорошим является метод локального поиска для решения задачи коммивояжера? Намного лучшим, чем произвольная выборка, при сходных временных затратах. Выполнив около 1,5 миллиона оценок маршрута задачи коммивояжера для столиц 48 континентальных штатов, мы получили самый лучший результат длиной 40 121,2 мили, что всего лишь на 19,5% больше, чем оптимальный маршрут длиной

33 523,7 мили. На рис. 7.9 показаны результаты применения метода локального поиска: многократные переходы от случайных маршрутов к удовлетворительным решениям примерно одинакового качества.

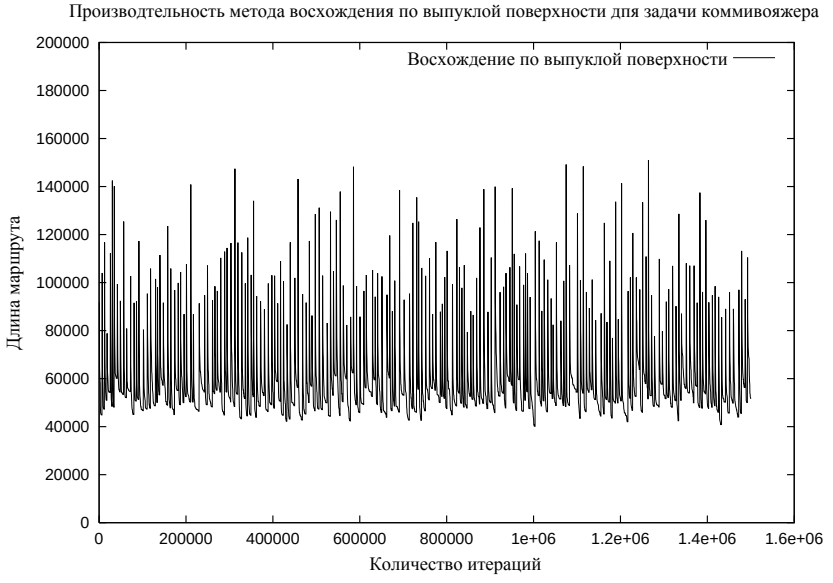


Рис. 7.9. Отношение времени поиска к качеству решений задачи коммивояжера для метода восхождения по выпуклой поверхности

Хотя этот результат намного лучше, чем полученный методом произвольных выборок, действительно хорошим его назвать трудно. К примеру, были бы вы довольны, если бы вам пришлось платить на 19,6% больше налогов, чем вы в действительности должны? Вывод: нам нужны более мощные методы для получения решения, близкого к оптимальному.

7.5.3. Имитация отжига

Имитация отжига (simulated annealing) представляет собой эвристическую процедуру поиска, которая допускает случайные переходы, что ведет к более дорогостоящим (и, соответственно, худшим) решениям. Это выглядит как шаг назад, но позволяет удержать поиск от зацикливания на оптимальном локальном решении. Если вернуться к нашей аналогии с покорением горных вершин, начиная с комнаты горнолыжного отеля, можно сказать, что теперь нам разрешается сойти вниз по лестнице или выпрыгнуть в окно, а затем продолжить восхождение в правильном направлении. Идея имитации отжига является аналогией физического процесса остывания расплавленных материалов и сопутствующего перехода в твердое состояние. В теории термодинамики энергетическое состояние системы определяется энергетическим состоянием каждой составляющей его частицы. Частица переходит из одного энергетического состояния в другое произвольным образом, при этом переходы между состояниями обуславливаются температурой системы.

В частности, вероятность перехода $P(e_i, e_j, T)$ из энергетического состояния e_i в состояние e_j при температуре T определяется формулой:

$$P(e_i, e_j, T) = e^{(e_i - e_j)/(k_B T)},$$

где k_B — постоянная Больцмана.

Каков смысл этой формулы? Чтобы ответить на этот вопрос, рассмотрим значение показателя степени при разных условиях. Вероятность перехода из высокоэнергетического состояния в низкоэнергетическое очень велика. Тем не менее, существует ненулевая вероятность перехода в высокоэнергетическое состояние, причем переходы малой амплитуды вероятнее, чем значительные. Кроме этого, чем выше температура, тем выше вероятность энергетических переходов.

Какое отношение все это имеет к комбинаторной оптимизации? При охлаждении физическая система стремится достичь состояния с минимальной энергией. Минимизация полной энергии представляет собой задачу комбинаторной оптимизации для любого набора дискретных частиц. Посредством произвольных переходов, генерируемых согласно данному распределению вероятностей, мы можем эмулировать физические процессы, чтобы решать произвольные задачи комбинаторной оптимизации. Псевдокод алгоритма для такой эмуляции приводится в листинге 7.17.

Листинг 7.17. Алгоритм имитации отжига

```
Simulated-Annealing()
   Создаем первоначальное решение S
   Инициализируем температуру t
   repeat
      for i = 1 to iteration-length do
         Генерируем произвольный переход из S в Si
         If (C(S) > C(Si)) then S = St
         else if (e^((C(S)-C(Si))/(k·t)) > random[0,1)) then S = Si
      Понижаем температуру t
   until (больше нет изменений в C(S))
   Return S
```

Подведение итогов

Имитация отжига является эффективным средством, потому что она больше времени уделяет "хорошим" элементам пространства решений, чем "плохим", а также потому, что она не зацикливается на одном локальном оптимальном решении.

Так же, как и в случае с локальным поиском, представление задачи состоит из представления пространства решений и задания несложной функции стоимости $C(s)$ для определения качества конкретного решения. Новым компонентом является *график охлаждения* (cooling schedule), параметры которого управляют вероятностью приемлемости "плохого" перехода в зависимости от времени.

В начале поиска мы стремимся использовать фактор случайности, чтобы исследовать пространство поиска, поэтому вероятность приемлемости перехода в низкоэнергетиче-

ское состояние должна быть высокой. В процессе поиска мы стремимся ограничить переходы переходами к локальным улучшениям и оптимизациям. График охлаждения можно регулировать с помощью следующих параметров:

- первоначальная температура системы. Обычно $t_1 = 1$;
- функция понижения температуры. Обычно $t_k = \alpha \cdot t_{k-1}$, где $0{,}8 \le \alpha \le 0{,}99$. Это означает экспоненциальное понижение температуры, а не линейное;
- количество итераций перед понижением температуры. Обычно разрешается провести от 100 до 1 000 итераций;
- критерии приемлемости. В типичном случае приемлем любой переход от состояния S_i к состоянию S_{i+1}, если $C(S_{i+1}) < C(S_i)$, а также отрицательный переход, если

$$e^{-\frac{(C(s_i)-C(s_{i+1}))}{k \cdot t_i}} \ge r,$$

  где r — случайное число в диапазоне $0 \le r < 1$. Константа k нормализует функцию стоимости, чтобы при начальной температуре принимались почти все переходы;
- критерии остановки. Если за последнюю итерацию (или несколько последних итераций) значение текущего решения не изменилось или не улучшилось, то поиск прекращается и выводится текущее решение.

Создание правильного графика охлаждения фактически выполняется методом проб и ошибок в процессе подстановки разных значений констант и наблюдения за результатами. Я рекомендую начинать работу с методом имитации отжига с ознакомления с уже существующими его реализациями. В частности, мою реализацию этого метода можно найти на сайте **http://www.algorist.com**, а другие реализации — в *разделе 13.5*.

Сравните профили выполнения всех трех рассмотренных эвристических алгоритмов. Облако точек решений методом произвольной выборки существенно хуже, чем множество решений, полученных другими эвристическими методами. Очевидно, что полосы решений, полученные методом восхождения по выпуклой поверхности, намного лучше.

Но самым лучшим изо всех является профиль выполнения алгоритма имитации отжига, показанный на рис. 7.10.

Все три прогона алгоритма имитации отжига дают намного лучшие решения, чем самое лучшее решение метода восхождения по выпуклой поверхности. Более того, чтобы получить максимальное улучшение результата, требуется сравнительно небольшое количество итераций, что видно по трем резким переходам к оптимальному решению.

При том же количестве выборок, что и для других методов (1 500 000), метод имитации отжига выдал решение стоимостью 36 617,4 мили, что всего лишь на 9,2% больше оптимального. Если вы готовы подождать несколько минут, то можно получить еще более качественное решение. В частности, выполнение 5 000 000 итераций понижает стоимость решения до 34 254,9 мили, что на 2,2% выше оптимального. Но дальнейшее увеличение количества итераций до 10 000 000 не принесло последующего улучшения результатов.

При умелом обращении самые лучшие эвристические алгоритмы, настроенные под решение задачи коммивояжера, могут выдать чуть более удачное решение, чем метод

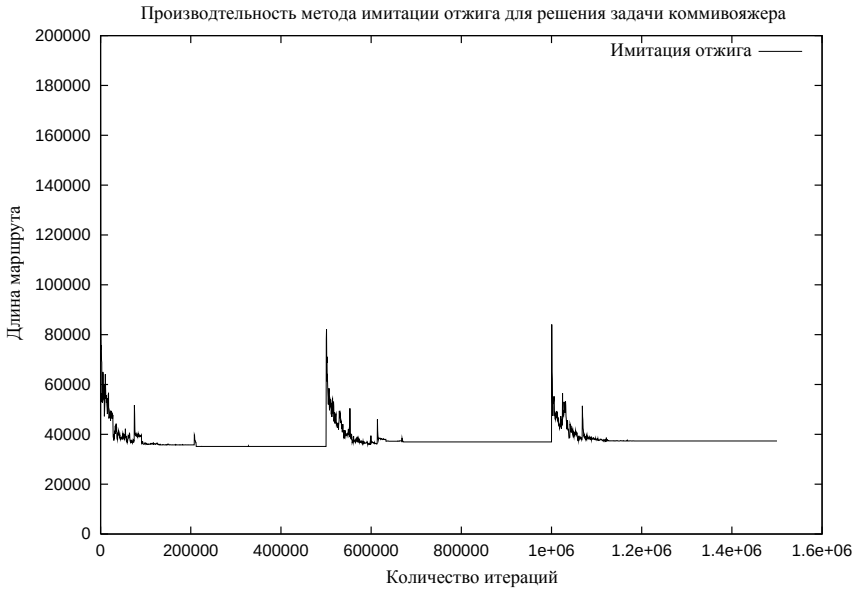


Рис. 7.10. Соотношения между временем поиска решений и их качеством для решения задачи коммивояжера методом им итации отжига

имитации отжига. Однако метод имитации отжига работает превосходно, не требуя специальной настройки. Я предпочитаю пользоваться им.

Реализация

Реализация эвристического алгоритма имитации отжига показана в листинге 7.18.

Листинг 7.18. Реализация метода имитации отжига

```
anneal(tsp_instance *t, tsp_solution *s)
{
   int i1, i2;           /* Пара элементов для обмена местами*/
   int i,j;              /* Счетчики */
   double temperature;   /* Текущая температура системы */
}  double current_value; /* Значение текущего состояния */
   double start_value;   /* Значение в начале цикла */
   double delta;         /* Значение после обмена */
   double merit, flip;   /* Условия принятия обмена*/
   double exponent;      /* Показатель степени для функции
                            энерг. состояния*/
   double random_float();
   double solution_cost(), transition();
   temperature = INITIAL_TEMPERATURE;
   initialize_solution(t->n, s);
   current_value = solution_cost(s, t);
   for (i=1; i<=COOLING_STEPS; i++) {
      temperature *= COOLING_FRACTION;
      start_value = current_value;
```

```
      for (j=1; j<=STEPS_PER_TEMP; j++) {
         /* Выбираем индексы элементов для обмена местами */
         i1 = random_int(1, t->n);
         i2 = random_int(1, t->n);
         flip = random_float ( 0 , 1 ) ;
         delta = transition(s,t,i1,i2);
         exponent = (-delta/current_value)/(K*temperature);
         merit = pow(E, exponent);
         if (delta < 0)              /* Принять удачный результат */
            current_value = current_value+delta;
         else { if (merit > flip)  /* Принять неудачный результат */
            current_value = current_value+delta;
         else                        /* Отклонить */
            transition(s,t,i1,i2) ;
         }
      }

      /* Восстанавливаем температуру в случае успеха */
      if ((current_value-start_value) < 0.0)
         temperature = temperature/COOLING_FRACTION;
   }
}
```

7.5.4. Применение метода имитации отжига

Теперь рассмотрим несколько примеров, демонстрирующих, как метод имитации отжига можно использовать для решения реальных задач комбинаторного поиска.

Задача максимального разреза

В задаче максимального разреза требуется разделить вершины взвешенного графа G на множества V_1 и V_2 таким образом, чтобы максимизировать в каждом множестве вес (или количество) ребер с одной вершиной. Для графов, представляющих электронные схемы, максимальный разрез графа определяет наибольший поток данных, который может одновременно протекать в схеме. Задача максимального разреза является NP-полной (см. *раздел 16.6*).

Как можно сформулировать задачу максимального разреза для решения методом имитации отжига? Пространство решений состоит из всех 2^{n-1} возможных разбиений вершин. Мы получаем двойную экономию по всем подмножествам вершин, т. к. можно полагать, что вершина v_1 зафиксирована на левой стороне разбиения. Подмножество вершин, сопровождающее эту вершину, можно представить с помощью битового вектора. Стоимостью решения является сумма весов разреза в текущей конфигурации. Механизм естественного перехода выбирает произвольным образом одну вершину и перемещает ее в другую часть разбиения, просто изменив на обратное значение соответствующего бита в битовом векторе. Изменение в функции стоимости составит разность между весом старых соседей вершины и весом ее новых соседей. Она вычисляется за время, пропорциональное степени вершины.

На практике именно к такому простому и естественному моделированию следует стремиться при поиске эвристического алгоритма.

Независимое множество

Независимым множеством графа G называется подмножество вершин S, не содержащее ребер, у которых обе конечные точки являются членами S. Максимальным независимым множеством графа является такой наибольший пустой порожденный подграф. Задача поиска независимых множеств возникает в задачах рассеивания, связанных с календарным планированием и теорией шифрования (см. *раздел 16.2*).

Естественное пространство состояний для решения задачи методом имитации отжига включает в себя все 2^n подмножества вершин, представленные в виде битового вектора. Так же, как и в случае с максимальным разрезом, простой механизм переходов добавляет или удаляет одну вершину из множества S.

Одной из наиболее естественных функций стоимости для подмножества S будет функция, возвращающая 0, если S содержит ребро, и $|S|$, в случае действительно независимого множества. Эта функция гарантирует, что мы непрерывно приближаемся к получению независимого множества. Но это условие настолько жесткое, что существует большая вероятность остаться в пределах лишь небольшой части возможного пространства поиска. Бóльшую степень гибкости в доступе к пространству поиска и ускорение работы функции стоимости можно получить, разрешив непустые графы на ранних этапах охлаждения. На практике будет лучше использовать функцию стоимости наподобие $C(S) = |S| - \lambda \cdot m_S/T$, где λ является постоянной, T представляет температуру, а m_S — количество ребер в подграфе, порожденным S. Зависимость $C(S)$ от T обеспечивает ускорение вытеснения ребер по мере охлаждения системы.

Размещение компонентов на печатной плате

Задача разработки печатных плат заключается в размещении на них должным образом компонентов, обычно интегральных схем. Заданными критериями компоновки могут быть минимизация площади или отношения длины платы к ее ширине, чтобы она помещалась в отведенное место, и минимизация длины соединяющих дорожек. Компоновка печатных плат является типичным примером запутанной задачи оптимизации со многими критериями. Для решения таких задач идеально подходит метод имитации отжига.

При формальной постановке задается коллекция прямоугольных модулей $r_1, \ldots, r_n$ с соответствующими размерами $h_i \times l_i$. Кроме этого, для каждой пары модулей (r_i, r_j) задается количество соединяющих их дорожек w_{ij}. Требуется найти такое размещение прямоугольников, которое минимизирует площадь печатной платы и длину соединяющих дорожек, при условии, что прямоугольники не могут накладываться, частично или полностью, друг на друга.

Пространство состояний для данной задачи должно описывать расположение каждого прямоугольника. Чтобы сделать задачу дискретной, можно наложить ограничение, при котором прямоугольники могут размещаться только на вершинах решетки целых чисел. Подходящим механизмом переходов может быть перемещение одного прямоугольника в другое место или обмен местами двух прямоугольников. Естественной функцией стоимости будет

$$C(S) = \lambda_{площадь}(S_{высота} \cdot S_{ширина}) + \sum_{i=1}^{n} \sum_{j=1}^{n} (\lambda_{дорожка} w_{ij} d_{ij} + \lambda_{наложение}(r_i \cap r_j)),$$

где $\lambda_{площадь}$, $\lambda_{дорожка}$ и $\lambda_{положение}$ являются постоянными, представляющими влияние этих факторов на функцию стоимости. По всей видимости, значение $\lambda_{положение}$ должно быть обратно пропорционально температуре, чтобы после предварительного размещения прямоугольников их позиции корректировались во избежание наложения прямоугольников друг на друга.

Подведение итогов

Метод имитации отжига является простым, но эффективным методом получения хотя и не оптимальных, но достаточно хороших решений задач комбинаторного поиска.

7.6. История из жизни. Только это не радио

— Считайте, что это радио, — мой собеседник тихо рассмеялся. — Только это не радио.

Меня срочно доставили на корпоративном реактивном самолете в научно-исследовательский центр одной большой компании, расположенный где-то к востоку от штата Калифорния. Они были настолько озабочены сохранением секретности, что мне даже никогда ни пришлось увидеть разрабатываемое ими устройство. Тем не менее, они отличнейшим образом абстрагировали задачу.

Задача была связана с технологией производства, называемой *селективной сборкой*. Эли Уитни (Eli Whitney) считается изобретателем системы *взаимозаменяемых компонентов*, которая сыграла важную роль в промышленной революции. Принцип взаимозаменяемости состоит в изготовлении деталей механизма с определенной степенью точности, называемой допуском на обработку, что позволяет использовать любую комбинацию деталей для сборки механизма. Таким образом существенно ускоряется процесс производства, т. к. детали теперь можно просто складывать вместе, а не подгонять каждую индивидуально с помощью напильника. Взаимозаменяемость также значительно облегчила ремонт изделий, позволяя выполнять замену вышедших из строя деталей без особых трудностей. Все это было очень хорошо, за исключением одного обстоятельства.

В частности, если размеры детали слегка не соответствовали допускам, то такую деталь нельзя было использовать. Однако нашелся сообразительный сотрудник, который предложил использовать детали с нарушениями в допусках совместно с деталями, изготовленными точно по требуемому размеру. Таким образом, использование плохих деталей совместно с хорошими дает приемлемый результат. В этом и заключается суть *селективной сборки*.

— Каждый прибор состоит из n деталей разных типов, — продолжал представитель компании. — Для i-го типа детали (скажем, фланцевой прокладки) имеется s_i экземпляров этой детали, для каждой из которых указана степень отклонения от эталонного размера. Нам необходимо подобрать детали друг к другу таким образом, чтобы получить максимально возможное количество работающих приборов.

— Предположим, каждый прибор состоит из трех деталей, а сумма отклонений от нормы всех деталей работающего прибора не должна превышать 50. Умело распределяя хорошие и плохие детали в каждом устройстве, мы можем использовать все детали и получить три работающих прибора.

Данная ситуация иллюстрируется на рис. 7.11.

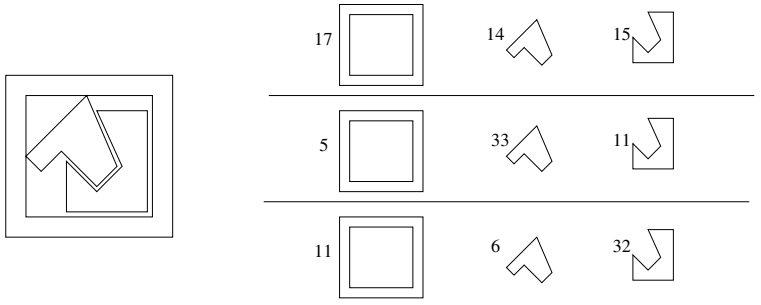


Рис. 7.11. Распределение деталей между тремя приборами, чтобы сумма отклонений для каждого не превышала 50

Я немного поразмышлял над задачей. Проще всего было бы использовать для сборки каждого прибора самые лучшие оставшиеся детали каждого типа, повторяя процесс до тех пор, пока собранный таким способом прибор не будет работать. Но таким образом мы получаем небольшое количество приборов с большим разбросом качества, в то время, как компании требуется максимальное количество приборов высокого качества.

Целью было совместить хорошие и плохие детали друг с другом таким образом, чтобы общее качество собранного устройства было бы приемлемым. Определенно, задача выглядела как *паросочетание* в графах (см. *раздел 15.6*). Допустим, мы создадим граф, в котором вершины представляют экземпляры деталей, и соединим ребрами все пары экземпляров с допустимой общей погрешностью в размерах. В паросочетании в графах мы ищем наибольшее количество ребер, в котором никакие два ребра не опираются на одну и ту же вершину. Это аналогично максимальному количеству сборок из двух деталей, которые можно получить из данного набора деталей.

— Вашу задачу можно решить методом паросочетания, — объявил я. — При условии, что все приборы состоят из двух деталей.

Мое заявление было встречено общим молчанием, за которым последовал дружный смех. — *Все* знают, что в приборе более двух деталей.

Таким образом, данный алгоритмический подход был отвергнут. Расширение задачи до сопоставления более чем двух деталей превращало ее в задачу паросочетания в гиперграфах[1], которая является NP-полной. Более того, только само время построения графа может быть экспоненциально зависимым от количества типов деталей, т. к. каждое возможное гиперребро (сборку) нужно будет создавать явным образом.

Пришлось начинать разработку алгоритма сначала. Итак, нужно собрать детали таким образом, чтобы общая сумма отклонений от нормы не превышала определенного допустимого значения. Это выглядело, как *задача упаковки контейнеров*. В задаче этого типа (см. *раздел 17.9*) требуется разложить набор элементов разного размера в наименьшее количество контейнеров с ограниченной емкостью k. В данной ситуации контейнеры представляли сборки, каждая из которых допускала общую сумму отклонений

[1] Гиперграф содержит ребра, каждое из которых может иметь более двух вершин. Их можно представить в виде общих коллекций вершин.

от нормы $\leq k$. Пакуемые элементы представляли отдельные детали, размер которых отражал качество изготовления.

Однако полученная задача не является чистой задачей разложения по контейнерам в силу неоднотипности деталей. Данное приложение накладывало ограничение на допустимое содержимое контейнеров. Требование создать наибольшее количество приборов означало, что нужно было найти метод разложения элементов, который максимизирует количество контейнеров, содержащих ровно одну деталь каждого типа.

Задача разложения по контейнерам является NP-полной, однако естественно подходит для решения методом эвристического поиска. Пространство решений состоит из вариантов разложения деталей по контейнерам. Для каждого контейнера мы сначала выбираем по произвольной детали каждого типа, чтобы получить начальную конфигурацию для поиска.

Потом мы выполняем локальный поиск, перемещая детали из одного контейнера в другой. Можно было бы перемещать по одной детали, но будет более эффективным обменивать детали определенного типа между двумя произвольно выбранными контейнерами. При таком обмене оба контейнера остаются полностью собранными приборами, предположительно с более приемлемым значением общего отклонения от нормы. Для операции обмена требовалось три произвольных целых числа — одно для выбора типа детали (в диапазоне от 1 до m) и два для выбора контейнеров, между которыми нужно выполнять обмен (например, в диапазоне от 1 до b).

Ключевым решением был выбор функции стоимости. Для каждой сборки был установлен жесткий общий предел отклонений от нормы, равный k. Но что может быть наилучшим способом оценки *нескольких* сборок? Можно было бы в качестве общей оценки просто возвращать количество приемлемых сборок — целое число в диапазоне от 1 до b. Хотя эта величина и была той, которую мы хотели оптимизировать, она не была чувствительна к продвижению в направлении правильного решения. Допустим, что в результате одного из обменов деталей между сборками допустимое отклонение одной из неработающих сборок стало намного ближе к пределу отклонений k для сборки. Это была бы значительно более удачная отправная точка для дальнейшего поиска решения, чем первоначальная.

В итоге, я остановился на следующей функции стоимости. Каждой работающей сборке я давал оценку в 1 пункт, а каждой неработающей — значительно меньшую оценку, в зависимости от того, насколько близка она была к предельному значению k. Оценка неработающей сборки понижалась экспоненциально, в зависимости от того, насколько ее общее отклонение от нормы превышало k. Таким образом, оптимизатор будет пытаться максимизировать количество работающих приборов, после чего станет подгонять очередную сборку как можно ближе к пределу.

Я воплотил этот алгоритм в коде, после чего обработал с его помощью набор деталей, взятых непосредственно из производственного цеха. Оказалось, что данные приборы состоят из 8 деталей важных типов. Детали некоторых типов были более дорогими, чем другие, поэтому для них было меньше рассматриваемых вариантов. Для детали с наименьшими разрешенными отклонениями было предоставлено только восемь экземпляров, поэтому из данного набора деталей можно было собрать только восемь приборов.

Я наблюдал за работой программы имитации отжига. Первые четыре сборки она выдала сразу же без особых усилий, и лишь после этого поиск немного замедлился и сборки 5 и 6 были предоставлены с небольшой задержкой. Потом наступила пауза, после которой была выдана седьмая сборка. Но, несмотря на все свои усилия, программа не смогла сложить восемь работающих приборов, прежде чем мне надоело наблюдать за ее действиями.

Я позвонил в компанию и хотел признаться в поражении, но они и слышать ничего об этом не хотели. Оказалось, что самостоятельно они смогли получить только шесть работающих приборов, так что мой результат был значительным прогрессом!

7.7. История из жизни. Отжиг массивов

В *разделе 3.9* рассказывалось, как мы использовали структуры данных высшего уровня для эмулирования нового метода секвенирования ДНК. Для нашего метода, интерактивного секвенирования методом гибридизации, нужно было создавать по требованию массивы специфичных олигонуклеотидов.

Наш метод заинтересовал одного биохимика из Оксфордского университета и, что более важно, в его лаборатории было необходимое оборудование для испытания этого метода. Автоматическая система подачи реактивов Southern Array Maker (производства компании Beckman Instruments) выкладывает дискретные последовательности олигонуклеотидов в виде шестидесяти четырех параллельных рядов на полипропиленовой подложке. Устройство создает массивы, добавляя отдельные элементы в каждую ячейку вдоль определенных рядов и столбцов массива. В табл. 7.2 показан пример создания массива всех $2^4 = 16$ пуриновых (*A* или *G*) 4-меров путем создания префиксов вдоль рядов и суффиксов вдоль столбцов.

Таблица 7.2. Массив всех пуриновых 4-меров

Префикс	Суффикс			
	AA	AG	GA	GG
AA	AAAA	AAAG	AAGA	AAGG
AG	AGAA	AGAG	AGGA	AGGG
GA	GAAA	GAAG	GAGA	GAGG
GG	GGAA	GGAG	GGGA	GGGG

Эта технология предоставляла идеальную среду для проверки возможности применения интерактивного секвенирования методом гибридизации в лаборатории, т. к. она позволяла создавать с помощью компьютера широкий набор массивов олигонуклеотидов.

Вот мы и должны были предоставить соответствующее программное обеспечение. Для создания сложных массивов требовалось решить трудную комбинаторную задачу. В качестве входа задавался набор n строк (представляющих олигонуклеотиды), из ко-

торых нужно было создать массив размером $m \times m$ (где $m = 64$ на устройстве Southern Array Maker). Нам нужно было запрограммировать создание рядов и столбцов, чтобы реализовать множество строк S. Мы доказали, что задача разработки плотных массивов является NP-полной, но это не имело большого значения, т. к. решать ее в любом случае пришлось мне и моему студенту Рики Брэдли (Ricky Bradley).

— Нам нужно будет использовать эвристический метод, — сказал я ему. — Как смоделировать эту задачу?

— Каждую строку можно разбить на составляющие ее префиксные и суффиксные пары. Например, строку *ACC* можно создать четырьмя разными способами: из префикса " (пустая строка) и суффикса *ACC*, префикса *A* и суффикса *CC*, префикса *AC* и суффикса *C* или префикса *ACC* и суффикса ". Нам нужно найти наименьший набор префиксов и суффиксов, которые совместно позволяют получить все данные строки, — сказал Рики.

— Хорошо. Это дает нам естественное представление для метода имитации отжига. Пространство состояний будет содержать все возможные подмножества префиксов и суффиксов. Естественные переходы между состояниями могут включать вставку или удаление строк из наших подмножеств или обмен пар местами.

— Какую функцию стоимости можно использовать? — спросил он.

— Нам требуется массив как можно меньшего размера, который охватывает все строки. Попробуем взять максимальное количество рядов (префиксов) или столбцов (суффиксов), используемых в нашем массиве, плюс те строки из множества S, которые еще не были задействованы.

Рики отправился на свое рабочее место и реализовал программу имитации отжига в соответствии с этими принципами. Программа выводила текущее состояние решения после каждого принятия перехода, и за ее работой было интересно наблюдать. Она быстро отсеяла ненужные префиксы и суффиксы и размер массива начал стремительно сокращаться. Но после нескольких сотен итераций прогресс начал замедляться. При переходе программа удаляла ненужный суффикс, выполняла вычисления в течение некоторого времени, а потом добавляла другой суффикс. После нескольких тысяч итераций не наблюдалось никакого реального улучшения.

— Кажется, что программа не может определить, когда она получает правильное решение, — предположил я. — Функция стоимости оценивает только минимизацию большей стороны массива. Давай добавим в нее выражения для оценки действий с другой стороной.

Рики внес соответствующие изменения в функцию стоимости, и мы снова запустили программу. На этот раз она уверенно обрабатывала и другую сторону. Более .rого, наши массивы стали выглядеть как тонкие прямоугольники, а не квадраты.

— Ладно. Давай добавим в функцию стоимости еще одно выражение, чтобы массивы стали квадратными.

Рики опять изменил программу. На этот раз массивы на выходе имели правильную форму и поиск двигался в нужном направлении. Но происходило это по-прежнему медленно.

— Многие операции вставки затрагивают недостаточное количество строк. Может быть, нам следует сделать уклон в пользу случайной выборки, чтобы важные префиксы и суффиксы выбирались чаще?

Рики внес очередные изменения. Теперь программа работала быстрее, но все равно иногда возникали паузы. Мы изменили график охлаждения. Результаты улучшились, но были ли они действительно хорошими? Не зная нижнего оптимального предела, нельзя было сказать, насколько удачным является наше решение. Мы продолжали настраивать разные аспекты нашей программы, пока все дальнейшие модификации не перестали приводить к улучшению.

Конечная версия программы улучшала первоначальный массив, используя следующие операции:

- *swap* — обменять префикс/суффикс в массиве с префиксом/суффиксом вне массива;
- *add* — добавить в массив случайный префикс/суффикс;
- *delete* — удалить из массива случайный префикс/суффикс;
- *useful add* — добавить в массив префикс/суффикс с наибольшей пригодностью;
- *useful delete* — удалить из массива префикс/суффикс с наименьшей пригодностью;
- *string add* — выбрать произвольную строку вне массива и добавить наиболее пригодный префикс и/или суффикс, чтобы покрыть эту строку.

Мы использовали стандартный график охлаждения с экспоненциально убывающей температурой (зависящей от размера задачи) и зависящим от температуры критерием Больцмана, чтобы выбирать состояния, имеющие более высокую стоимость. Наша конечная функция стоимости определялась таким образом:

$$cost = 2 \times max + min + \frac{(max - min)^2}{4} + 4(str_{total} - str_{in})$$

где max — максимальный размер чипа, min — минимальный размер чипа, $str_{total} = |S|$, а str_{in} — количество строк из множества S, находящихся в настоящее время в чипе.

Насколько хороший результат мы получили? На рис. 7.12 показано четыре этапа сжатия специального массива, состоящего из 5 716 однозначных 7-меров вируса иммунодефицита человека (ВИЧ).

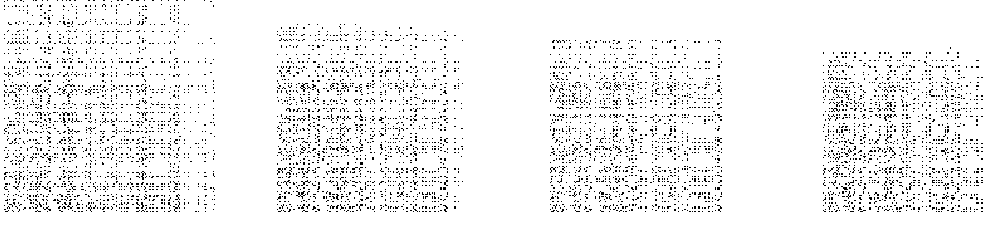


Рис. 7.12. Сжатие массива ВИЧ методом имитации отжига после 0,500, 1000 и 5750 итераций

Черные пикселы массива представляют первое появление 7-мера ВИЧ. Конечный размер чипа — 130×132, что является прогрессом по сравнению с первоначальным размером 192×192. Чтобы завершить оптимизацию, программе потребовалось около

15 минут, что было приемлемо для данного приложения. Насколько хорошим был этот результат? Так как имитация отжига — всего лишь эвристический метод, то мы, по сути, не знаем, насколько близким к оптимальному является наше решение. Лично я думаю, что результат довольно хороший, но не могу быть в этом полностью уверенным. Метод имитации отжига является хорошим инструментом для решения сложных задач оптимизации. Однако, если вы хотите получить отличные результаты, будьте готовы потратить больше времени на дополнительную настройку и оптимизацию программы, чем ушло на создание ее первоначального варианта. Это тяжелая работа, но зачастую у вас нет иного выхода.

7.8. Другие эвристические методы поиска

Для поиска хороших решений задач комбинаторной оптимизации было предложено несколько эвристических методов. Подобно методу имитации отжига, многие эвристические методы основаны на аналогии с реальными физическими процессами. Одними из популярных методов являются *генетические алгоритмы*, *нейронные сети* и *алгоритм муравейника*.

Интуитивные представления, лежащие в основе этих методов, привлекательны своей понятностью, но скептики считают такие подходы шаманством, основанным на красивых аналогиях с природными явлениями, а не на результатах исследования задач, достигнутых с помощью других методов.

Вопрос заключается не в том, возможно ли, приложив достаточные усилия, получить приемлемые решения задач с помощью этих методов. Ясно, что возможно. Но настоящий вопрос состоит в том, позволяют ли эти методы получить *лучшие* решения при *меньшей сложности реализации*, чем другие рассмотренные методы.

В общем, я так не думаю. Но в духе непредвзятого исследования мы кратко рассмотрим генетические алгоритмы, которые пользуются наибольшей популярностью. Более подробную информацию см. в *разделе "Замечания к главе"*.

Генетические алгоритмы

Идея генетических алгоритмов возникла как отражение теории эволюции и естественного отбора. Посредством естественного отбора организмы приспосабливаются к выживанию в определенной среде. В генетическом коде организма происходят случайные мутации, которые передаются его потомкам. Если данная мутация окажется полезной, то тогда у потомков будет больше шансов выжить. Если же мутация вредная, то вероятность выживания потомков и передачи этого качества по наследству уменьшается.

Генетические алгоритмы содержат "популяцию" кандидатов на решение данной задачи. Из этой популяции случайным образом выбираются элементы, которые "размножаются" так, что в потомке комбинируются свойства двух родителей. Вероятность выбора элемента для размножения основывается на его "физической форме", которой, по существу, является стоимость предоставляемого им решения. Элементы, имеющие плохую физическую форму, вымирают, уступая место потомкам элементов, пребывающих в более хорошей форме.

Идея генетических алгоритмов чрезвычайно привлекательна. Но эти алгоритмы не могут работать так эффективно над практическими задачами комбинаторной оптимизации, как метод имитации отжига. На то есть две причины. Во-первых, моделирование приложений посредством генетических операторов, таких как мутация и пересечение на битовых строках, является в высшей степени неестественным. Псевдобиология вносит дополнительный уровень сложности в решаемую задачу. Во-вторых, решение нетривиальных задач посредством генетических алгоритмов занимает очень много времени. Операции пересечения и мутации обычно не используют структуры данных, специфичных для данной задачи, вследствие чего большинство переходов дает низкокачественные решения и процесс поиска наилучшего решения протекает медленно. В этом случае аналогия с эволюцией (которой для осуществления важных изменений требуются миллионы лет) оказывается уместной.

Чтобы отговорить вас от использования генетических алгоритмов в своих приложениях, мы не будем продолжать рассматривать их. Но если вы все же очень хотите поэкспериментировать с ними, то подсказки по их реализации можно найти в *разделе 13.5*.

Подведение итогов

Я лично *никогда* не сталкивался ни с одной задачей, для решения которой генетические алгоритмы оказались бы самым подходящим средством. Более того, я никогда не встречал никаких результатов вычислений, полученных посредством генетических алгоритмов, которые бы производили на меня положительное впечатление. Пользуйтесь методом имитации отжига для своих экспериментов с эвристическим поиском.

7.9. Параллельные алгоритмы

Ум хорошо, а два лучше, а, более обобщенно, n умов лучше, чем $n-1$ умов. Такой подход кажется легким способом решения трудных задач. В самом деле, для решения некоторых задач параллельные алгоритмы являются наиболее эффективным средством. Например, для реализации реалистичной анимации графические приложения высокого разрешения реального времени должны выводить около тридцати кадров в секунду. Единственным способом справиться с такой задачей может быть выделение отдельного процессора для обработки каждого кадра или для обработки одной части кадра. Большие системы линейных уравнений для научных целей регулярно решаются параллельным методом.

Но параллельные алгоритмы имеют несколько скрытых недостатков, о которых нужно знать.

- *Потенциальный выигрыш часто невелик.* Допустим, что у вас имеется исключительный доступ к двадцати процессорам. Теоретически, используя все эти процессоры, можно в двадцать раз повысить скорость работы самой быстрой последовательной программы. Это, конечно же, прекрасно, но, возможно, производительность можно повысить, использовав более удачный последовательный алгоритм. Время, требуемое на переработку кода для исполнения на параллельных процессорах, возможно, лучше потратить на улучшение последовательной версии програм-

мы. Также нужно иметь в виду, что средства отладки и повышения производительности кода, такие как профилировщики, лучше работают на обычных компьютерах, чем на многопроцессорных.

- *Простое ускорение работы еще ничего не означает.* Допустим, что ваша параллельная программа работает в 20 раз быстрее на 20-процессорной машине, чем на компьютере с одним процессором. Это просто замечательно, не так ли? Конечно же, если вы всегда получаете линейное повышение скорости и имеете доступ к сколь угодно большому количеству процессоров, то со временем вы перекроете результаты любого последовательного алгоритма. Но тщательно разработанный последовательный алгоритм часто может оказаться эффективнее параллелизируемого кода, выполняемого на типичном многопроцессорном компьютере. Скорее всего, причиной низкой производительности вашей параллельной программы на однопроцессорной машине является недостаточно хороший последовательный алгоритм. Поэтому измерение ускорения работы такой программы при использовании нескольких процессоров является некорректной оценкой преимуществ параллелизма.

  Классическим примером такой ситуации является минимаксный алгоритм поиска в дереве игры, применяющийся в компьютерных программах игры в шахматы. Поиск методом исчерпывающего перебора в этом дереве удивительно легко поддается параллелизации: просто для каждого поддерева выделяется отдельный процессор. Но при таком подходе большая часть работы этих процессоров выполняется впустую, т. к. одни и те же позиции исследуются на нескольких процессорах. Использование же более интеллектуального алгоритма альфа-бета-отсечений может с легкостью уменьшить объем работы на 99,99%, по сравнению с чем любые достоинства поиска методом исчерпывающего перебора на параллельных процессорах выглядят ничтожными. Алгоритм альфа-бета-отсечений можно распараллелить, но это задача не из легких. Кроме этого, увеличение количества используемых процессоров дает неожиданно малое повышение производительности.

- *Параллельные алгоритмы плохо поддаются отладке.* Если только вашу задачу невозможно разбить на несколько независимых подзадач, то при ее исполнении на нескольких процессорах разные процессоры должны взаимодействовать друг с другом, чтобы выдать правильный конечный результат. К сожалению, вследствие недетерминистического характера этого взаимодействия параллельные программы трудно отлаживать. Пожалуй, самым лучшим примером является шахматный суперкомпьютер Deep Blue. Хотя последняя версия этого компьютера в конце концов победила чемпиона мира по шахматам Гарри Каспарова, в предыдущих соревнованиях машина терпела поражения из-за программных ошибок, по большому счету вызванных обширным параллелизмом.

Я рекомендую рассматривать параллельные вычисления только в тех случаях, когда все другие попытки решить задачу последовательным способом оказываются слишком медленными. Но даже в таких случаях я бы ограничился использованием алгоритмов, которые параллелизируют задачу, разбивая ее на несколько подзадач, для исполнения которых разные процессоры не должны общаться друг с другом, кроме как для объединения конечных результатов. Такой примитивный параллелизм может быть достаточно простым как для реализации, так и для отладки, т. к. его конечным результатом,

по сути, является хорошая последовательная реализация. Но даже здесь есть свои подводные камни.

7.10. История из жизни. "Торопиться в никуда"

В *разделе 2.8* я описал наши усилия по созданию быстрой программы для исследования проблемы Уоринга. Код был достаточно быстрым и мог решить задачу за несколько недель, работая в фоновом режиме на настольном компьютере. Но моему коллеге этот вариант не понравился.

— Давайте запустим его на многопроцессорном компьютере, — предложил он. — В конце концов, в этом коде внешний цикл выполняет одни и те же вычисления для каждого целого числа от 1 до 1 000 000 000. Я могу разбить этот диапазон чисел на одинаковые интервалы и обработать каждый из них на отдельном процессоре. Увидите, как это будет легко.

Он приступил к работе по выполнению нашей программы на компьютере Intel IPSC-860 с 32 узлами, каждый из которых имел свои собственные 16 Мбайт памяти — (очень мощная машина для того времени). Но в течение нескольких следующих недель я постоянно получал от него известия о проблемах. Например:

- программа работает прекрасно, но только прошлой ночью один процессор вышел из строя;
- все шло хорошо, но компьютер случайно перезагрузили, так что мы потеряли все результаты;
- согласно корпоративным правилам никто ни при каких обстоятельствах не может использовать все процессоры более чем 13 часов.

Но, в конце концов, обстоятельства сложились благоприятно. Он дождался стабильной работы компьютера, захватил 16 процессоров (половину компьютера), разбил целые числа от 1 до 1 000 000 000 на 16 одинаковых диапазонов и запустил вычисление каждого диапазона на отдельном процессоре. Следующий день он провел, отбивая атаки разъяренных пользователей, которые не могли попасть на компьютер из-за его программы. Как только первый процессор закончил обработку своего диапазона (числа от 1 до 62 500 000), он объявил негодующей толпе, что скоро и остальные процессоры закончат свою работу.

Но этого не произошло. Наш коллега не принял во внимание то обстоятельство, что время для обработки каждого целого числа увеличивалось пропорционально увеличению чисел. В конце концов, проверка представимости числа 1 000 000 000 в виде суммы трех пирамидальных чисел требует больше времени, чем такая же проверка для числа 100. Таким образом, последующие процессоры сообщали об окончании работы через все более длительные интервалы времени. Из-за особенностей устройства суперкомпьютера освободившиеся процессоры нельзя было задействовать снова, пока не была завершена вся работа. В конце концов, половина машины и большинство ее пользователей оказались в заложниках у одного, последнего процессора.

Какие выводы можно сделать из этой истории? Если вы собираетесь распараллелить вычисления, позаботьтесь о том, чтобы должным образом распределить работу между

процессорами. В приведенном примере правильный баланс нагрузки, достигнутый посредством несложных вычислений или с помощью алгоритма разбиения, рассматриваемого в *разделе 8.5*, мог бы значительно сократить время исполнения задачи, а также время, в течение которого нашему коллеге пришлось выносить нападки других пользователей.

Замечания к главе

Обсуждение перебора с возвратом в значительной степени основано на моей книге "Programming Challenges" [SR03]. В частности, рассмотренная здесь процедура перебора с возвратом является обобщенной версией процедуры, представленной в *главе 8* этой книги. В данной главе также имеется мое решение знаменитой задачи о восьми ферзях, где требуется найти все расположения на шахматной доске восьми ферзей, ни один из которых не находится под ударом любого другого.

Метод имитации отжига был первоначально рассмотрен в статье [KGV83], в которой также обсуждалось его применение для решения задачи размещения на печатных платах сверхбольших интегральных схем. Приложения в *разделе 7.5.4* основаны на материале из книги [AK89].

В представленных здесь эвристических решениях задачи коммивояжера в качестве локальной операции используется обмен вершин. В действительности, намного более мощной операцией является обмен ребер. При каждой операции обмениваются максимум два ребра в маршруте, по сравнению с четырьмя ребрами при обмене местами вершин. Это повышает вероятность локального улучшения. Но чтобы эффективно поддерживать порядок полученного маршрута, требуются более сложные структуры данных, которые рассматриваются в книге [FJMO93].

Разные эвристические методы поиска подробно представлены в книге [AL97], которую я рекомендую всем, кто хочет углубить свои знания в области эвристических методов поиска. В этой книге описан алгоритм *поиска с запретами* (tabu search), который является разновидностью метода имитации отжига, использующего дополнительные структуры данных, чтобы избежать переходов в недавно рассмотренные состояния. Алгоритм муравейника рассматривается в книге [DT04]. Более доброжелательную точку зрения на генетические и им подобные алгоритмы, чем та, что представлена в этой главе, см. в книге [MFOO].

Подробную информацию по комбинаторному поиску оптимальных позиций на шахматной доске можно найти в статье [RHS89]. Наша работа по использованию метода имитации отжига для сжатия массивов ДНК была представлена в журнале [BS97]. Дополнительную информацию по селективной сборке см. в журнале [Pug86] и журнале [CGJ98]. Наша работа по параллельным вычислениям с пирамидальными числами была представлена в журнале [DY94].

7.11. Упражнения

Перебор с возвратом

1. [3] *Беспорядком* (derangement) называется такая перестановка p множества $\{1, ..., n\}$, в которой ни один элемент на находится на своем правильном месте, т. е., $p_i \neq i$ для всех

$1 \le i \le n$. Разработайте программу перебора с возвратом с применением отсечений для создания всех беспорядочных перестановок n элементов.

2. [4] *Мультимножества* (multiset) могут содержать повторяющиеся элементы. Таким образом, мультимножество из n элементов может иметь меньше, чем $n!$ разных перестановок. Например, мультимножество {1, 1, 2, 2} имеет только шесть разных перестановок: {1, 1, 2, 2}, {1, 2, 1, 2}, {1, 2, 2, 1}, {2, 1, 1, 2}, {2, 1, 2, 1} и {2, 2, 1, 1}. Разработайте и реализуйте алгоритм для создания всех перестановок мультимножества.

3. [5] Разработайте и реализуйте алгоритм для проверки, являются ли два графа изоморфными по отношению друг к другу. Задача изоморфизма графов рассматривается в *разделе 16.9*. При правильном отсечении можно без проблем выполнять проверку графов, содержащих сотни вершин.

4. [5] Анаграммой называется перестановка букв слова или фразы таким образом, чтобы получилось другое слово или фраза. Иногда результаты таких перестановок поражают. Например, фраза MANY VOTED BUSH RETIRED (многие проголосовали за отставку Буша) является анаграммой фразы TUESDAY NOVEMBER, THIRD (вторник, третье ноября)[1]. Разработайте и реализуйте алгоритм создания анаграмм, используя комбинаторный поиск и словарь.

5. [8] Разработайте и реализуйте алгоритм для решения задачи изоморфизма подграфов. Даны графы G и H. Существует ли такой подграф H' графа H, для которого граф G является изоморфным графу H'? Как ваша программа работает в таких особых случаях изоморфизма подграфов, как гамильтонов цикл, клика, независимое множество?

6. [8] В проекте по реконструкции автострады дано $n(n-1)/2$ упорядоченных по длине расстояний. Задача заключается в том, чтобы найти положения n точек на линии, которые порождают эти расстояния. Например, расстояния {1, 2, 3, 4, 5, 6} можно определить, расположив вторую точку на расстоянии 1 единицы от первой, третью — на расстоянии 3 единиц от второй, а четвертую — на расстоянии 2 единиц от третьей. Разработайте и реализуйте алгоритм для вывода всех решений этой задачи. Чтобы минимизировать поиск, используйте по мере возможности аддитивные ограничения. При правильном отсечении можно без проблем решать задачи, содержащие сотни узлов.

Комбинаторная оптимизация

Для каждой из приведенных далее задач реализуйте программу комбинаторного поиска для оптимального решения входного экземпляра небольшого размера и/или разработайте и реализуйте эвристический алгоритм имитации отжига для получения приемлемых решений. Дайте оценку практической работе вашей программы.

7. [5] Разработайте и реализуйте алгоритм для решения задачи минимизации ширины полос, рассматриваемой в *разделе 13.2*.

8. [5] Разработайте и реализуйте алгоритм для решения задачи максимальной приемлемости, рассматриваемой в *разделе 14.10*.

9. [5] Разработайте и реализуйте алгоритм для решения задачи поиска максимальной клики, рассматриваемой в *разделе 16.1*.

---

[1] День президентских выборов в США в 1992 г., когда действующий президент Буш проиграл Биллу Клинтону. — *Прим. перев.*

10. [5] Разработайте и реализуйте алгоритм для решения задачи минимальной вершинной раскраски, рассматриваемой в *разделе 16.7*.
11. [5] Разработайте и реализуйте алгоритм для решения задачи реберной раскраски, рассматриваемой в *разделе 16.8*.
12. [5] Разработайте и реализуйте алгоритм для решения задачи поиска разрывающего множества вершин, рассматриваемой в *разделе 16.11*.
13. [5] Разработайте и реализуйте алгоритм для решения задачи покрытия множества, рассматриваемой в *разделе 18.1*.

Задачи, предлагаемые на собеседовании

14. [4] Напишите функцию поиска всех перестановок букв в данной строке.
15. [4] Реализуйте алгоритм перечисления всех k-элементных подмножеств множества, содержащего n элементов.
16. [5] Анаграммой называется перестановка букв слова или фразы таким образом, чтобы получилось другое слово или фраза. Например, анаграммой строки *Steven Skiena* является строка *Vainest Knees*. Предложите алгоритм для создания всех анаграмм данной строки.
17. [5] Каждая кнопка телефонного номеронабирателя содержит несколько букв. Напишите программу для генерирования всех возможных слов, которые можно получить из данной последовательности нажатых кнопок (например, 145345).
18. [7] Имеется пустая комната и n человек снаружи. На каждом этапе можно впустить одного человека в комнату или выпустить одного человека из комнаты. Можете ли вы упорядочить последовательность из $2n$ этапов таким образом, чтобы каждая возможная комбинация людей возникла только один раз?
19. [4] Используя генератор случайных чисел, который генерирует случайные целые числа в диапазоне от 0 до 4 с одинаковой вероятностью, напишите генератор случайных чисел (rng07), который генерирует целые числа в диапазоне от 0 до 7 с одинаковой вероятностью. Укажите ожидаемое количество вызовов функции rng04 для каждого вызова функции rng07.

Задачи по программированию

Эти задачи доступны на сайтах **http://www.programming-challenges.com** и **http://uva.onlinejudge.org**.

1. Little Bishops. 110801/861.
2. 15-Puzzle Problem. 110802/10181.
3. Tug of War. 110805/10032.
4. Color Hash. 110807/704.

ГЛАВА 8

Динамическое программирование

Наиболее трудными алгоритмическими задачами являются задачи оптимизации, в которых требуется найти решение, максимизирующее или минимизирующее определенную функцию. Классическим примером задачи оптимизации является задача коммивояжера, в которой требуется найти маршрут с минимальной стоимостью для посещения всех вершин графа. Как мы видели в *главе 1*, для решения задачи коммивояжера можно легко предложить несколько алгоритмов, которые выдают кажущиеся удовлетворительными решения, но *не всегда* выдают маршрут с минимальной стоимостью.

Для алгоритмов задач оптимизации требуется доказательство, что они всегда возвращают наилучшее возможное решение. "Жадные" алгоритмы, которые принимают наилучшее локальное решение на каждом шаге, обычно эффективны, но не гарантируют глобальное оптимальное решение. Алгоритмы поиска методом исчерпывающего перебора всегда выдают оптимальный результат, но обычно временная сложность таких алгоритмов чрезмерно высока.

Динамическое программирование сочетает лучшие возможности обоих подходов. Этот метод предоставляет возможность разрабатывать алгоритмы специального назначения, которые систематически исследуют все возможности (таким образом гарантируя правильность решения) и в то же самое время сохраняют ранее полученные промежуточные результаты (таким образом обеспечивая эффективность работы). Сохранение *последствий* всех возможных решений и систематическое использование этой информации минимизируют общий объем работы.

Если вы понимаете суть динамического программирования, эта технология разработки алгоритмов, возможно, является для вас самой удобной для практического применения. Я считаю, что алгоритмы динамического программирования часто легче разработать заново, чем искать их готовую реализацию в какой-либо книге. Однако, пока вы не понимаете динамическое программирование, оно кажется вам каким-то шаманством.

Динамическое программирование — это технология для эффективной реализации рекурсивных алгоритмов посредством сохранения промежуточных результатов. Секрет ее применения заключается в определении, выдает ли простой рекурсивный алгоритм одинаковые результаты для одинаковых подзадач. Если выдает, то вместо повторения вычислений ответ каждой подзадачи можно сохранять в таблице для использования в дальнейшем, что дает возможность получить эффективный алгоритм. Начинаем разработку с определения и отладки рекурсивного алгоритма. Только добившись правильной работы нашего рекурсивного алгоритма, мы переходим к поиску мер по ускорению его работы, сохраняя результаты в матрице.

Динамическое программирование обычно является подходящим методом решения задач оптимизации в случае комбинаторных объектов, которые имеют естественный по-

рядок организации компонентов слева направо. К таким объектам относятся строки символов, корневые деревья, многоугольники, а также последовательности целых чисел. Изучение динамического программирования лучше всего начинать с исследования готовых примеров. Для демонстрации практической пользы от динамического программирования в этой главе приводится несколько историй из жизни, в которых оно сыграло решающую роль в решении поставленной задачи.

8.1. Кэширование и вычисления

По сути, динамическое программирование является компромиссом, при котором повышенный расход памяти компенсируется экономией времени. Многократное вычисление некого значения безвредно само по себе, пока затраченное на это время не оказывает отрицательного влияния на производительность. В таком случае для повышения производительности будет разумнее не повторять вычисления, а сохранять результаты первоначальных вычислений и потом обращаться к ним в случае надобности.

Экономия времени исполнения за счет повышенного расхода памяти в динамическом программировании лучше всего проявляется при рассмотрении рекуррентных соотношений, таких как числа Фибоначчи. В последующих разделах мы рассмотрим три программы для вычисления этих последовательностей.

8.1.1. Генерирование чисел Фибоначчи методом рекурсии

Числа Фибоначчи были впервые исследованы в XIII веке итальянским математиком Фибоначчи для моделирования размножения кроликов. Фибоначчи предположил, что количество пар кроликов, рождающихся в данный год, равно сумме пар кроликов, рожденных в два предыдущие годы, начиная с одной пары кроликов в первом году. Для подсчета количества кроликов, рожденных в n-м году, он определил следующее рекуррентное соотношение:

$$F_n = F_{n-1} + F_{n-2}$$

для которого $F_0 = 0$ и $F_1 = 1$. Таким образом, $F_2 = 1$, $F_3 = 2$. Ряд продолжается в виде последовательности {3, 5, 8, 13, 21, 34, 55, 89, 144, ...}. Как потом выяснилось, формула Фибоначчи не очень хорошо подходит для описания размножения кроликов, но зато оказалось, что она обладает множеством интересных свойств. Так как числа Фибоначчи определяются рекурсивной формулой, то для вычисления n-го числа Фибоначчи легко создать рекурсивную программу. Пример такого рекурсивного алгоритма на языке C приводится в листинге 8.1.

Листинг 8.1. Рекурсивная функция для вычисления *n*-го числа Фибоначчи

```
long fib_r(int n)
{
   if (n == 0) return(0);
   if (n == 1) return(1);
   return(fib_r(n-1) + fib_r(n-2));
}
```

Ход выполнения этого рекурсивного алгоритма можно проиллюстрировать его рекурсивным деревом, как показано на рис. 8.1.

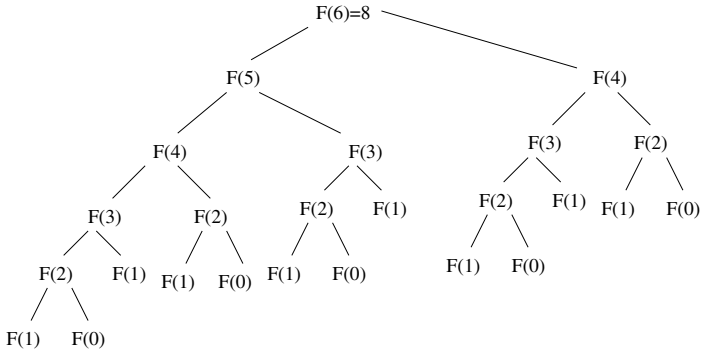


Рис. 8.1. Дерево для рекурсивного вычисления чисел Фибоначчи

Как и все рекурсивные алгоритмы, это дерево обрабатывается посредством обхода в глубину. Я настоятельно рекомендую проследить выполнение этого примера вручную, чтобы освежить ваши знания о рекурсии.

Обратите внимание, что число $F(4)$ вычисляется на обеих сторонах дерева, а число $F(2)$ вычисляется в этом небольшом примере целых пять раз. Полный вес этих избыточных вычислений становится ясным при исполнении программы. Для вычисления первых 45 чисел Фибоначчи моей программе потребовалось больше 7 минут. Скорее всего, используя правильный алгоритм, эти числа можно было бы быстрее вычислить вручную.

Сколько времени этот алгоритм будет вычислять число $F(n)$? Так как $F_{n+1} / F_n \approx \varphi = (1+\sqrt{5})/2 = 1{,}61803$, это означает, что $F_n > 1{,}6^n$. Так как листьями нашего дерева являются только 0 и 1, то такая большая сумма означает, что у нас должно быть, по крайней мере, $1{,}6^n$ листов или вызовов процедуры! Иными словами, эта небольшая простенькая программа имеет экспоненциальную временную сложность.

8.1.2. Генерирование чисел Фибоначчи посредством кэширования

Но в действительности мы можем решить эту задачу намного эффективнее. Для этого мы явно сохраняем (или *кэшируем*) результаты вычисления каждого числа Фибоначчи. $F(k)$ в таблице. Теперь, прежде чем вычислять значение, мы сначала проверяем его наличие в таблице, таким образом избегая повторных вычислений. Соответствующий код приводится в листинге 8.2.

Листинг 8.2. Вычисление чисел Фибоначчи с использованием кэширования

```
#define MAXN 45    /* Наибольшее представляющее интерес число n*/
#define UNKNOWN -1 /* Пустая ячейка */
long f[MAXN+1];    /* Массив для хранения вычисленных значений fib */
long fib_c(int n)
```

```
{
   if (f[n] == UNKNOWN)
      f[n] = fib_c(n-1) + fib_c(n-2);
   return(f[n]);
}
long fib_c_driver(int n)
{
   int i;                    /* Счетчик */
   f[0] = 0;
   f[1] = 1;
   for (i=2; i<=n; i++) f[i] = UNKNOWN;
   return(fib_c(n));
}
```

Чтобы вычислить $F(n)$, мы вызываем процедуру `fib_c_driver(n)`. Первым делом эта процедура инициализирует кэш двумя известными значениями ($F(0)$ и $F(1)$) и флагом `UNKNOWN` для остальных, неизвестных, значений. Потом вызывается рекурсивный алгоритм вычисления, модифицированный для предварительной проверки на наличие вычисленного значения данного числа Фибоначчи.

Версия с кэшированием очень быстро доходит до максимального целого числа, которое можно представить типом `long integer`. Причины этого становятся понятными после изучения дерева рекурсии, показанного на рис. 8.2.

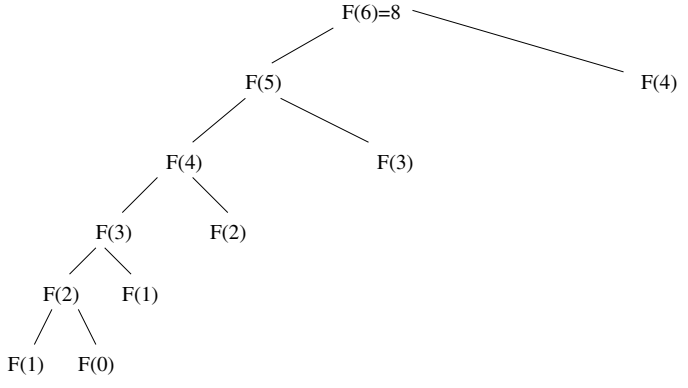


Рис. 8.2. Дерево вычисления чисел Фибоначчи методом кэширования

В данном случае отсутствует сколь-либо значительное ветвление, т. к. вычисления выполняются только в левой ветви. Вызовы в правой ветви находят нужные значения в кэше и немедленно возвращают управление.

Какова временная сложность этого алгоритма? Дерево рекурсии является более информативным, чем код. Значение $F(n)$ вычисляется за линейное время (т. е. за время $O(n)$), т. к. рекурсивная функция `fib_c(k)` вызывается ровно два раза для каждого значения k, где $0 \le k \le n$.

Данный общий метод явного кэширования результатов вызовов рекурсивной функции для того, чтобы избежать повторных вычислений, позволяет максимально использовать преимущества динамического программирования, что делает его заслуживающим

более внимательного рассмотрения. В принципе, кэширование можно применять с любым рекурсивным алгоритмом. Но сохранение частичных результатов не принесет никакой пользы при работе таких рекурсивных алгоритмов, как быстрая сортировка, перебор с возвратами и обход в глубину, т. к. все рекурсивные вызовы в этих алгоритмах имеют разные значения параметров. Нет смысла сохранять то, что больше не будет использовано.

Кэширование результатов вычислений имеет смысл только в случае достаточно небольшого пространства значений параметров, когда мы можем себе позволить расходы на хранение данных. Так как аргументом рекурсивной функции `fib_c(k)` является целое число из диапазона от 0 до n, то кэшировать нужно только $O(n)$ значений. Нам выгодно пойти на линейные затраты памяти вместо экспоненциальных затрат времени. Но, как мы увидим далее, полностью избавившись от рекурсии, можно получить еще лучшую производительность.

Подведение итогов

Явное кэширование результатов рекурсивных процедур позволяет максимально использовать преимущества динамического программ ирования, главным достоинством которого является такое же время выполнения, что и у более элегантных решений. Если вы предпочитаете бесхитростное написание большого объема кода поиску красивого решения, вы можете не читать последующий материал.

8.1.3. Генерирование чисел Фибоначчи посредством динамического программирования

Число Фибоначчи F_n можно вычислить за линейное время с большей эффективностью, если явно задать последовательность рекуррентных вычислений, как показано в листинге 8.3.

Листинг 8.3. Вычисление числа Фибоначчи без рекурсии

```
long fib_dp(int n)
{
   int i;          /* Счетчик */
   long f[MAXN+1]; /* Массив для хранения вычисленных значений fib */
   f[0] = 0;
   f[1] = 1;
   for (i=2; i<=n; i++) f[i] = f[i-1]+f[i-2];
   return(f[n]);
}
```

Как можно видеть, в данном варианте процедуры рекурсия совсем не используется. Мы начинаем вычисление последовательности Фибоначчи с ее наименьшего значения до заданного числа и сохраняем все промежуточные результаты. Таким образом, когда нам нужно вычислить число F_i, мы уже имеем требуемые для этого числа F_{i-1} и F_{i-2}. Линейная временная сложность этого алгоритма должна быть очевидной. Вычисление каждого из n значений выполняется как простое суммирование двух целых чисел, сложность которого, как по времени, так и по памяти, равна $O(n)$.

Но более внимательное исследование процесса решения задачи показывает, что совсем не обязательно хранить все промежуточные результаты в течение вычисления всей последовательности. Так как вычисляемое значение зависит от двух аргументов, то только их и нужно сохранять. Соответствующая реализация показана в листинге 8.4.

Листинг 8.4. Окончательная версия процедуры вычисления чисел Фибоначчи

```
long fib_ultimate(int n)
{
   int i;                    /* Счетчик */
   long back2=0, back1=1;    /* Последние два значения f[n] */
   long next;                /* Промежуточная сумма */
   if (n == 0) return (0);
   for (i=2; i<n; i++) {
      next = back1+back2;
      back2 = back1;
      back1 = next;
   }
   return(back1+back2);
}
```

Данная реализация понижает сложность по памяти до постоянной, при этом никак не ухудшая сложность по времени.

8.1.4. Биномиальные коэффициенты

В качестве другого примера устранения рекурсии указанием порядка вычислений рассмотрим вычисление *биномиальных коэффициентов*. Биномиальные коэффициенты являются наиболее важным классом натуральных чисел. В комбинаторике биномиальный коэффициент $\binom{n}{k}$ представляет количество возможных подмножеств из k элементов множества из n элементов.

Каким образом находятся биномиальные коэффициенты? Так как $\binom{n}{k} = n!/((n-k)!k!)$, то их значения можно получить, вычисляя факториалы. Но этот метод имеет серьезный недостаток. Промежуточные вычисления могут вызвать арифметическое переполнение, даже если конечный результат не превышает максимальное допустимое целое число в компьютере.

Более надежным способом вычисления биномиальных коэффициентов является использование неявного рекуррентного соотношения, используемого для построения треугольника Паскаля:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
```

Каждый элемент строки является суммой двух элементов слева и справа от него в предшествующей строке. Это подразумевает следующее рекуррентное соотношение: $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$.

Почему эта формула дает правильный результат? Рассмотрим, входит ли n-й элемент в одно из подмножеств $\binom{n}{k}$ из k элементов. Если входит, то мы можем завершить создание подмножества, выбрав другие $k-1$ элементов из оставшихся $n-1$ элементов множества. Если же не входит, то нам нужно выбрать все k элементов из оставшихся $n-1$ элементов множества. Эти два случая не пересекаются и включают в себя все возможности, поэтому в сумме учитываются все k подмножеств.

Для любого рекуррентного соотношения требуется база. Какие значения биномиальных коэффициентов нам известны без вычислений? Левая часть формулы легко приводится к $\binom{n-k}{0}$. Сколько можно создать подмножеств, содержащих 0 элементов, из некого множества? Ровно одно — пустое. Если это не кажется вам убедительным, с равным успехом в качестве базы можно принять $\binom{m}{1} = m$. Правая часть формулы легко приводится к $\binom{k}{k}$. Сколько можно создать подмножеств, содержащих k элементов, из множества, содержащего k элементов? Ровно одно — первоначальное множество. Эти базовые экземпляры и рекуррентное соотношение определяют все интересующие нас биномиальные коэффициенты.

Самым лучшим способом вычисления такого рекуррентного соотношения будет создание таблицы всех возможных значений, имеющей требуемый размер, как показано на рис. 8.3.

m / n	0	1	2	3	4	5
0	A					
1	B	G				
2	C	1	H			
3	D	2	3	I		
4	E	4	5	6	J	
5	F	7	8	9	10	K

а)

m / n	0	1	2	3	4	5
0	1					
1	1	1				
2	1	1	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	**5**	1

б)

Рис. 8.3. Порядок вычисления биномиальных коэффициентов *(а)*, содержимое матрицы после вычислений *(б)*

Инициализированные ячейки таблицы помечены буквами от A до K, обозначающими порядок, в котором им были присвоены значения. Каждой неинициализированной ячейке присваивается значение, равное сумме значений двух ячеек из предыдущего ряда: той, что непосредственно над ней, и той, что сверху и слева. Цифры от 1 до 10, маркирующие треугольник ячеек, обозначают порядок вычисления подмножества $\binom{5}{4} = 5$ с помощью кода, представленного в листинге 8.5.

Листинг 8.5. Вычисление биномиального коэффициента

```
long binomial_coefficient(n, m)
int n, m;                /* Вычислить n, выбрать m */
{
   int  i, j;            /* Счетчики  */
   long bc[MAXN][MAXN]; /* Таблица биномиальных коэффициентов */
   for (i=0; i<=n; i++) bc[i][0] = 1;
   for (j=0; j<=n; j++) bc[j][j] = 1;
   for (i=1; i<=n; i++)
      for (j=1; j<i; j++)
         c[i][j] = bc[i-1][j-1] + bc[i-1][j];
   return( bc[n][m] );
}
```

Внимательно изучите эту функцию, чтобы понять, как она работает. Далее в этой главе мы будем уделять больше внимания вопросам формулирования и анализа соответствующей рекуррентности, чем технике работы с таблицами.

8.2. Поиск приблизительно совпадающих строк

Поиск совпадающих комбинаций символов в текстовых строках является задачей, важность которой не подлежит сомнению. В *разделе 3.7.2* были представлены алгоритмы для поиска точных совпадений комбинаций символов, т. е. выяснения, где именно в текстовой строке T находится искомая подстрока P. Но, к сожалению, жизнь не всегда так проста. Например, слова, как в тексте, так и в искомой строке, могут быть написаны с ошибками, вследствие чего точного совпадения не получится. Эволюционные изменения в геномных последовательностях или языковых конструкциях приводят к тому, что мы вводим в строке поиска "Не убей", когда нужно найти "Не убий".

Каким образом можно нейтрализовать орфографические ошибки, чтобы можно было найти подстроку, наиболее близкую к искомой? Чтобы можно было выполнять поиск неточно совпадающих строк, нам нужно сначала определить функцию стоимости, с помощью которой мы будем выяснять, насколько далеко две строки находятся друг от друга, т. е. измерять расстояние между парами строк. Для этого нам нужно учитывать количество операций, которые придется выполнить, чтобы преобразовать одну строку в другую. Возможны такие действия:

- *замена* — заменяет один символ строки P другим символом из текста T. Например, "shot" превращается в "spot";
- *вставка* — вставляет один символ в строку P так, чтобы она совпадала с подстрокой в тексте T. Например, "ago" превращается в "agog";
- *удаление* — удаляет один символ из строки P так, чтобы она совпадала с подстрокой в тексте T. Например, "hour" превращается в "our".

Для правильной постановки вопроса сходства строк нам нужно установить стоимость каждой из этих операций преобразования строк. Присваивая каждой операции стоимость равную 1, мы определяем *расстояние редактирования* между двумя строками.

Задача нечеткого сравнения строк возникает во многих приложениях и рассматривается в *разделе 18.4*.

Нечеткое сравнение строк может казаться трудной задачей, т. к. нам нужно решить, где именно в строке-образце и тексте нужно вставить или удалить символы и сколько символов нужно вставить или удалить. Попробуем подойти к рассмотрению задачи с другого конца. Какая информация нам понадобится для принятия окончательного решения? Что может случиться при сравнении с последним символом каждой строки?

8.2.1. Применение рекурсии для вычисления расстояния редактирования

При создании рекурсивного алгоритма можно использовать то обстоятельство, что либо последний символ в строке совпадет с искомым, либо нам придется его заменить, удалить или добавить. В результате отсечения символов, участвующих в этой последней операции редактирования, останется пара более коротких строк. Пусть i и j будут последними символами префиксов строк P и T соответственно. В результате последней операции образуются три пары более коротких строк, соответствующих совпадающим строкам либо строкам, полученным после замены, добавления или удаления. *Если* бы нам была известна стоимость редактирования этих более коротких строк, мы могли бы решить, какая опция дает наилучшее решение, и соответственно выбрать эту опцию. Оказывается, с помощью рекурсии мы *можем* узнать эту стоимость.

Пусть $D[i, j]$ — минимальное количество различий между образцами P_1, P_2, ..., P_i и фрагментом текста T, оканчивающимся на j. Иными словами, $D[i, j]$ представляет самый дешевый из трех возможных способов расширения более коротких строк:

- если ($P_i = T_j$), тогда $D[i - 1, j - 1]$, в противном случае $D[i - 1, j - 1] + 1$. Это означает, что в зависимости от того, одинаковы ли последние символы строк, мы получаем совпадение i-го и j-го символов или заменяем их;
- $D[i - 1, j] + 1$. Это означает, что в строке образца имеется дополнительный символ, который нужно учесть, поэтому указатель позиции в тексте не продвигается и выполняется вставка символа;
- $D[i, j - 1] + 1$. Это означает, что в тексте имеется лишний символ, который нужно удалить, поэтому указатель позиции в строке-образце не продвигается и выполняется удаление символа.

В листинге 8.6 представлена программа вычисления стоимости редактирования.

Листинг 8.6. Вычисление стоимости редактирования методом рекурсии

```
#define MATCH 0      /* Символ перечислимого типа для совпадения */
#define INSERT 1     /* Символ перечислимого типа для вставки */
#define DELETE 2     /* Символ перечислимого типа для удаления */
int string_compare(char *s, char *t, int i, int j)
{
   int k;              /* Счетчик*/
   int opt [3];        /* Стоимость трех опций */
   int lowest_cost;    /* Самая низкая стоимость */
```

```
   if (i == 0) return(j * indel(' '));
   if (j == 0) return(i * indel(' '));
   opt[MATCH] = string_compare(s, t, i-1, j-1) + match(s[i],t[j]);
   opt[INSERT] = string_compare(s,t,i,j-1) + indel(t[j]);
   opt[DELETE] = string_compare(s,t,i-1,j) + indel(s[i]);
   lowest_cost = opt[MATCH];
   for (k=INSERT; k<=DELETE; k++)
      if (opt[k] < lowest_cost) lowest_cost = opt[k];
   return( lowest_cost );
}
```

Легко убедиться, что эта программа абсолютно корректна. Однако работает она недопустимо медленно. На моем компьютере сравнение двух строк длиной в 11 символов занимает несколько секунд, а более-менее длинные строки обрабатываются целую вечность.

Почему этот алгоритм работает так медленно? Потому что он вычисляет одни и те же значения по нескольку раз, в результате чего имеет экспоненциальную временную сложность. В каждой позиции строки происходит тройное ветвление рекурсии, вследствие чего количество вызовов растет со скоростью 3^n, а на самом деле даже быстрее, т. к. при большинстве вызовов сокращается только один из двух индексов, а не оба.

8.2.2. Применение динамического программирования для вычисления расстояния редактирования

Так как же можно сделать этот алгоритм пригодным для использования на практике? В решении этого вопроса важным является то обстоятельство, что при большинстве рекурсивных вызовов выполняются вычисления, которые уже были выполнены. Откуда нам это известно? Возможно только $|P|\cdot|T|$ однозначных рекурсивных вызовов, т. к. именно столько имеется разных пар (i, j), передаваемых в качестве параметров. Сохранив вычисленные значения для каждой из этих (i, j) пар в таблице, мы можем при необходимости извлечь их из этой таблицы, вместо того, чтобы вычислять их снова.

Таблица для хранения вычисленных значений представляет собой двумерную матрицу m, каждая из $|P|\cdot|T|$ ячеек которой содержит значение стоимости оптимального решения подзадачи, а также указатель на родительский элемент, объясняющий, как мы попали в эту ячейку. Объявление структуры таблицы показано в листинге 8.7.

Листинг 8.7. Структура таблицы вычисление стоимости редактирования

```
typedef struct {
   int cost;           /* Стоимость попадания в данную ячейку */
   int parent;         /* Родительская ячейка */
} cell;
cell m[MAXLEN+1] [MAXLEN+1]; /* Таблица динамич. программирования */
```

Между динамической и рекурсивной версиями реализации алгоритма поиска неточно совпадающих строк имеются три различия. Во-первых, промежуточные значения получаются из таблицы, а не в результате рекурсивных вызовов. Во-вторых, обновляется содержимое поля родительского указателя каждой ячейки, что в дальнейшем позволит

восстановить последовательность редактирования. В-третьих, в реализации используется более общая функция `goal_cell()`, а не просто возвращается значение `m[|P|][|T|].cost`. Как следствие, мы можем применять данную процедуру для решения более обширного класса задач. Реализация алгоритма представлена в листинге 8.8.

Листинг 8.8. Вычисление стоимости редактирования

```
int string_compare(char *s, char *t)
{
   int i, j, k;          /* Счетчики */
   int opt[3];           /* Стоимость трех вариантов */
   for (i=0; i<MAXLEN; i++) {
      row_init(i);
      column_init(i);
   }
   for (i=1; i<strlen(s); i++) {
      for (j=1; j<strlen(t); j++)
         opt[MATCH] = m[i-1][j-1].cost + match(s[i],t[j]);
         opt[INSERT] = m[i][j-1].cost + indel(t[j]);
         opt[DELETE] = m[i-1][j].cost +  indel(s[i]);
         m[i][j].cost = opt[MATCH];
         m[i][j].parent = MATCH;
         for (k=INSERT; k<=DELETE; k++) {
            if (opt[k] < m[i][j].cost) {
               m[i][j].cost = opt[k];
               m[i][j].parent = k;
            }
         }
      }
   goal_cell(s, t, &i, &j);
   return( m[i][j].cost );
}
```

Здесь строки и индексы используются несколько необычным образом. В частности, полагается, что в начало каждой строки был добавлен пробел, вследствие чего первый значащий символ строки `s` находится в позиции `s[1]`. Это позволяет нам синхронизировать индексы матрицы `m` с индексами строк. Вспомните, что нам нужно выделить нулевую строку и столбец матрицы `m` для хранения граничных значений, совпадающих с пустым префиксом. В качестве альтернативы мы бы могли оставить входные строки без изменений и просто откорректировать должным образом индексы.

Для определения значения ячейки (i, j) требуются три готовых значения из ячеек $(i-1, j-1)$, $(i, j-1)$ и $(i-1, j)$. Подойдет любой порядок вычислений, удовлетворяющий этому условию, в том числе построчный, используемый в этой программе[1].

[1] Допустим, мы создадим граф с вершиной для каждой ячейки матрицы и ориентированным ребром (x, y), означающим, что для вычисления значения ячейки y нужно значение ячейки x. Любая топологическая сортировка на получившемся бесконтурном орграфе (кстати, почему это будет бесконтурный орграф?) определяет приемлемый порядок вычислений.

В качестве примера приводятся матрицы стоимости преобразования строки p = "thou shalt not" в строку t = "you should not" за пять шагов (табл. 8.1).

Таблица 8.1. Пример матрицы динамического программирования

	T		y	o	u	-	s	h	o	u	l	d	-	n	o	t
P	поз.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
:		**0**	1	2	3	4	5	6	7	8	9	10	11	12	13	14
t:	1	**1**	1	2	3	4	5	6	7	8	9	10	11	12	13	13
h:	2	2	**2**	2	3	4	5	5	6	7	8	9	10	11	12	13
o:	3	3	3	**2**	3	4	5	6	5	6	7	8	9	10	11	12
u:	4	4	4	3	**2**	3	4	5	6	5	6	7	8	9	10	11
-:	5	5	5	4	3	**2**	3	4	5	6	6	7	7	8	9	10
s:	6	6	6	5	4	3	**2**	3	4	5	6	7	8	8	9	10
h:	7	7	7	6	5	4	3	**2**	**3**	4	5	6	7	8	9	10
a:	8	8	8	7	6	5	4	3	3	**4**	5	6	7	8	9	10
l:	9	9	9	8	7	6	5	4	4	4	**4**	5	6	7	8	9
t:	10	10	10	9	8	7	6	5	5	5	5	**5**	6	7	8	8
-:	11	11	11	10	9	8	7	6	6	6	6	6	**5**	6	7	8
n:	12	12	12	11	10	9	8	7	7	7	7	7	6	**5**	6	7
o:	13	13	13	12	11	10	9	8	7	8	8	8	7	6	**5**	6
t:	14	14	14	13	12	11	10	9	8	8	9	9	8	7	6	**5**

8.2.3. Восстановление пути

Функция сравнения строк возвращает стоимость оптимального выравнивания, но не выполняет собственно выравнивание. Нам полезно знать, что для преобразования строки "thou shalt not" в строку "you should not" требуется только пять операций редактирования, но было бы еще полезнее знать последовательность этих операций.

Возможные решения определенной задачи динамического программирования описываются путями в матрице динамического программирования, начинающимися с первоначальной конфигурации (пары пустых строк (0, 0)) и заканчивающимися конечным требуемым состоянием (парой заполненных строк ($|P|$,$|T|$)). Ключом к созданию решения задачи является реконструкция решений, принимаемых на каждом шаге оптимального пути, ведущего к целевому состоянию. Эти решения записаны в поле указателя на родителя каждой ячейки массива.

Нужные решения можно воспроизвести, выполнив проход по решениям в обратном направлении от целевого состояния, следуя указателям на родительские ячейки массива, пока не придем к начальной ячейке пути решения задачи. Поле указателя на родителя ячейки $m[i, j]$ содержит информацию о типе операции, выполненной в этой ячейке — MATCH, INSERT или DELETE. Обратная трассировка пути решения в табл. 8.2 преобразования строки "thou-shalt-not" в строку "you-should-not" выдает нам последовательность операций редактирования DSMMMMMISMSMMMM. Это означает, что мы удаляем первую букву "t", заменяем букву "h" буквой "y", оставляем без изменений

следующие пять букв, после чего вставляем букву "o", заменяем букву "a" буквой "u" и, наконец, заменяем букву "t" буквой "d".

Таблица 8.2. Матрица указателей на родителей

	T		y	o	u	-	s	h	o	u	l	d	-	n	o	t
P	поз.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	0	**-1**	1	1	1	1	1	1	1	1	1	1	1	1	1	1
t:	1	**2**	0	0	0	0	0	0	0	0	0	0	0	0	0	0
h:	2	2	**0**	0	0	0	0	0	1	1	1	1	1	1	1	1
o:	3	2	0	**0**	0	0	0	0	0	1	1	1	1	1	0	1
u:	4	2	0	2	**0**	1	1	1	1	0	1	1	1	1	1	1
-:	5	2	0	2	2	**0**	1	1	1	1	0	0	0	1	1	1
s:	6	2	0	2	2	2	**0**	1	1	1	1	0	0	0	0	0
h:	7	2	0	2	2	2	2	**0**	**1**	1	1	1	1	1	0	0
a:	8	2	0	2	2	2	2	2	0	**0**	0	0	0	0	0	0
l:	9	2	0	2	2	2	2	2	0	0	**0**	1	1	1	1	1
t:	10	2	0	2	2	2	2	2	0	0	0	**0**	0	0	0	0
-:	11	2	0	2	2	0	2	2	0	0	0	0	**0**	1	1	1
n:	12	2	0	2	2	2	2	2	0	0	0	0	2	**0**	1	1
o:	13	2	0	0	2	2	2	2	0	0	0	0	2	2	**0**	1
t:	14	2	0	2	2	2	2	2	2	0	0	0	2	2	2	**0**

Трассировка указателей на родительские ячейки восстанавливает решение в обратном порядке. Но с помощью рекурсии мы можем восстановить прямой порядок решения. Соответствующий код представлен в листинге 8.9.

Листинг 8.9. Восстановление решения в прямом порядке

```
reconstruct_path(char *s, char *t, int i, int  j)
{
   if (m[i][j].parent == -1) return;
   if (m[i][j].parent == MATCH) {
      reconstruct_path(s, t, i-1, j-1);
      match_out(s, t, i, j);
      return;
   }
   if (m[i][j].parent == INSERT) {
      reconstruct_path(s, t, i, j-1);
      insert_out(t, j);
      return;
   }
   if (m[i][j].parent == DELETE) {
      reconstruct_path(s, t, i-1, j);
      delete_out(s, i);
      return;
   }
}
```

Для многих задач, включая задачу вычисления расстояния редактирования, путь решения можно восстановить из матрицы стоимости, не прибегая к явному сохранению указателей на предыдущий элемент. Для этого нам нужно выполнить трассировку назад, начиная со значений стоимости трех возможных родительских ячеек и соответствующих символов строки, чтобы восстановить операцию, в результате выполнения которой мы оказались в текущей ячейке при данной стоимости.

8.2.4. Разновидности расстояния редактирования

Процедуры сравнения строк (`string_compare`) и восстановления пути (`reconstruct_path`) вызывают несколько функций, которые еще не были определены. Эти функции можно разбить на четыре категории.

- *Инициализация таблицы.* Функции `row_init()` и `column_init()` инициализируют нулевые строку и столбец таблицы соответственно. В задаче вычисления расстояния редактирования ячейки (i, 0) и (0, i) соответствуют сравнению строк длиной i с пустой строкой. Для этого требуется ровно i вставок/удалений, поэтому код этих функций очевиден (листинг 8.10).

Листинг 8.10. Процедура инициализации строк и столбцов таблицы

```
row_init(int i)
{
   m[0][i].cost = i;
   if (i>0)
      m[0][i].parent = INSERT;
   else
      m[0][i].parent = -1;
}
column_init(int i)
{
   m[i][0].cost = i;
   if (i>0)
      m[i][0].parent = DELETE;
   else
      m[i][0].parent = -1;
}
```

- *Стоимость операций.* Функции `match(c,d)` и `indel(c)` возвращают стоимость преобразования символа c в символ d и вставки/удаления символа c соответственно. Для стандартной задачи вычисления расстояния редактирования функция `match()` возвращает 0, если символы одинаковые, и 1 в противном случае. Функция `indel()` всегда возвращает 1, независимо от аргумента. Но можно также использовать функции стоимости, специфичные для конкретных приложений. Такие функции могут быть менее взыскательными к заменяемым символам, расположенным друг возле друга на стандартной раскладке клавиатуры, или символам, которые выглядят или звучат похоже.

  Общая реализация функций стоимости показана в листинге 8.11.

Листинг 8.11. Функции стоимости

```
int match(char c, char d)
{
   if (c == d) return(0);
   else return(1);
}
int indel(char c)
{
   return(1);
}
```

- *Определение целевой ячейки.* Функция `goal_cell()` возвращает индексы ячейки, обозначающей конечную точку решения. В случае задачи вычисления расстояния редактирования это значение определяется длиной двух входных строк. Но другие приложения, которые мы вскоре рассмотрим, не имеют фиксированного расположения решений.

  Общая реализация функции определения местонахождения целевой ячейки приводится в листинге 8.12.

Листинг 8.12. Функция определения местонахождения целевой ячейки

```
goal_cell(char *s, char *t, int *i, int *j)
{
   *i = strlen(s) - 1;
   *j = strlen(t) - 1;
}
```

- *Операция обратной трассировки.* Функции `match out()`, `insert_out()` и `delete_out()` выполняют соответствующие действия для каждой операции редактирования при трассировке решения. В случае задачи вычисления расстояния редактирования это означает вывод названия операции или обрабатываемого символа, в зависимости от требований приложения.

  Общая реализация функций трассировки решения приводится в листинге 8.13.

Листинг 8.13. Функции трасировки решения

```
insert_out(char *t, int j)
{
   printf("I");
}
delete_out(char *s, int i)
{
   printf("D");
}
match_out(char *s, char *t, int i, int j)
{
   if (s[i]==t[j]) printf("M");
   else printf("S");
}
```

Для задачи вычисления расстояния редактирования эти функции довольно простые. Но следует признать, что правильное выполнение операций получения граничных состояний и манипулирования индексами является трудной задачей. Хотя при должном понимании применяемого метода алгоритмы динамического программирования легко разрабатывать, для правильной реализации деталей требуется внимательно продумать и всесторонне протестировать предлагаемое решение.

На первый взгляд кажется, что для такого простого алгоритма потребовалось слишком много вспомогательной работы. Но теперь, только слегка модифицировав эти общие функции, мы можем решить несколько важных задач, как особые случаи задачи вычисления расстояния редактирования.

- *Поиск подстроки в тексте.* Допустим, что мы хотим найти в тексте T подстроку, приблизительно совпадающую с подстрокой P. Например, будем искать строку "Skiena" и возможные ее варианты (Skienna, Skena, Skeena, Skina и т. п.). Поиск с помощью нашей первоначальной функции вычисления расстояния редактирования будет малочувствительным, т. к. большая часть стоимости любого редактирования будет определяться удалением фрагментов текста, не совпадающих со строкой "Skiena". В данном случае оптимальным решением будет поиск всех разбросанных в тексте совпадений с "...S...k...i...e...n...a" и удаление всего остального.

  Нам требуется такой способ вычисления расстояния редактирования, в котором стоимость начала совпадения не зависит от местонахождения в тексте, чтобы учитывались и совпадения в середине текста. Теперь целевое состояние находится не обязательно в конце строк, а в таком месте текста, где совпадение со всем образцом имеет минимальную стоимость. Модифицировав эти две функции, мы получим требуемое правильное решение, как показано в листинге 8.14.

Листинг 8.14. Модифицированные функции для поиска неточно совпадающих строк

```
row_init(int i)
{
   m[0][i].cost = 0;      /* ИЗМЕНЕНИЕ */
   m[0][i].parent = -1;   /* ИЗМЕНЕНИЕ */
}
goal_cell(char *s, char *t, int *i, int *j)
{
   int k;      /* Счетчик */
   *i = strlen(s) - 1;
   *j = 0;
   for (k=1; k<strlen(t); k++)
      if (m[*i][k].cost < m[*i][*j].cost) *j   k;
}
```

- *Самая длинная общая подпоследовательность.* Допустим, что нам нужно найти самую длинную последовательность разбросанных символов, общую для обеих строк. (Данная задача рассматривается в *разделе 18.8*).

  Общая подпоследовательность представляет собой последовательность всех разбросанных совпадающих символов в обеих строках. Например, самой длинной об-

щей подпоследовательностью строк "democrat" и "republican" является "eca". Чтобы максимизировать количество таких совпадений, требуется предотвратить замену несовпадающих символов. Когда замена символов запрещена, избавиться от несовпадающих символов можно только с помощью операций вставки и удаления. Для получения самого дешевого выравнивания будет выполнено наименьшее количество таких операций, поэтому в ней должна сохраниться самая длинная общая подпоследовательность. Чтобы получить требуемое выравнивание, мы модифицируем функцию стоимости совпадений, чтобы повысить стоимость замены символов, как показано в листинге 8.15.

Листинг 8.15. Модифицированная функция стоимости совпадений

```
int match(char c, char d)
{
   if (c == d) return(0);
   else return(MAXLEN);
}
```

В действительности, чтобы сделать замену полностью непривлекательной операцией редактирования, будет достаточным сделать ее стоимость выше, чем совместная стоимость вставки и удаления.

- *Максимальная монотонная подпоследовательность.* Числовая последовательность является *монотонно возрастающей*, если каждый следующий элемент этой последовательности превышает предыдущий. Задача *максимальной монотонной подпоследовательности* состоит в удалении наименьшего количества элементов из входной строки S с целью получения монотонно возрастающей подпоследовательности. Например, для последовательности 243517698 максимальной возрастающей подпоследовательностью является 23568.

  По сути, это просто задача поиска самой длинной общей подпоследовательности, где второй строкой являются элементы строки S, отсортированные в возрастающем порядке. Любая из этих двух общих последовательностей должна, во-первых, содержать символы в соответствующем порядке в строке S и, во-вторых, содержать только символы с возрастающими номерами позиций в последовательности упорядочивания, так что более длинная из этих последовательностей является решением. Конечно же, этот подход можно модифицировать для поиска самой длинной убывающей последовательности, просто изменив порядок сортировки на обратный.

Как можно видеть, базовую процедуру вычисления расстояния редактирования можно с легкостью приспособить для решения многих интересных задач. Секрет заключается в определении, что конкретная задача является всего лишь частным случаем общей задачи нечеткого сравнения строк.

Внимательный читатель может заметить, что для вычисления затрат на выравнивание не требуется содержание всех $O(mn)$ ячеек. Если мы будем вычислять рекуррентное соотношение, заполняя столбцы матрицы слева направо, то нам никогда не потребуется больше двух столбцов для хранения всей требуемой для этого вычисления информации. Таким образом, для вычисления рекуррентного соотношения достаточно

объема памяти $O(m)$, при этом сложность по времени остается прежней. Но, к сожалению, не имея полной матрицы, мы не можем восстановить выравнивание.

Экономия памяти является важным моментом динамического программирования. Так как объем памяти на любом компьютере ограничен, то сложность по памяти, равная $O(nm)$, является более узким местом, чем такая же сложность по времени. К счастью, это проблема решается с помощью алгоритма "разделяй и властвуй", который выполняет выравнивание за время $O(nm)$, требуя при этом память объемом $O(m)$. Этот алгоритм рассматривается в *разделе 18.4*.

8.3. Самая длинная возрастающая последовательность

Решение задачи посредством динамического программирования состоит из трех шагов:

1. Сформулировать решение в виде рекуррентного соотношения или рекурсивного алгоритма.
2. Показать, что количество разных значений параметра, принимаемых рекуррентностью, ограничено полиномиальной функцией (будем надеяться, небольшой степени).
3. Указать порядок вычисления рекуррентного соотношения, с тем, чтобы частичные результаты были всегда доступными, когда они требуются.

Чтобы разобраться в этих деталях, рассмотрим разработку алгоритма поиска самой длинной монотонно возрастающей подпоследовательности в последовательности из n чисел. По правде говоря, эта задача была описана, как частный случай задачи вычисления расстояния редактирования в предшествующем разделе, где она называлась задачей поиска максимальной монотонной подпоследовательности. Тем не менее, будет полезно рассмотреть разработку ее решения с самого начала. В действительности, алгоритмы динамического программирования часто легче разработать сначала, чем искать существующее решение.

Возрастающая последовательность отличается от *серии*, в которой элементы физически находятся рядом друг с другом. Выбранные элементы обеих структур должны быть отсортированы в возрастающем порядке слева направо. Рассмотрим, например, последовательность $S = \{2, 4, 3, 5, 1, 7, 6, 9, 8\}$.

Самая длинная возрастающая подпоследовательность последовательности S состоит из пяти символов: $\{2, 3, 5, 6, 8\}$. В действительности, подпоследовательностей этой длины в последовательности S имеется восемь. (Попробуйте найти их.) А самых длинных серий, длиной в два символа, имеется четыре: (2, 4), (3, 5), (1, 7) и (6, 9).

Поиск самой длинной возрастающей серии в числовой последовательности является простой задачей. По сути, разработка алгоритма с линейным временем исполнения не должна вызвать особых трудностей. Но задача поиска самой длинной возрастающей подпоследовательности значительно сложнее. Как мы можем определить, какие разбросанные элементы пропустить? Чтобы применить динамическое программирование, нам нужно создать рекуррентное соотношение, которое вычисляет длину самой длинной последовательности. Чтобы найти подходящее рекуррентное соотношение, задайте

себе вопрос, какая информация о первых $n-1$ элементах последовательности S помогла бы найти решение для всей последовательности?

- Длина самой длинной возрастающей подпоследовательности последовательности $s_1, s_2, ..., s_{n-1}$ кажется полезной информацией. Более того, это будет самая длинная возрастающая подпоследовательность в S, если только s_n не предоставит возрастающую последовательность такой же длины.

  К сожалению, длина этой последовательности не является достаточной информацией для получения полного решения. Допустим, что каким-то образом мы узнали, что самая длинная возрастающая подпоследовательность последовательности $s_1, s_2, ..., s_{n-1}$ содержит пять символов и что $s_n = 9$. Будет ли длина последней самой длинной возрастающей подпоследовательности последовательности S равняться 5 или 6?

- Нам необходимо знать количество символов в самой длинной последовательности, которую расширит s_n. Чтобы быть уверенным в том, что мы знаем это, нам в действительности нужно знать количество символов в самой длинной последовательности, которую может расширить *любое* возможное значение для s_n.

Это предоставляет нам базовую идею для создания рекуррентного соотношения. Определяем l_i, как количество символов самой длинной последовательности, заканчивающейся на s_i.

Самая длинная возрастающая подпоследовательность, содержащая n-е число, получается в результате добавления этого числа в конец самой длинной возрастающей последовательности слева от n и оканчивающейся числом, меньшим, чем s_n. Длина l_i вычисляется с помощью следующего рекуррентного соотношения:

$$l_i = \max_{0<j<i} l_j + 1, \text{ где } s_j < s_i,$$

$$l_0 = 0$$

Эти значения определяют количество символов в самой длинной возрастающей последовательности, заканчивающейся данным числом. Количество символов в самой длинной возрастающей подпоследовательности полной перестановки можно выразить формулой $\max_{1\le i\le n} l_i$, т. к. самая лучшая последовательность должна когда-нибудь закончиться. В табл. 8.3 приводятся данные для предыдущего примера.

Таблица 8.3. Данные для задачи поиска самой длинной возрастающей подпоследовательности

Последовательность s_i	2	4	3	5	1	7	6	9	8
Длина l_i	1	2	3	3	1	4	4	5	5
Предшественник p_i	–	1	1	2	–	4	4	6	6

Какую вспомогательную информацию нам следует сохранить, чтобы восстановить саму последовательность, а не только ее длину? Для каждого элемента s_i сохраняется его предшественник, а именно индекс p_i элемента, непосредственно предшествующего s_i в самой длинной возрастающей последовательности, заканчивающейся на s_i. Так как все эти указатели направлены влево, то самую длинную последовательность можно

с легкостью восстановить, начав с ее последнего значения и следуя указателям на другие ее члены.

Какова временная сложность этого алгоритма? Если каждое из n значений l_i вычисляется путем сравнения s_i с n значениями слева от него (где $n \geq i - 1$), то общее время этого анализа будет равно $O(n^2)$. На самом деле, умело используя словарные структуры данных, это рекуррентное соотношение можно вычислить за время $O(n \lg n)$. Поскольку простое рекуррентное соотношение легче поддается программированию, лучше начать с его реализации.

Подведение итогов

Когда у вас достаточно опыта динамического программирования, то такие алгоритмы будет легче создавать с нуля, чем искать готовое подходящее решение.

8.4. История из жизни. Эволюция омара

Придя утром на работу, я застал возле моего кабинета двух аспирантов. Это были два будущих кандидата наук, работающих в области высокопроизводительной компьютерной графики. Они занимались исследованием новых методов для создания красивых компьютерных изображений, но картина, которую они нарисовали мне в то утро, красивой не была.

— Видите ли, мы хотим создать программу для морфинга одного изображения в другое, — начали объяснять они.

— Что вы понимаете под морфингом? — спросил я.

— Для реализации спецэффектов в фильмах мы хотим создавать промежуточные сцены при преобразовании одного изображения в другое. Допустим, мы хотим превратить вас в актера Хамфри Богарта. Чтобы такое превращение выглядело правдоподобным, нам нужно создать множество промежуточных кадров, таких, что в первых вы выглядите сами собой, а в последних, как он.

— Если вы вправду можете превратить меня в Богарта, то вы действительно нашли что-то стоящее, — согласился я.

— Но у нас возникают проблемы, т. к. процесс выглядит не очень правдоподобно. — Они показали мне морфинг ужасного качества. — Проблема состоит в том, что нам нужно найти правильное соответствие между чертами двух изображений. Результат никуда не годится, когда мы получаем неправильное соответствие и пытаемся преобразовать, скажем, губу в ухо.

— Еще бы. Значит, вы хотите, чтобы я вам дал алгоритм, чтобы сопоставлять губы с губами?

— Даже проще, чем это. Мы преобразуем каждую строку первоначального изображения в такую же строку конечного изображения. Нам нужно найти оптимальное соответствие между темными пикселами в линии объекта A и темными пикселами соответствующей линии объекта B. Вот как здесь, — показали они мне пример успешных сопоставлений (рис. 8.4).

— Понятно, — сказал я. — Вы хотите сопоставить крупные темные области с крупными, а небольшие — с небольшими.

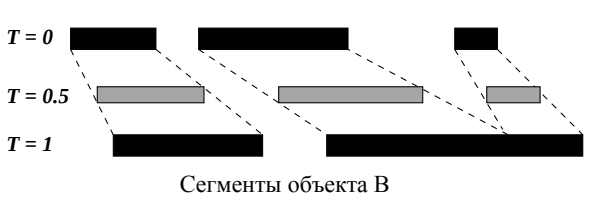


Рис. 8.4. Успешное сопоставление двух линий пикселов

— Да, при условии, что такое сопоставление не смещает их слишком сильно влево или вправо. В таком случае, вероятно, лучше слить области или разбить область на две, чем смещать их слишком далеко, т. к. это может привести к сопоставлению подбородка с бровями. Как нам лучше всего сделать это?

— Последний вопрос. Вам нужно будет сопоставлять два интервала таким образом, чтобы они пересекались?

— Думаю, что нет. Пересекающиеся интервалы нельзя сопоставлять. Это было бы равносильно обмену местами правого и левого глаза.

Я попытался изобразить задумчивость, но я не такой хороший актер, как Богарт. Идея решения этой задачи возникла у меня, лишь только они начали говорить о линиях пикселов. Они хотели преобразовать один массив пикселов в другой массив с минимальными изменениями. Это выглядело как редактирование строки пикселов, что является классическим применением динамического программирования. (Нечеткое сравнение строк обсуждается в *разделах 8.2* и *18.4*.)

То обстоятельство, что интервалы не могли пересекаться, окончательно утвердило эту идею в качестве решения. Это означало, что любая задача сопоставления полосы темных пикселов объекта A с полосой темных пикселов объекта B разбивалась на две меньшие подзадачи, т. е. на полосу пикселов слева и справа от сопоставляемого пиксела. Конечной стоимостью глобального сопоставления будет стоимость данного сопоставления плюс сумма значений стоимости сопоставлений всех пикселов справа и слева. Создание оптимального сопоставления левой части является меньшей и соответственно более легкой задачей. Кроме этого, может быть только $O(n^2)$ возможных подзадач для левой части, т. к. каждая полностью описывается парой из n верхних пикселов и n нижних пикселов.

— Для решения вашей задачи мы используем алгоритм динамического программирования, — заявил я. — Но решить ее можно несколькими способами, в зависимости от того, что вы хотите редактировать, пикселы или серии пикселов. Я бы, наверное, преобразовал каждую строку пикселов в список отсортированных по правой крайней точке серий черных пикселов. Каждая серия помечается ее начальной позицией и длиной. Поддерживается стоимость самого дешевого сопоставления между самыми левыми i сериями и самыми левыми j сериями для всех i и j. Используются следующие операции редактирования:

- *сопоставление всей серии.* Выполняется сопоставление верхней серии i с нижней серией j, при этом стоимость сопоставления будет зависеть от разницы в длине серий и их позиций;

- *слияние серий*. Несколько верхних серий сопоставляются с одной нижней серией. Стоимость этой операции будет зависеть от количества сливаемых серий, их относительных позиций и их длины;
- *разбиение серии*. Одна верхняя серия сопоставляется с несколькими последовательными нижними сериями. Это всего лишь операция, обратная операции слияния. Поэтому стоимость этой операции также будет зависеть от количества серий, их относительных позиций и их длины.

— Для каждой пары серий (i, j) и всех применимых случаев мы вычисляем стоимость операции редактирования и добавляем ее к уже вычисленной и сохраненной стоимости редактирования части, расположенной слева от начальной точки редактирования. Самый дешевый из этих случаев и будет определять стоимость операции редактирования $c[i, j]$.

Аспиранты все это записали в свои блокноты, а потом поинтересовались:

— Значит, стоимость сопоставления двух серий будет функцией их длины и позиции. Но как нам определять относительную стоимость?

— Это ваше дело. Динамическое программирование можно применить для оптимизации сопоставлений только тогда, когда мы знаем функции стоимости. Вы сами должны решить, какую стоимость устанавливать для операций изменений длины линий и смещения их позиций. Я рекомендую реализовать динамическое программирование и попробовать разные варианты стоимости на нескольких разных изображениях и выбрать из них такие, которые кажутся отвечающими вашим требованиям.

Они переглянулись, улыбнулись и помчались в лабораторию. Применив динамическое программирование для сопоставления частей изображения, они создали свою программу морфинга. Результаты работы этой программы можно видеть на рис. 8.5 на примере превращения омара в человеческое лицо.

К сожалению, они так и не удосужились превратить меня в Хамфри Богарта.

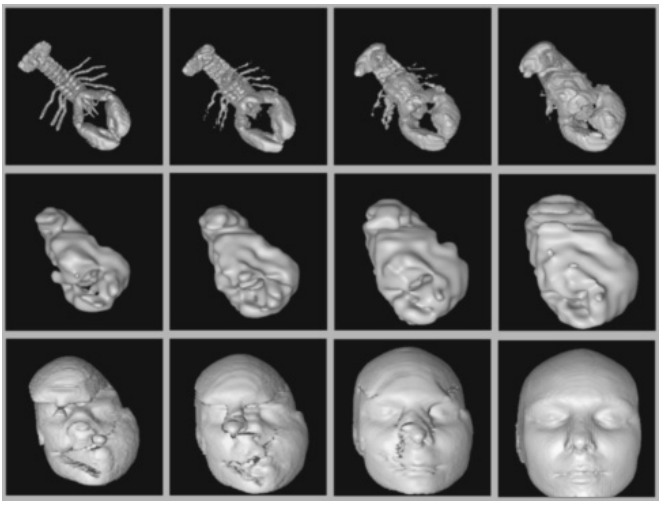


Рис. 8.5. Превращение омара в человеческое лицо с помощью динамического программирования

8.5. Задача разбиения

Допустим, что нужно просмотреть несколько книг на полке, чтобы найти определенную информацию. Для выполнения этого задания выделено трое сотрудников, каждому из которых будет дана определенная часть книг для просмотра. Чтобы сохранить начальный порядок книг, проще всего разделить полку на три части и назначить по части каждому сотруднику для просмотра.

Но как правильно распределить книги среди сотрудников? Если все книги имеют одинаковое количество страниц, то тогда это сделать очень легко: разделить все книги на три равные части. Например:

100 100 100 | 100 100 100 | 100 100 100

Таким образом, каждому сотруднику достанется три книги с общим количеством в триста страниц.

А если количество страниц в книгах не одинаковое? Допустим, что мы распределили таким же образом следующие книги:

100 200 300 | 400 500 600 | 700 800 900

Я лично предпочел бы просматривать первую часть, размером только в 600 страниц, но не последнюю, размером в 2 400 страниц. В этом случае самое справедливое распределение книг было бы следующим:

100 200 300 400 500 | 600 700 | 800 900

Таким образом, самая большая часть будет содержать 1 700 страниц, а самая меньшая — 1 300.

В общем виде задачу можно сформулировать так:

ЗАДАЧА. Разделение множества целых чисел на подмножества без перестановок.

Вход. Множество S положительных чисел $\{s_1, ..., s_n\}$ и целое число k.

Выход. Разделить множество S на k или меньше подмножеств таким образом, чтобы максимальная из сумм подмножеств была как можно меньше.

Данная задача, называемая задачей *линейного разбиения* (linear partition), часто возникает в параллельных процессах, когда нам нужно распределить нагрузку среди нескольких процессоров таким образом, чтобы минимизировать общее время исполнения.

Узким местом в таких вычислениях является процессор, которому назначено больше всего работы. В *разделе 7.10* было описано неудачное решение такой задачи.

Отложите книгу на несколько минут и попробуйте разработать алгоритм для решения задачи линейного разбиения.

Для начинающего алгориста наиболее естественным подходом к решению этой задачи может казаться использование эвристического алгоритма. Например, он может вычислить средний размер раздела, равный $\sum_{i=1}^{n} s_i / k$, после чего попытаться выполнить разбиение как можно ближе к этому среднему. Но такие эвристические методы решения обречены на провал при некоторых входных экземплярах, т. к. они не выполняют систематическую оценку всех возможных решений.

Вместо этого рассмотрим рекурсивный подход исчерпывающего поиска. Обратите внимание на то, что k-й раздел начинается после $(k-1)$-го разделителя. Где же мы можем поставить этот последний разделитель? Между i-м и $(i+1)$-м элементами для i такого, что $1 \le i \le n$. Каковы будут наши затраты на эту операцию? Общей стоимостью будет большая из двух величин: стоимость последнего раздела $\sum_{j=i+1}^{n} s_j$ или стоимость наибольшего раздела, созданного слева от i. Каким будет размер этого левого раздела? Чтобы минимизировать общую стоимость, нам нужно разделить $\{s_1, ..., s_i\}$ элементов, по возможности, на одинаковые части, используя оставшиеся $k-2$ разделителей. Но ведь это та же самая задача, только в меньшем экземпляре, и, значит, ее можно решить рекурсивно!

Исходя из этого, определим $M[n, k]$ как минимально возможную стоимость по всем операциям разбиения множества $\{s_1, ..., s_n\}$ на k подмножеств, где стоимость разбиения определяется как наибольшая сумма элементов в одной из ее частей. Определив таким образом эту функцию, мы можем ее вычислить:

$$M[n,k] = \min_{i=1}^{n} \max(M[i,k-1], \sum_{j=i+1}^{n} s_j$$

Для рекуррентного соотношения необходимо указать граничные условия. Эти граничные условия всегда устанавливают наименьшие возможные значения для всех аргументов. Для данной задачи наименьшим значением первого аргумента будет $n = 1$, что означает, что первый раздел состоит из одного элемента. Независимо от количества применяемых разделителей, первый раздел нельзя создать меньшим, чем s_1. Наименьшим значением второго аргумента будет $k = 1$, что означает, что множество S вообще не разбивается на подмножества. Таким образом:

$$M[1,k] = s_1 \text{ для всех } k > 0 \text{ и } M[n,1] = \sum_{i=1}^{n} s_i$$

По определению, это рекуррентное соотношение должно возвратить значение оптимального размера раздела. Сколько времени уйдет на вычисление, если сохранять частичные результаты? Всего в таблице имеется kn ячеек. Сколько времени займет вычисление результата $M[n', k']$? Для вычисления этого значения требуется найти минимальную из n' величин, каждая из которых является максимальной в таблице поиска и суммой, самое большее, n' элементов. Если заполнение каждой из kn ячеек занимает время n^2, то всю рекуррентность можно вычислить за время $O(kn^3)$.

Меньшие значения вычисляются раньше больших, с тем, чтобы на каждом шаге вычислений имелись необходимые данные. Реализация алгоритма приводится в листинге 8.16.

Листинг 8.16. Алгоритм решения задачи линейного разбиения

```
partition(int s[], int n, int k)
{
   int m[MAXN+1][MAXK+1];      /* Таблица значений */
   int d[MAXN+1][MAXK+1];      /* Таблица разделителей */
   int p[MAXN+1];              /* Массив префиксных сумм */
```

```
   int cost;                    /* Стоимость тестового разбиения */
   int i,j, x;                  /* Счетчики */
   p[0] = 0;                    /* Создаем префиксные суммы */
   for (i=1; i<=n; i++) p[i]=p[i-1]+s[i];
   for (i=1; i<=n; i++) m[i][1] = p[i];  /* Инициализируем границы */
   for (j=1; j<=k; j++) m[1][j] = s[1];
   for (i=2; i<=n; i++) /* Вычисляем основное рекуррентное
                           соотношение */
      for (j=2; j<=k; j++) {
         m[i] [j] = MAXINT;
         for (x=1; x<=(i-1); x++) {
            cost = max(m[x][j-1], p[i]-p[x]);
            if (m[i] [j] > cost) {
               m[i] [j] = cost;
               d[i][j] = x;
            }
         }
      }
   reconstruct_partition(s, d, n, k); /* Выводим решение разделения */
}
```

В действительности реализация из листинга 8.15 работает быстрее, чем мы рассчитали. При первоначальном анализе предполагалось, что обновление каждой ячейки матрицы занимает время $O(n^2)$. Это предположение основано на том обстоятельстве, что мы выбираем наилучшую из, самое большее, n возможных точек для размещения разделителя, для каждого из которых требуется сумма из, самое большее, n возможных членов. Но в действительности можно с легкостью избежать вычисления этих сумм, сохраняя набор из n префиксных сумм $p[i] = \sum_{k=1}^{i} s_k$, т. к. $\sum_{k=i}^{j} s_k = p[j] - p[k]$. Это позволяет вычислять каждую ячейку за линейное время, что дает время исполнения $O(kn^2)$.

Изучив рекуррентное соотношение и матрицы динамического программирования, представленные в табл. 8.4, вы должны убедиться в том, что конечное значение $M(n, k)$ будет стоимостью наибольшего диапазона в оптимальном разбиении.

Таблица 8.4. Матрицы динамического программирования для двух входных экземпляров

M	k		
n	1	2	2
1	1	1	1
1	2	1	1
1	3	2	1
1	4	2	2
1	5	3	2
1	6	3	2
1	7	4	3
1	8	4	3
1	9	5	3

D	k		
n	1	2	3
1			
1		1	1
1	–	1	2
1	–	2	2
1	–	2	3
1	–	3	4
1	–	3	4
1	–	4	5
1	–	4	6

M	k		
n	1	2	3
1	1	1	1
2	3	2	2
3	6	3	3
4	10	6	4
5	15	9	6
6	21	11	9
7	28	15	11
8	36	21	15
8	45	24	17

D	k		
n	1	2	3
1			
2		1	1
3	–	2	2
4	–	3	3
5	–	3	4
6	–	4	5
7	–	5	6
8	–	5	6
9	–	6	7

Но для большинства приложений нам требуется информация о том, как фактически выполнять само разбиение. Без этой информации мы, образно выражаясь, имеем лишь купон с большой скидкой на товар, которого нет на складе.

Для восстановления оптимального решения используется вторая матрица, *D*. При каждом обновлении значения *M*[*i, j*] мы записываем позицию разделителя, которая требовалась для получения этого значения. Чтобы восстановить путь, который привел к оптимальному решению, мы идем назад от *D*[*n, k*] и добавляем разделитель в каждой указанной позиции. Эту обратную трассировку лучше всего выполнять с помощью рекурсивной процедуры, показанной в листинге 8.17.

Листинг 8.17. Рекурсивная процедура восстановления решения

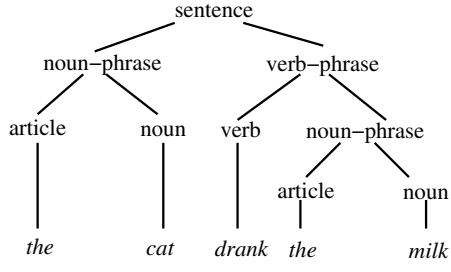
```
reconstruct_partition(int s[],int d[MAXN+1][MAXK+1], int n, int k)
{
   if   (k==1)
      print_books(s, 1, n);
   else {
      reconstruct_partition(s, d, d[n][k],k-1);
      print_books(s, d[n][k]+1,n);
   }
}
print_books(int s[], int start, int end)
{
int i; /* Счетчик */
for (i=start; i<=end; i++) printf("  %d  ",s[i]);
printf("\n");
}
```

8.6. Синтаксический разбор

При компилировании исходного кода программы компилятор определяет, отвечает ли структура программы требованиям синтаксиса данного языка программирования. В случае полного соответствия программа компилируется; в противном случае компилятор выдает сообщение об ошибке. Для этого требуется точное описание синтаксиса языка, которое обычно дается контекстно-свободной грамматикой, как показано в примере на рис. 8.6, *а*.

sentence ::= noun–phrase
verb–phrase
noun–phrase ::= article noun
verb–phrase ::= verb noun–phrase
article ::= *the, a*
noun ::= *cat, milk*
verb ::= *drank*

а)



б)

Рис. 8.6. Контекстно-свободная грамматика *(а)* и дерево синтаксического разбора *(б)*

Каждое *правило* (production) грамматики определяет интерпретацию именованного символа в левой части правила в виде последовательности символов в левой части правила. Правая сторона правила может содержать сочетание нетерминальных символов, или просто нетерминалов, (которые также определяются правилами) или терминальных символов, которые определяются просто как строки, например "the", "a", "cat", "milk" и "drank".

Синтаксический разбор (parsing) текстовой строки S согласно правилам контекстно-свободной грамматики G представляет собой алгоритмическую задачу создания дерева синтаксического разбора правил замены, определяющего строку S в виде единого нетерминального символа грамматики G. На рис. 8.6, *б* показано дерево синтаксического разбора простого предложения согласно правилам грамматики, представленной на рис. 8.6, *а*.

В студенческие времена синтаксический разбор казался мне ужасно сложным предметом. Но несколько лет тому назад один приятель без труда объяснил мне его за ланчем. Правда, теперь я понимаю динамическое программирование намного лучше.

Мы полагаем, что длина текстовой строки S равна n символам, а сама грамматика G имеет постоянный размер. Такое предположение является справедливым, т. к., независимо от размера компилируемой программы, грамматика определенного языка программирования (например, C или Java) имеет фиксированный размер.

Кроме этого, мы полагаем, что определения каждого правила представлены в *нормальной форме Хомского* (Chomsky normal form). Это означает, что правая сторона каждого нетривиального правила состоит из двух нетерминальных символов, т. е. $X \rightarrow YZ$, или одного терминального символа, т. е. $X \rightarrow \alpha$. Любую контекстно-свободную грамматику можно механически преобразовать в нормальную форму Хомского, многократно сокращая длинные правые стороны за счет добавления дополнительных нетерминальных символов и правил. Таким образом, данное предположение не приводит к потери общности.

Как можно выполнить эффективный синтаксический разбор строки S, используя контекстно-свободную грамматику, в которой каждое представляющее интерес правило состоит из двух нетерминальных символов? Здесь ключевым является то обстоятельство, что правило, применяемое в корне дерева синтаксического разбора (скажем, $X \rightarrow YZ$), разделяет строку S в позиции i таким образом, что левая часть строки ($S[1, i]$) генерируется нетерминальным символом Y, а правая часть ($S[i + 1, n]$) генерируется элементом Z.

Это обстоятельство наводит на мысль об использовании динамического программирования, в котором мы отслеживаем все нетерминальные символы, генерируемые каждой подстрокой строки S. Определяем булеву функцию $M[i, j, X]$, которая возвращает значение ИСТИНА тогда и только тогда, когда подстрока $S[i, j]$ генерируется нетерминальным символом X. Это происходит тогда, когда существует правило $X \rightarrow YZ$ и такая точка разрыва k между i и j, что левая часть генерирует Y, а правая — Z. Иными словами,

$$M[i,j,X] = \bigvee_{(X \rightarrow YZ) \in G} \left(\bigvee_{i=k}^{j} M[i,k,Y] \cdot M[k+1,j,Z] \right)$$

где символ V означает логическое ИЛИ по всем правилам и позициям разделения, а символ $\vee$ означает операцию логического И над двумя булевыми значениями.

Однобуквенные терминальные символы определяют граничные условия рекуррентности. В частности, $M[i, i, X]$ определяется как ИСТИНА тогда и только тогда, когда существует такое правило $X \rightarrow a$, что $S[i] = a$.

Какова временная сложность этого алгоритма? Размер пространства состояний равен $O(n^2)$, т. к. существует $n(n + 1)/2$ подстрок, определяемых парами (i, j). Умножение этого значения на количество нетерминальных символов (скажем, v) не влияет на асимптотический верхний предел, т. к. грамматика определена постоянного размера. Чтобы вычислить значение $M[i, j, X]$, нужно протестировать все промежуточные значения k, так что вычисление каждой из $O(n^2)$ ячеек занимает время $O(n)$. Отсюда получаем кубическую, т. е. $O(n^3)$, временную сложность алгоритма синтаксического разбора.

Остановка для размышлений. Экономичный синтаксический разбор

ЗАДАЧА. Нередко в программах имеются синтаксические ошибки, из-за которых программа не компилируется. Для данной контекстно-свободной грамматики G и строки ввода S определить наименьшее количество замен символов, которые нужно выполнить в строке S, чтобы конечная строка была приемлемой для грамматики G.

Решение. Когда я впервые столкнулся с этой задачей, она казалась чрезвычайно трудной. Но после некоторых размышлений я пришел к выводу, что она является общим случаем задачи вычисления расстояния редактирования, для решения которой динамическое программирование подходит самым естественным образом. Синтаксический разбор также вначале казался трудной задачей, но потом поддался решению тем же методом. Действительно, мы можем решить объединенную задачу, обобщив отношение рекуррентности, которое мы использовали для выполнения простого синтаксического разбора.

Определим *целочисленную функцию* $M'[i, j, X]$, которая возвращает количество минимальных изменений в строке $S[i, j]$, позволяющих создать ее нетерминальным символом X. Этот символ будет создаваться правилом $x \rightarrow yz$. Некоторые изменения подстроки s могут находиться слева от точки раздела, а другие — справа, но нас интересует только минимизация суммы. Иными словами,

$$M'[i, j, X] = \min_{(X \rightarrow YZ) \in G} \left(\min_{i=k}^{j} M'[i, k, Y] + M'[k + 1, j, Z] \right)$$

Также слегка изменяются граничные условия. Если существует правило $X \rightarrow a$, то стоимость сравнения в позиции i зависит от содержимого $S[i]$, где $S[i] = a$, $M[i, i, X] = 0$. В противном случае требуется одна замена, поэтому $M[i, i, X] = 1$, если $S[i] \neq a$. Но если в грамматике отсутствует правило $X \rightarrow a$, то не существует способа выполнить замену в односимвольной строке, чтобы получить что-либо, создающее X, поэтому $M[i, i, X] = \infty$ для всех i. ■

8.6.1. Триангуляция с минимальным весом

То же самое рекуррентное соотношение, используемое в алгоритме синтаксического разбора, можно также использовать для решения интересной задачи вычислительной

геометрии. Триангуляцией многоугольника $P = \{v_1, ..., v_n, v_1\}$ называется набор непересекающихся диагоналей, которые разделяют данный многоугольник на треугольники. *Весом* триангуляции называется сумма длин всех ее диагоналей. Как показано на рис. 8.7, для любого многоугольника может существовать несколько разных триангуляций.

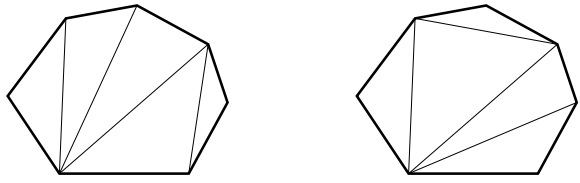


Рис. 8.7. Две разные триангуляции для выпуклого семиугольника

Нам нужно найти для данного многоугольника p триангуляцию с минимальным весом. Триангуляция является базовым компонентом большинства геометрических алгоритмов (см. *раздел 17.3*).

Чтобы применить динамическое программирование для решения этой задачи, нам нужно найти способ разбить многоугольник на меньшие части. Обратите внимание на то, что каждая сторона многоугольника должна входить ровно в один треугольник. Чтобы создать треугольник на основе этой стороны, необходимо определить третью вершину этого треугольника, как показано на рис. 8.8.

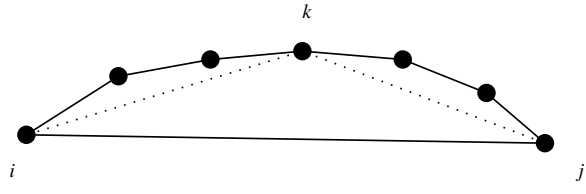


Рис. 8.8. Выбор вершины k для стороны (i, j) многоугольника

Когда мы определим правильную объединяющую вершину, многоугольник будет разбит на два меньших многоугольника, для каждого из которых нужно будет выполнить оптимальную триангуляцию. Пусть $T[i, j]$ обозначает стоимость триангуляции от вершины v_i к вершине v_j без учета длины хорды d_{ij} между v_i и v_j. Последнее обстоятельство позволяет избежать двойного подсчета этих внутренних хорд в следующем рекуррентном соотношении:

$$T[i,j] = \min_{k=i+1}^{j-1}(T[i,k] + T[k,j] + d_{ik} + d_{kj})$$

Это основное условие применимо, когда вершины i и j являются непосредственными соседями, т. к. $T[i, i + 1] = 0$.

Поскольку количество вершин в каждом поддиапазоне правой части отношения меньше, чем в левой части, вычисление может выполняться на интервале между i и j. Псевдокод соответствующего алгоритма приводится в листинге 8.18.

Листинг 8.18. Алгоритм триангуляции

```
Minimum-Weight-Triangulation(P)
    for i = 1 to n − 1 do T[i, i + 1] = 0
    for gap = 2 to n − 1
        for i = 1 to n − gap do
            j = i + gap
            T[i, j] = min_{k=i+1}^{j−1}(T[i, k] + T[k, j] + d_{P_i,P_k} + d_{P_k,P_j})
    return T[1, n]
```

Триангуляция T имеет $\binom{n}{2}$ значений, вычисление каждого из которых занимает время $O(j-i)$, если вычислять секции в порядке возрастания размера. Так как $j-i=O(n)$, то сложность по времени полного вычисления будет $O(n^3)$, а по памяти — $O(n^2)$.

Что произойдет, если точки имеются внутри многоугольника? В таком случае динамическое программирование нельзя применить таким же образом, т. к. ребра триангуляции не обязательно разбивают многоугольник на две разные части. Теперь вместо только $\binom{n}{2}$ возможных поддиапазонов их количество возрастет экспоненциально. Более того, известно, что общий случай данной задачи является NP-полной.

Подведение итогов

Для любой задачи оптимизации объектов, упорядоченных слева направо, например, символов строки, элементов перестановки, вершин многоугольника или листьев дерева поиска, применение динамического программирования, скорее всего, позволит создать эффективный алгоритм для получения оптимального решения.

8.7. Ограничения динамического программирования. Задача коммивояжера

Динамическое программирование подходит не для всех задач. Важно понимать, почему его применение может не дать желаемого результата, и уметь избегать ситуаций, чреватых появлением неправильного или неэффективного алгоритма.

Материалом для наших алгоритмических экспериментов снова будет задача коммивояжера, в которой мы ищем самый короткий маршрут для посещения всех городов. Но мы ограничимся следующим случаем:

ЗАДАЧА. Самый длинный простой путь.

Вход. Взвешенный граф G, в котором указаны начальная (s) и конечная (t) вершины.

Выход. Самый длинный маршрут от s до t, в котором все вершины посещаются только один раз.

Между этой задачей и задачей коммивояжера имеется два несущественных различия. Во-первых, требуется найти путь, а не замкнутый маршрут. Эта разница несущественна, т. к. мы можем получить замкнутый маршрут, просто включив в путь ребро (t, s). Во-вторых, требуется найти наиболее длинный путь, а не наиболее короткий маршрут.

Опять же, разница не является существенной: нам просто требуется посетить как можно больше вершин (в идеале, все), точно так же, как и в задаче коммивояжера. Ключевым словом в постановке данной задачи является слово *простой*, означающее, что любую вершину мы можем посетить только один раз.

Для невзвешенных графов (в которых вес каждого ребра равен единице) самый длинный путь от s к t будет равным $n - 1$. Задача поиска таких Гамильтоновых путей (если они существуют) является важной задачей теории графов, которая рассматривается в *разделе 16.5*.

8.7.1. Вопрос правильности алгоритмов динамического программирования

Алгоритмы динамического программирования правильны лишь настолько, насколько правильны рекуррентные соотношения, на которых они основаны. Допустим, мы определим функцию $LP[i, j]$ как длину максимального простого пути от вершины i к вершине j. Обратите внимание на то, что самый длинный простой путь от вершины i к вершине j перед тем, как попасть в вершину j. должен пройти через некую вершину x. Таким образом, последним ребром пути должно быть ребро (x, j). Здесь напрашивается следующее рекуррентное соотношение для вычисления длины максимального пути, где $c(x, j)$ — вес ребра (x, j):

$$LP[i, j] = \max_{(x,j)\in E} LP[i, x] + c(x, j)$$

Идея кажется разумной, однако я вижу в ней, по крайней мере, два недостатка.

Прежде всего, в этом рекуррентном соотношении не предусмотрено ничего для обеспечения простоты. Откуда мы знаем, что вершина j не использовалась ранее в самом длинном простом пути от вершины i к вершине x? Если использовалась, то добавление ребра (x, j) создаст цикл. Чтобы предотвратить такое развитие событий, мы должны определить такое рекуррентное соотношение, которое помнит пройденный путь. Возможно, мы могли бы достичь этого, определив функцию $LP'[i, j, k]$ как самый длинный путь от вершины i к вершине j, не включающий вершину k? Это был бы шаг в правильном направлении, но он все равно не даст нам работающее должным образом рекуррентное соотношение.

Вторая проблема затрагивает порядок вычисления. Какой элемент мы вычисляем первым? Так как вершины графа не упорядочены слева направо или в порядке возрастания, то неясно, какими должны быть меньшие подпрограммы. При отсутствии такого упорядочивания мы легко окажемся в бесконечном цикле.

Динамическое программирование можно использовать для решения любой задачи, в которой соблюдается *принцип оптимальности*. Иными словами, это означает, что множество частичных решений может быть оптимально расширено с учетом *состояний* после частичных решений, а не специфики самих частичных решений. Например, когда мы решали, расширять ли нечеткое сравнение строк за счет замены, вставки или удаления, нам не требовалось знать, какая последовательность операций была выполнена до этого момента. В действительности может существовать несколько разных последовательностей редактирования, которые выдают стоимость C на первых p симво-

лах строки образца P и t символах строки T. Очередные решения принимаются на основе *последствий* предыдущих решений, а не на основе самих предыдущих решений.

Когда важна не стоимость выполняемых операций, а их специфика, задача не удовлетворяет принципу оптимальности. В качестве примера такой задачи можно привести разновидность задачи вычисления расстояния редактирования, в которой не разрешается использовать комбинации операций в определенных последовательностях. Но при правильной постановке задачи принцип оптимальности соблюдается во многих комбинаторных задачах.

8.7.2. Эффективность алгоритмов динамического программирования

Время исполнения любого алгоритма динамического программирования зависит от двух факторов: количества частичных решений, которые необходимо отслеживать, и длительности вычисления каждого частичного решения. Обычно более важным является первый аспект, а именно, размер пространства состояний.

Во всех приведенных здесь примерах частичные решения полностью описываются посредством указания *точек остановки* при вводе. Это объясняется тем, что элементы обрабатываемых комбинаторных объектов (строк, числовых последовательностей или многоугольников) неявно упорядочены. Это упорядочивание нельзя нарушить, не изменив при этом полностью всю задачу. При установленном порядке элементов существует сравнительно небольшое количество возможных точек остановки, что позволяет получить эффективные алгоритмы.

В случае же отсутствия жесткой упорядоченности объектов количество возможных частичных решений растет экспоненциально. Допустим, что состоянием нашего частичного решения является весь путь P от начальной до конечной вершины. Таким образом, $LP[i, j, P]$ представляет самый длинный простой путь от вершины i к вершине j, где P обозначает точную последовательность промежуточных вершин в этом пути. Это можно вычислить с помощью следующего рекуррентного соотношения, в котором $P + x$ обозначает присоединение вершины x в конец пути P:

$$LP[i, j, P + x] = \max_{(x,j)\in E, x, j\notin P} LP[i, x, P] + c(x, j)$$

Эта формулировка корректна, но насколько она эффективна? Путь P является упорядоченной последовательностью из не более $n - 3$ вершин. Количество таких путей доходит до $(n - 3)!$, а это очень много! Фактически, этот алгоритм использует комбинаторный поиск (наподобие поиска с возвратом) для создания всех возможных промежуточных путей. В действительности, функция max отчасти вводит в заблуждение, т. к. для создания состояния $LP[i, j, P + x]$ возможно только одно значение x и одно значение P.

Но этой идее можно найти лучшее применение. Пусть $LP'[i, j, S]$ представляет самый длинный простой путь от вершины i к вершине j, промежуточными вершинами которого являются как раз те, что образуют *подмножество* S. Таким образом, если $S = \{a, b, c\}$, то существует шесть путей, совместимых с S: *iabcj, iacbj, ibacj, ibcaj, icabj* и *icbaj*. Размер этого пространства состояний может быть максимум 2^n, что меньше,

чем перечисление всех путей. Кроме этого, эту функцию можно вычислить с помощью следующего рекуррентного соотношения:

$$LP'[i, j, S \cup \{x\}] = \max_{(x,j) \in E, x, j \notin S} LP[i, x, S] + c(x, j)$$

где $S \cup \{x\}$ обозначает объединение S и x.

После этого самый длинный простой путь от вершины i к вершине j можно найти, максимизировав по всем возможным промежуточным подмножествам вершин:

$$LP[i, j] = \max_{S} LP'[i, j, S]$$

Существует только 2^n подмножеств множества n вершин, так что это большое улучшение по сравнению с перечислением всех $n!$ маршрутов. На практике этот метод можно было бы с уверенностью использовать для решения задачи коммивояжера с количеством вершин, близким к 30, в то время как значение $n = 20$ будет неприемлемым для алгоритма с временной сложностью $O(n!)$. Тем не менее, динамическое программирование наиболее эффективно на множестве хорошо упорядоченных объектов.

Подведение итогов

При отсутствии естественного упорядочивания слева направо алгоритм, использующий динамическое программирование, обычно обречен на экспоненциальную сложность, как по времени, так и по памяти.

8.8. История из жизни. Динамическое программирование и язык Prolog

— Наш эвристический алгоритм очень хорошо проявляет себя на практике. — Мой коллега одновременно хвастался и просил о помощи.

Унификация является основным вычислительным механизмом в языках логического программирования, таких как Prolog. Программа на языке Prolog состоит из набора правил, где каждое правило имеет голову и действие, которое выполняется, когда голова правила совпадает или объединяется с текущим вычислением.

Выполнение программы на языке Prolog начинается с указания цели, например, $p(a, X, Y)$, где a является константой, а X и Y — переменными. После этого система систематически сравнивает голову цели с головой каждого правила, которое можно унифицировать с целью. Под унификацией имеется в виду привязка переменных к константам, если их можно соотнести. В представленной в листинге 8.19 бессмысленной программе на языке Prolog цель $p(X, Y, a)$ унифицируется с любым из первых двух правил, т. к. для X и Y можно выполнить привязку, чтобы сопоставить дополнительные символы. Цель $p(X, X, a)$ сопоставится только с первым правилом, т. к. привязываемые к первой и второй позиции переменные должны быть одинаковыми.

Листинг 8.19. Пример программы на языке Prolog

```
p(a,a,a) := h(a);
p(b,a,a) := h(a) * h(b);
p(c,b,b) :=h(b) + h(c);
p(d,b,b):=h(d) + h(b);
```

— Чтобы ускорить операции унификации, мы хотим выполнить предварительную обработку набора голов правил, чтобы можно было бы быстро определить, какие правила соответствуют данной цели. Для этого нам нужно организовать правила в нагруженном дереве (trie).

Нагруженные деревья (см. *раздел 12.3*) являются полезными структурами данных для работы со строками. Каждый лист нагруженного дерева представляет одну строку. Каждый узел на пути от корня к листу маркируется одним символом строки, при этом *i*-й узел пути соответствует *i*-му символу строки.

— Согласен. Нагруженное дерево является естественным способом для представления ваших голов правил. Создание нагруженного дерева для набора строк символов является прямолинейной задачей: мы просто вставляем строки, начиная с корня. Так в чем заключается ваша проблема? — спросил я.

— Эффективность нашего алгоритма унификации очень зависит от минимизирования количества ребер в нагруженном дереве. Так как мы знаем все правила наперед, то у нас имеется свобода действий, чтобы переупорядочить позиции символов в правилах. Например, вместо того, чтобы корневой узел всегда представлял первый аргумент правила, мы можем избрать, чтобы он представлял третий аргумент. Мы бы хотели воспользоваться этой свободой действий для того, чтобы создать нагруженное дерево минимального размера для набора правил.

Мой собеседник показал мне пример (рис. 8.9).

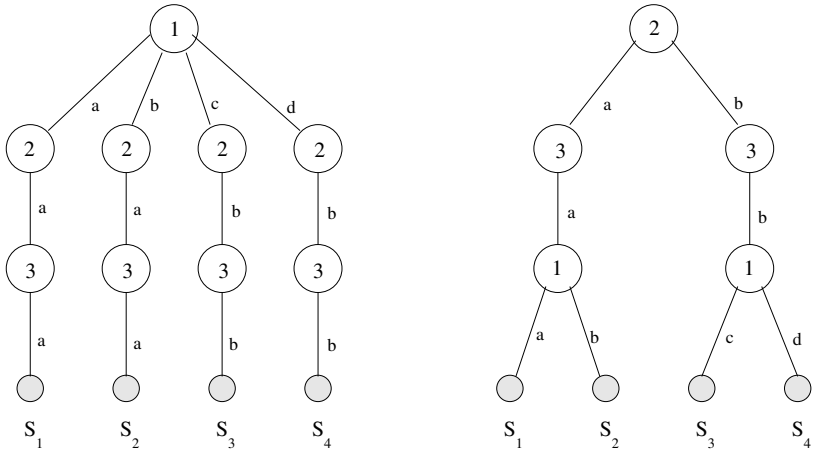


Рис. 8.9. Два разных нагруженных дерева для одного и того же набора правил

В нагруженном дереве, созданном согласно первоначальному порядку позиций строк (1, 2, 3), всего используется 12 ребер. Но переставив позиции символов в (2, 3, 1), мы можем получить нагруженное дерево лишь с 8 ребрами.

— Интересно... — начал было я отвечать, но он снова прервал меня.

— Есть еще одно ограничение. Листья нагруженных деревьев необходимо содержать упорядоченными, чтобы листья основного дерева располагались слева направо в том же самом порядке, в каком правила выводятся на страницу.

— Но почему листья нагруженных деревьев должны содержаться в заданном порядке? — спросил я.

— Порядок правил в программах Prolog имеет очень, очень большую важность. Если изменить порядок правил, то программа возвратит другой результат.

Потом он перешел к описанию того, что им требовалось от меня.

— У нас есть "жадный"" эвристический алгоритм для создания хороших, но не оптимальных, нагруженных деревьев. Этот алгоритм основан на выборе в качестве корня дерева символа в такой позиции, которая минимизирует степень корня. Иными словами, алгоритм выбирает такую позицию символа, в которой имеется наименьшее число разных символов. Этот эвристический алгоритм работает исключительно хорошо на практике. Но нам нужно доказать, что задача построения наилучшего нагруженного дерева является NP-полной, чтобы наша статья была полностью завершена. Вот в этом нам и нужна ваша помощь.

Я пообещал доказать, что задача имеет такой уровень трудности. Казалось, что для построения минимального дерева, в самом деле, требовалось использовать какую-то нетривиальную комбинаторную оптимизацию, но я не видел, каким образом можно было бы включить слева направо упорядочивание правил в доказательство сложности. Более того, я не мог припомнить ни одной NP-полной задачи, содержащей такое ограничение в виде упорядочивания слева направо. В конце концов, если бы данный набор правил содержал позицию символа, общую для всех правил, то она должна бы быть исследованной первой в любом дереве минимального размера. Так как правила были упорядоченными, то каждый узел в поддереве должен представлять корень серии последовательных правил. Таким образом, существовало только $\binom{n}{2}$ узлов, которые было возможно выбрать из этого дерева...

Есть! Это и было ключевым аспектом решения.

На следующий день я снова встретился с профессором. — Я не могу доказать, что ваша задача является NP-полной. Но что Вы скажете по поводу эффективного алгоритма динамического программирования для построения наилучшего нагруженного дерева? — Я с удовольствием наблюдал, как недовольное выражение его лица сменилось улыбкой, когда он осознал, о чем идет речь. Эффективный алгоритм для получения требуемого решения было значительно лучше, чем доказательство невозможности такого решения.

Мое рекуррентное соотношение работало таким образом. Допустим, что у нас имеется n упорядоченных голов правил, каждая из которых имеет m аргументов. Выборка в p-й позиции, $1 \le p \le m$, разделяет головы правил на серии R_1, ..., R_r, где каждое правило в отдельной серии $R_x = s_1$, ..., s_j имеет одинаковое значение символа $s_i[p]$. Правила в каждой серии должны быть последовательными, поэтому существует только $\binom{n}{2}$ возможных серий, о которых нужно заботиться. Стоимостью выборки в позиции p является стоимость завершения обработки деревьев, формируемых каждой созданной серией, плюс одно ребро для каждого дерева, чтобы связать его с зондированием p:

$$C[i,j] = \min_{p=1}^{m} \sum_{k=1}^{r} (C[i_k, j_k] + 1)$$

Один из аспирантов немедленно приступил к реализации этого алгоритма, чтобы можно было сравнить его работу с работой эвристического алгоритма заказчика. На многих входных экземплярах как оптимальный, так и "жадный" алгоритмы создавали одинаковое нагруженное дерево. Но для некоторых экземпляров производительность алгоритма динамического программирования была на 20% лучше, чем производительность "жадного" алгоритма, т. е. на 20% лучше, чем "очень хорошая работа" на практике. Время компилирования алгоритма динамического программирования было несколько большим, чем "жадного" алгоритма, но при оптимизации компилирования всегда лучше обменять немного большее время компиляции на лучшее время исполнения программы. Стоит ли 20% улучшения производительности этих усилий? Это зависит от конкретной ситуации. Насколько полезной была бы для вас 20-процентная надбавка в вашей зарплате?

То обстоятельство, что правила должны были оставаться упорядоченными, было решающим фактором, который мы использовали в решении с применением динамического программирования. Фактически при отсутствии этого обстоятельства я смог доказать, что задача в самом деле была NP-полной.

Подведение итогов

Глобальное оптимальное решение (полученное, например, с помощью динамического программирования) часто заметно лучше, чем решение, полученное посредством типичного эвристического алгоритма. Насколько важным является такое улучшение, зависит от конкретного приложения, но оно никогда не будет лишним.

8.9. История из жизни. Сжатие текста для штрих-кодов

Инджиун (Ynjiun) провел лазерной указкой по рваным и смятым кускам этикетки со штрих-кодом. Через несколько секунд система выдала ответ. Он победно усмехнулся. — Практически безотказно.

Мне показывали свои достижения работники научно-исследовательского центра компании Symbol Technologies, ведущего мирового производителя оборудования для сканирования штрих-кодов. В следующий раз, когда вы будете стоять в очереди к кассиру в продуктовом магазине, обратите внимание на тип используемого ими сканирующего оборудования. Скорее всего, на корпусе будет маркировка *Symbol*.

Хотя мы принимаем штрих-коды как должное, работа с ними требует на удивление сложной технологии. Надобность в штрих-кодах существует потому, что обычные системы оптического распознавания символов не обеспечивают достаточной надежности для операций с товарно-материальными запасами. Технология штрих-кодов, знакомая каждому из нас по их присутствию на каждой пачке овсянки и упаковке жевательной резинки, позволяет закодировать десятизначный номер с уровнем коррекции, делающим практически невозможными ошибки при сканировании, даже если консервная банка со штрих-кодом перевернута вверх ногами или деформирована. Иногда кассиру не удается отсканировать штрих-код, но если вы слышите характерный звук аппарата, то вы знаете, что код был считан правильно.

Десять цифр кода обычной этикетки со штрих-кодом предоставляют возможность записать на ней только идентификационный номер товара. Вследствие этого любое приложение, занимающееся обработкой штрих-кодов, должно использовать базу данных, соотносящую штрих-код с соответствующим товаром. В течение долгого времени заветной целью штрих-кодовой индустрии была разработка более емкой штрих-кодовой символики, позволяющей кодировать целые документы и обеспечивающей их надежное воспроизведение.

— PDF-417 является нашей новой, двумерной штрих-кодовой технологией, — объяснил Инджиун. (Пример этикетки со штрих-кодом этого типа показан на рис. 8.10.)

— Какой объем информации можно уместить с помощью этой технологии на одном квадратном дюйме? — спросил я у него.

— Это зависит от заданного уровня коррекции ошибок, но в общем около 1 000 байтов, что достаточно для небольшого текстового файла или изображения, — ответил он.

— Вам, наверное, приходится использовать какую-либо технологию сжатия данных, чтобы максимизировать объем сохраняемого на этикетке текста. (Стандартные алгоритмы сжатия данных рассматриваются в *разделе 18.5*.)

— Да, мы действительно используем определенный способ сжатия данных, — согласился Инджиун. — Мы знаем, какие типы текста наши клиенты будут помещать на этикетки. Некоторые тексты будут состоять полностью из прописных букв, а другие — из букв обоих регистров и цифр. Наш код предоставляет три разных текстовых режима, каждый из которых поддерживает отдельное подмножество алфавитно-цифровых знаков. Мы можем описать каждый знак, используя только пять битов, при условии, что режим не меняется. Для описания знака из другого режима мы сначала выдаем команду переключения режимов (длиной в пять битов), а потом код символа.

— Понятно. Значит, вы определили для каждого режима группы символов таким образом, чтобы свести к минимуму операции переключения режимов при работе с типичными текстовыми файлами. (Схема переключения режимов для разных групп символов показана на рис. 8.11.)

Рис. 8.10. Этикетка с Гетгисбергским посланием, закодированным двумерным штрих-кодом по технологии PDF-417

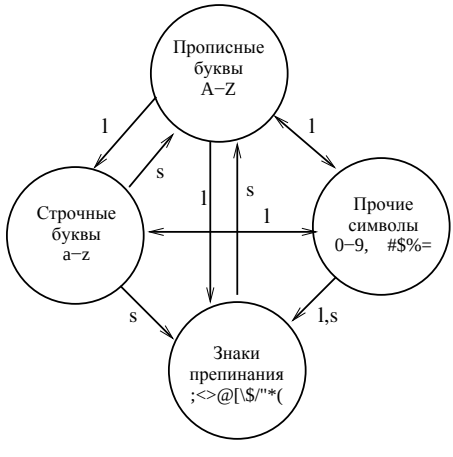


Рис. 8.11. Схема переключения режимов в PDF-4 1 7

— Совершенно верно. Мы поместили все цифры в один режим, а все символы знаков препинания в другой. Мы также реализовали команды *переключения* (switch) и *фиксирования* (latch) режимов. Операция переключения режимов применяется для переключения в новый режим только для следующего символа, скажем, знака препинания. Таким образом мы можем избежать затрат на возвращение в текстовый режим после вывода знака препинания. А операция фиксирования режимов используется для постоянного переключения в новый режим, если в этом режиме нужно вывести несколько символов подряд.

— Интересно! — сказал я. — При таком переключении режимов должно существовать множество разных способов закодировать текст в виде штрих-кода. Как вы получаете закодированный текст наименьшего объема?

— Мы используем "жадный" алгоритм, который выполняет просмотр на несколько символов вперед и решает, какой режим лучше всего использовать. Этот способ работает довольно хорошо.

Я продолжал выпытывать у него подробности по этому вопросу. — Откуда вы знаете, что этот способ работает хорошо? Ведь может существовать значительно лучшая кодировка, которую вы просто не находите.

— Вообще говоря, мы этого не знаем. Но поиск оптимальной кодировки, вероятно, является NP-полной задачей. Не так ли? — спросил он упавшим голосом.

Я задумался. Каждая кодировка начинается в определенном режиме и состоит из последовательности кодов символов и операций переключения и фиксирования режимов. В любой позиции в тексте мы можем помещать код следующего символа (если он имеется в текущем режиме) или выполнять операцию переключения или фиксирования режима. При продвижении в тексте слева направо текущее состояние полностью отражается текущей позицией символа и текущим состоянием режима. Для данной пары "позиция/режим" нас интересует самая дешевая из всех возможных кодировок достижения этой точки...

Я почувствовал сильное волнение.

— Оптимальную кодировку для любого текста в PDF-417 можно определить с помощью динамического программирования. Для каждого возможного режима $1 \le m \le 4$ и для каждой возможной позиции символа $1 \le i \le n$ мы будем сопровождать информацию о самой дешевой кодировке, найденной для первых i символов и заканчивающейся в режиме m. Нашим следующим ходом из каждой пары состояний "позиция/режим" является или сопоставление, или переключение или фиксирование режима, так что рассмотрению подлежит очень небольшое число возможных операций.

По сути, мы имеем следующее соотношение:

$$M[i,j] = \min_{1 \le m \le 4} (M[i-1,m] + c(S_i, m, j))$$

где $c(S_i, m, j)$ представляет стоимость кодирования символа S_i и переключения из режима m в режим j. Самая дешевая возможная кодировка получается в результате обратной трассировки от узла $M[n, m]$, где m представляет значение i, которое минимизирует $\min_{1 \le i \le 4} M[n, i]$. Каждую из $4n$ ячеек можно заполнить за постоянное время, так

что оптимальную кодировку можно найти за линейное по отношению к длине строки время.

Инджиун отнесся к этому несколько скептически, но попросил, чтобы мы реализовали для него оптимальный метод кодирования. У нас возникли определенные затруднения в связи с некоторыми странностями переключения режимов, но мой студент Йо-Линг Лин (Yaw-Ling Lin) оказался на высоте и решил эту задачу. В *Symbol* сравнили наш кодировщик со своим, обработав для этого 13 000 этикеток, и пришли к выводу, что динамическое программирование, в среднем, позволяло поместить на этикетку на 8% больше информации. Это было важным улучшением, т. к. никому не хочется терять 8% емкости хранения, особенно в ситуации, когда максимальная емкость хранения составляет всего лишь несколько сот байтов. Для определенных приложений эти лишние 8% позволили использовать только одну этикетку, когда раньше требовалось две. Конечно же, *среднее* улучшение на 8% означало, что для определенных типов информации улучшение будет намного большим. В то время как наш кодировщик работал чуть медленнее, чем кодировщик на базе "жадного" алгоритма, это не представляло важности, т. к. в любом случае распечатка этикеток была бы узким местом.

Наблюдаемый нами эффект замены эвристического алгоритма алгоритмом поиска глобального оптимального решения, пожалуй, является типичным для большинства приложений. В общем, если вы не допустите серьезных ошибок в реализации эвристического алгоритма, то, скорее всего, он будет выдавать приемлемое решение. Но использование вместо него алгоритма поиска оптимального результата обычно дает хоть и небольшое, но немаловажное улучшение, которое может иметь положительный эффект для вашего приложения.

Замечания к главе

Методика динамического программирования впервые представлена в [Bel58]. Алгоритм определения расстояния редактирования первоначально был представлен Вагнером и Фишером (Wagner, Fischer) в книге [WF74]. Более быстрый алгоритм для задачи разделения книг рассматривается в книге [KMS97].

Вычислительная сложность триангуляции с минимальным весом несвязного набора точек (в отличие от многоугольников) долго оставалась нерешенной задачей, пока, наконец, не была решена Мулцером и Роте (Mulzer, Rote) (см. книгу [MR06]).

Такие методики, как динамическое программирование и поиск с возвратом, можно использовать для создания эффективных (хотя все же не с полиномиальным временем исполнения) алгоритмов для решения многих NP-полных задач. Обзор этих методик см. в книге [Woe03].

Система морфинга, которая была описана в истории из жизни в *разделе 8.4*, более подробно обсуждается в книге [HWK94]. Дополнительную информацию по задаче минимизации нагруженных деревьев в Prolog, которая рассматривалась в *разделе 8.8*, можно найти в нашем докладе [DRR + 95]. Двумерные штрих-коды, рассматриваемые в *разделе 8.9*, были разработаны в значительной степени усилиями Тео Павлидиса (Theo Pavlidis) и Инджиуна Ванга (Ynjiun Wang) в университете Stoney Brook. Подробности см. в [PSW92].

Алгоритм динамического программирования, рассматриваемый для решения задачи синтаксического анализа, называется CKYalgonthm, по именам его трех независимых разработчиков: Коки, Касами и Янгера (Cocke, Kasami, Younger). Подробности см. в [You67]. Обобщение синтаксического анализа для задачи вычисления расстояния редактирования рассматривается Ахо и Петерсоном (Aho, Peterson) в [AP72].

8.10. Упражнения

Расстояние редактирования

1. [3] При вводе текста часто допускаются ошибки перестановки соседних символов, например, когда вместо "Steve" вводится "Setve". Согласно традиционному определению расстояния редактирования для исправления таких ошибок требуется две замены.

   Включите в нашу функцию расстояния редактирования операцию обмена, чтобы такие ошибки перестановки можно было бы исправить за одну операцию.

2. [4] Дано три строки символов: X, Y и Z, где $|X| = n$, $|Y| = m$ и $|Z| = n + m$. Строка Z называется *перетасовкой* (shuffle) строк X и Y тогда и только тогда, когда ее можно создать, перемешивая символы строк X и Y таким образом, чтобы в получившейся строке сохранялся первоначальный (слева направо) порядок исходных строк.

   а) Докажите, что строка "cchocohilaptes" является перетасовкой строк "chocolate" и "chips", но строка "chocochilatspe" — нет.

   б) Предоставьте эффективный алгоритм динамического программирования для определения, является ли строка Z перетасовкой строк X и Y. (Подсказка: значения создаваемой вами матрицы динамического программирования должны быть булевыми, а не числовыми.)

3. [4] Самой длинной общей *подстрокой* (не подпоследовательностью) двух строк X и Y является самая длинная строка, которая входит в виде серии последовательных символов в обе строки. Например, самой длинной общей подстрокой строк *photograph* и *tomography* является строка *ograph*.

   а) Пусть $n = |X|$ и $m = |Y|$. Предоставьте алгоритм динамического программирования с временной сложностью $\Theta(nm)$ для поиска самой длинной общей подстроки, основанный на алгоритме поиска самой длинной общей подпоследовательности или вычисления расстояния редактирования.

   б) Предоставьте более простой алгоритм с временной сложностью $\Theta(nm)$, в котором не используется динамическое программирование.

4. [6] Самой длинной общей подпоследовательностью (longest common subsequence, LCS) двух последовательностей T и P называется такая самая длинная последовательность L, которая является подпоследовательностью как последовательности T, так и последовательности P. Кратчайшей общей надпоследовательностью (shortest common supersequence, SCS) последовательностей T и P называется такая кратчайшая последовательность L, подпоследовательностями которой являются как последовательность T, так и последовательность P.

   а) Предоставьте эффективный алгоритм поиска LCS и SCS для двух последовательностей.

   б) Пусть $d(T, P)$ будет минимальным расстоянием редактирования между строками T и P при условии, что замены запрещены, т. е. разрешены только вставка и удаление симво-

лов. Докажите, что $d(T, P) = |SCS(T, P)| - |LSC(T, P)|$, где $|SCS(T, P)|$ и $|LSC(T, P)|$ обозначают соответственно размер кратчайшей общей надпоследовательности и самой длинной общей подпоследовательности T и P.

"Жадные" алгоритмы

5. [4] Пусть $P_1, P_2, \ldots P_n$ — это n программ, которые нужно сохранить на диске емкостью D мегабайт. Для хранения программы P_i требуется s_i Мбайт дискового пространства. Сохранить все программы на диске нельзя, т. к. $D < \sum_{i=1}^{n} s_i$.

   а) Можно ли максимизировать количество сохраняемых на диске программ с помощью "жадного" алгоритма, который выбирает программы в порядке неубывающего дискового пространства s_i? Предоставьте доказательство или контрпример.

   б) Можно ли сказать, что "жадный" алгоритм, выбирающий программы в порядке неубывающего дискового пространства s_i, использует наибольшее возможное дисковое пространство? Предоставьте доказательство или контрпример.

6. [5] В Соединенных Штатах используются монеты достоинством в 1, 5, 10, 25 и 50 центов. Теперь рассмотрим страну, в которой используются монеты достоинством в $\{d_1, \ldots, d_k\}$ единиц. Нам нужен алгоритм, который дает сдачу размером в n единиц, используя наименьшее количество монет этой страны.

   а) При решении этой задачи "жадный" алгоритм многократно выбирает монету самого большого достоинства, не превышающую текущий размер сдачи, до тех пор, пока размер сдачи не станет равен нулю. Докажите, что "жадный" алгоритм не всегда выбирает наименьшее количество монет для страны с достоинством монет в {1, 6, 10} единиц.

   б) Предоставьте эффективный алгоритм, который правильно определяет наименьшее количество монет, которое требуется, чтобы дать сдачу стоимостью в n единиц, используя монеты достоинством в $\{d_1, \ldots, d_k\}$ единиц. Выполните анализ времени исполнения данного алгоритма.

7. [5] В Соединенных Штатах используются монеты достоинством в 1, 5, 10, 25 и 50 центов. Теперь рассмотрим страну, в которой используются монеты достоинством в $\{d_1, \ldots, d_k\}$ единиц. Нам нужно определить количество $C(n)$ разных способов дать сдачу стоимостью n единиц. Например, для страны с монетами достоинством в {1, 6, 10} единиц имеем $C(5) = 1$, $C(6) = 2$, ..., $C(9) = 2$, $C(10) = 3$ и $C(12) = 4$.

   а) Сколько существует способов дать сдачу стоимостью в 20 единиц монетами достоинством в {1, 6, 10} единиц?

   б) Предоставьте эффективный алгоритм для вычисления $C(n)$ и выполните анализ его сложности. (Подсказка: подумайте о вычислении $C(n, d)$, т. е. о вычислении количества способов дать сдачу стоимостью n единиц, используя монеты наивысшего достоинства d. Будьте внимательны, чтобы не дать лишнюю сдачу.)

8. [6] Рассмотрим *задачу планирования загрузки однопроцессорной системы* количеством n заданий J. Для каждого задания i установлено время обработки t_i и крайнее время завершения d_i. Допустимым расписанием исполнения заданий является такая перестановка заданий, что при исполнении заданий в данном порядке каждое задание завершается до крайнего времени его завершения. "Жадный" алгоритм для решения задачи *планировки загрузки однопроцессорной системы* выбирает первой задание с самым ранним крайним временем завершения.

Докажите, что если допустимое расписание существует, то создаваемое таким "жадным" алгоритмом расписание является допустимым.

Числовые задачи

9. [6] *Задача о рюкзаке* задается таким образом: имея множество целых чисел $S = \{s_1, s_2, ..., s_n\}$ и целевое число T, найти такое подмножество множества S, сумма которого в точности равна T. Например, множество $S = \{1, 2, 5, 9, 10\}$ содержит подмножество, сумма членов которого составляет $T = 22$, но не $T = 23$. Предоставьте правильный алгоритм для решения задачи о рюкзаке с временем исполнения $O(nT)$.

10. [6] В задаче разделения множества целых чисел требуется выяснить, содержит ли множество целых чисел $S = \{s_1, ..., s_n\}$ такое подмножество I, для которого

$$\sum_{i \in I} s_i = \sum_{i \notin I} s_i$$

Пусть $\sum_{i \in S} s_i = M$. Предоставьте алгоритм динамического программирования с временем исполнения $O(nM)$ для решения задачи разделения множества целых чисел.

11. [5] Допустим, что имеется n чисел (некоторые из которых, возможно, отрицательные), расположенных по кругу. Разработайте эффективный алгоритм поиска наибольшей суммы смежных чисел в дуге.

12. [5] Некий язык для обработки строк позволяет разбивать строку на две части. Стоимость этого разбиения равна n единицам времени, т. к. для этого требуется выполнить копирование старой строки. Программист хочет разбить строку на несколько частей, при этом общее время выполнения разбиения может зависеть от порядка, в котором осуществляется разбиение. Например, допустим, мы хотим разбить 20-символьную строку в позициях после символов 3, 8 и 10. Если разбиения осуществляются в порядке слева направо, то первое разбиение стоит 20 единиц времени, второе — 17, а третье — 12, что дает общую стоимость в 49 единиц времени. Если же разбиения осуществляются в порядке слева направо, то первое разбиение стоит 20 единиц времени, второе — 10, а третье — 8, что дает общую стоимость в 38 единиц времени. Предоставьте алгоритм динамического программирования, который берет в качестве входа список позиций символов и определяет самое дешевое разбиение за время $O(n^3)$.

13. [5] Рассмотрим следующий способ сжатия данных. Имеется таблица, содержащая m текстовых строк, каждая длиной, самое большее, k символов. Требуется закодировать строку данных D длиной n символов, используя для этого наименьшее возможное количество текстовых строк из таблицы. Например, если таблица содержит строки (*a*, *ba*, *abab*, *b*), а строка данных имеет вид *bababbaababa*, то наилучшим способом кодирования будет (*b, abab, ba, abab, a*), в котором используются всего пять строк из таблицы. Предоставьте алгоритм с временем исполнения $O(nmk)$ для определения длины наилучшей кодировки. Можно полагать, что каждую кодируемую строку можно выразить, по крайней мере, одной комбинацией строк в таблице.

14. [5] Традиционный матч мирового чемпионата по шахматам состоит из 24 игр. Если матч заканчивается вничью, то текущий чемпион сохраняет свой титул. Каждая игра может быть выиграна, проиграна или сыграна вничью, где выигрыш равен 1, проигрыш — 0, а ничья — 01/2. Цвет фигур меняется каждую игру. Белые имеют преимущество, т. к. они ходят первыми. В первой игре чемпион играет белыми. Его шансы выиг-

рыша, ничьей и проигрыша равны w_w, w_d и w_l при игре белыми и b_w, b_d и b_l при игре черными соответственно.

а) Напишите рекуррентное соотношение вероятности чемпиона удержания титула. Полагается, что в матче осталось сыграть g игр и что чемпиону нужно выиграть i игр (результат которых может быть 1/2).

б) На основе данного рекуррентного соотношения, разработайте алгоритм динамического программирования для вычисления вероятности текущего чемпиона сохранения своего титула.

в) Выполните анализ временной сложности вашего алгоритма для матча из n игр.

15. [8] Если бросить яйцо с достаточной высоты, оно разобьется. В частности, в любом достаточно высоком здании должен быть f-й этаж, при падении с которого яйцо разобьется, но при падении с $(f-1)$-го этажа — нет. Если яйцо всегда разбивается, то тогда $f = 1$. Если яйцо никогда не разбивается, то тогда $f = n + 1$.

Нужно найти критический этаж f в n-этажном здании. Для этого можно выполнять только одну операцию — бросить яйцо с определенного этажа и наблюдать за результатами. Нужно выполнить как можно меньше таких операций, но в любом случае требуется потратить не больше, чем k яиц. Разбитые яйца снова использовать нельзя. Пусть $E(k, n)$ означает минимальное количество бросков, достаточное для получения решения.

а) Докажите, что $E(1,n) = n$.

б) Докажите, что $E(k, n) = \Theta(n^{1/k})$.

в) Выведите рекуррентное соотношение для $E(k, n)$. Каким будет время исполнения динамической программы для вычисления $E(k, n)$?

Задача на графах

16. [4] Рассмотрим город, улицы в котором задаются решеткой $X \times Y$. Нам нужно пройти из верхнего левого угла решетки в нижний правый угол. К сожалению, в городе имеются районы с плохой репутацией, и мы не хотим проходить по улицам в этих районах. Имеется матрица BAD размером $X \times Y$, в которой $BAD[i, j]$ = "*yes*" тогда и только тогда, когда перекресток улиц i и j находится в районе, через который мы не хотим проходить.

а) Приведите пример такого содержимого матрицы BAD, для которого не существует пути между требуемыми точками без прохождения через плохие районы.

б) Предоставьте алгоритм сложностью $O(XY)$ для поиска пути, позволяющего избежать нежелательных районов.

в) Предоставьте алгоритм сложностью $O(XY)$ для поиска *кратчайшего* пути, позволяющего избежать нежелательных районов. Полагается, что все кварталы одинаковой длины. Чтобы решение было зачтено частично, алгоритм может иметь время исполнения $O(X^2Y^2)$.

17. [5] Даны такие же условия, как и в предыдущей задаче, т. е. город, улицы в котором определены решеткой $X \times Y$. Нам нужно пройти из верхнего левого угла города/решетки к нижнему правому углу. Плохие районы определены матрицей BAD размером $X \times Y$, в которой $BAD[i, j]$ = "yes" тогда и только тогда, если перекресток улиц i и j находится в районе, через который мы не хотим проходить.

Если бы в городе не было неблагополучных районов, которые мы вынуждены избегать, то длина кратчайшего пути между указанными точками была бы равной $(X-1) + (Y-1)$

кварталам. Более того, было бы много таких путей, каждый из которых состоял бы только из движений вправо и вниз.

Разработайте алгоритм, который принимает в качестве входа массив *BAD* и возвращает количество безопасных путей длиной $X + Y - 2$. Чтобы решение было засчитано полностью, время исполнения алгоритма должно быть равным $O(XY)$.

Задачи по разработке

18. [4] В библиотеке нужно разместить на полках n книг в установленном в каталоге порядке. Поэтому мы можем представить позицию конкретной книги как b_i, ее толщину как t_i, а высоту как h_i, где $1 \le i \le n$. Все полки в библиотеке имеют одинаковую длину L.

    Допустим, что все книги имеют одинаковую высоту h (т. е., $h = h_i = h_j$ для всех i, j), а расстояние между полками больше чем h, вследствие чего любую книгу можно поставить на любую полку. "Жадный" алгоритм поставит на первую полку наибольшее возможное количество книг, пока не будет достигнуто такое наименьшее значение i, для которого книга b_i не будет вмещаться на данную полку, после чего повторит эту процедуру с последующими полками. Докажите, что "жадный" алгоритм всегда находит оптимальный порядок размещения полок и укажите его временную сложность.

19. [6] Данная задача является вариантом предыдущей. В данном случае книги разной высоты, но высоту каждой полки можно подогнать под высоту самой высокой книги на ней. Таким образом, стоимость определенного размещения является суммой высот наивысших книг на каждой полке.

    - Предоставьте пример, показывающий, что "жадный" алгоритм, заполняющий каждую полку насколько возможно, не всегда дает минимальную общую высоту.
    - Разработайте алгоритм для решения этой задачи и выполните анализ его временной сложности. *Подсказка*: используйте динамическое программирование.

20. [5] Требуется найти самый легкий способ набрать определенный номер из n цифр на стандартном телефонном кнопочном номеронабирателе, используя только два пальца. Начальными позициями этих двух пальцев являются кнопки звездочки (*) и решетки (#), а стоимость перемещения пальца от одной кнопки к другой пропорциональна эвклидову расстоянию между кнопками. Разработайте алгоритм для набора номера с наименьшим общим расстоянием, пройденным пальцами, с номеронабирателя с k разными кнопками (для стандартных телефонов $k = 16$). Попытайтесь получить время исполнения алгоритма $O(nk^2)$.

21. [6] Имеется вектор из n действительных чисел, для которого нужно найти подвектор с максимальной суммой. Например, в векторе {31, –41, 59, 26, –53, 58, 97, –93, –23, 84} подвектором с максимальной суммой будут третий по седьмой элементы, а именно: 59 + 26 + (–53) + 58 + 97 = 187. Если все элементы массива положительные, то решением является весь массив, а когда все элементы отрицательные, то решением является пустой подвектор с суммой элементов, равной нулю.

    - Разработайте простой алгоритм с временем исполнения $\Theta(n^2)$ для поиска подвектора последовательных элементов с максимальной суммой.
    - Далее разработайте для решения этой же задачи алгоритм динамического программирования с временем исполнения $\Theta(n)$. Чтобы решение было зачтено частично, можно разработать алгоритм типа "разделяй и властвуй" с временем исполнения $O(n\log n)$.

22. **[7] Даны алфавит из k символов, строка $x = x_1x_2...x_n$ из символов этого алфавита и таблица умножения символов алфавита. Нужно определить, возможно ли заключить части строки в скобки таким образом, чтобы в результате получить a, где a является символом алфавита. Таблица умножения не обладает ни перестановочным, ни ассоциативным свойствами, вследствие чего порядок выполнения операций умножения имеет значение.**

	a	b	c
a	a	c	c
b	a	a	b
c	c	c	c

Для примера, рассмотрим предшествующую таблицу умножения и строку *bbbba*. Заключение частей строки в скобки в виде $(b(bb))(ba)$ дает результат a, но в виде $((((bb)b)b)a)$ дает результат c.

Предоставьте алгоритм с полиномиальной временной сложностью по отношению к n и k для определения, можно ли заключить в скобки части данной строки, чтобы получить целевой элемент согласно данной таблице умножения.

23. [6] Даны константы α и β. Допустим, что стоимость движения влево в дереве равна α, а вправо — β. Разработайте алгоритм с оптимальной стоимостью создания дерева в наихудшем случае для ключей $k_1, ..., k_n$, вероятность просмотра каждого из которых составляет $p_1, ..., p_n$.

Задачи, предлагаемые на собеседовании

24. [5] Для определенного набора достоинств монет найти наименьшее количество монет, которым можно дать сдачу определенного размера.

25. [5] Имеется массив из n чисел, каждое из которых может быть положительным, отрицательным или нулем. Предоставьте эффективный алгоритм для определения индексов элементов i и j максимальной суммы от i-го до j-го числа.

26. [7] При вырезании символа из страницы журнала также вырезается символ на обратной стороне страницы. Предоставьте алгоритм для определения, возможно ли создать определенную строку из вырезанных из определенного журнала символов. Можно полагать, что имеется функция, которая позволяет определить символ на обратной стороне страницы и его позицию для любого данного символа на лицевой стороне страницы.

Задачи по программированию

Эти задачи доступны на сайтах **http://www.programming-challenges.com** и **http://uva.onlinejudge.org**.

1. Is Bigger Smarter? 111101/10131.
2. Weights and Measures. 111103/10154.
3. Unidirectional TSP. 111104/10116.
4. Cutting Sticks. 111105/10003.
5. Ferry Loading. 111106/10261.

Труднорешаемые задачи и аппроксимирующие алгоритмы

В этой главе обсуждаются методы доказательства того, что для решения данной задачи не существует эффективного алгоритма. Читателям, не склонным к теоретизированию, необходимость доказывать что-либо, будет, скорее всего, неприятна, а сама мысль тратить время на доказательство чего-то несуществующего покажется абсолютно неприемлемой. Однако, если мы не знаем, как решить какую-то задачу, бывает небесполезно выяснить, что она не имеет решения.

Теория NP-полноты является чрезвычайно полезным инструментом разработчика алгоритмов, даже когда дает отрицательные результаты. В частности, теория NP-полноты позволяет разработчику алгоритмов тратить усилия более эффективно, показав, что поиск эффективного алгоритма для данной задачи обречен на неудачу. Зато, если вы потерпели неудачу, доказывая нерешаемость задачи, вы можете ожидать, что существует эффективный алгоритм для ее решения. Две истории из жизни, приведенные в этой книге, описывают случаи, закончившиеся благополучно, несмотря на неоправданные жалобы разработчиков на сложность задачи.

Теория NP-полноты также позволяет распознать свойства, которые делают определенную задачу сложной. Это дает нам представление о том, как следует моделировать ее разными способами или использовать ее более благоприятные характеристики. Понимание того, какие задачи являются сложными, исключительно важно для разработчиков алгоритмов, и оно может быть приобретено только в попытках доказать сложность задач.

Для доказательства сложности задач мы будем использовать такой фундаментальный принцип, как *сведение* между парой задач, показывая, что эти задачи в действительности одинаковые. Для демонстрации этой идеи мы рассмотрим последовательность сведений, каждое из которых дает эффективный алгоритм или доказательство, что такого алгоритма не существует. Кроме этого, предоставляется краткое введение в:

- теоретико-сложностные аспекты NP-полноты, одного из наиболее фундаментальных понятий теории вычислительных систем;
- теорию аппроксимирующих алгоритмов, позволяющую получить эвристический алгоритм, который, возможно, выдаст решение, *близкое* к оптимальному.

9.1. Сведение задач

В этой книге нам встретилось несколько задач, для решения которых мы не могли найти никаких эффективных алгоритмов. Теория NP-полноты предоставляет инструменты, с помощью которых можно показать, что на определенном уровне все эти задачи в действительности являются одной и той же задачей.

Ключевым принципом для демонстрации сложности задачи является *сведение*, или преобразование одной задачи в другую. С объяснением этой идеи может помочь такая аллегория NP-полноты. Несколько парней по очереди дерутся друг с другом, чтобы выяснить, кто из них "круче". Адам победил Билла, который затем одолел Чака. Кто же из этих троих "крутой", и есть ли смысл вообще говорить об этом? Без какого-либо внешнего стандарта ответить невозможно. Если бы Чак оказался Чаком Норрисом, в "крутости" которого никто не сомневается, то Адам и Билл были бы не менее "крутыми", чем он. С другой стороны, допустим, что отношения выясняют мальчишки из начальных классов. Хотя никому не придет в голову называть меня "крутым", даже если я мог бы справиться с Адамом. Отсюда следует, что ни одного из этих троих нельзя считать "крутым". В этой аллегории NP-полноты каждая драка представляет сведение, а Чак Норрис играет роль задачи выполнимости (satisfiability), т. е. достоверно трудно решаемой задачи. Я же выступаю в роли неэффективного алгоритма, с возможностью исправления.

Сведение — это операция преобразования одной задачи в другую. Чтобы описать сведение, нам нужно использовать довольно строгие определения. Алгоритмическая *задача* представляет собой вопрос общего характера, для которого даются входные параметры и условия, формулирующие, что можно считать удовлетворительным ответом или решением. *Экземпляром* (instance) называется задача, для которой указаны параметры входа. Разницу между задачей и ее экземпляром можно продемонстрировать на следующем примере:

Задача. Задача коммивояжера.

Вход. Взвешенный граф G.

Выход. Маршрут, минимизирующий $\sum_{i=1}^{n-1} d[v_i, v_{i+1}] + d[v_n, v_1]$.

Любой взвешенный граф определяет экземпляр задачи коммивояжера, а каждая конкретная задача имеет, по крайней мере, один маршрут с минимальной стоимостью. Для общего экземпляра задачи коммивояжера требуется найти алгоритм для определения оптимального маршрута для всех возможных экземпляров задачи.

9.1.1. Ключевая идея

Теперь рассмотрим две алгоритмические задачи, называющиеся Bandersnatch и Bobilly. Допустим, что у нас имеется алгоритм решения задачи Bandersnatch (листинг 9.1).

Листинг 9.1. Решение задачи Bandersnatch

```
Bandersnatch(G)
  Преобразуем G в экземпляр Y задачи Bo-billy
  Вызываем процедуру Bo-billy для решения экземпляра Y
  Возвращаем результат Bo-billy(Y) в качестве ответа для Bandersnatch(G)
```

Этот алгоритм выдаст *правильное* решение задачи Bandersnatch при условии, что при ее преобразовании в задачу Bo-billy всегда сохраняется правильность ответа. Иными словами, для любого экземпляра G преобразование имеет свойство:

$$\text{Bandersnatch}(G) = \text{Bo-billy}(Y)$$

Преобразование экземпляров одного типа задачи в экземпляры задачи другого типа с сохранением правильных ответов и называется *сведением* (reduction).

Теперь допустим, что при таком сведении экземпляр G преобразуется в Y за время $O(P(n))$. Тогда возможны два варианта:

- *если* процедура Bo-billy выполняется за время $O(P'(n))$, то это означает, что время решения задачи Bandersnatch равно $O(P(n) + P'(n))$, т. е. сумме времени преобразования задачи Bandersnatch в задачу Bo-billy и времени решения последней;
- *если* известно, что выражение $\Omega(P'(n))$ является нижним пределом при вычислении Bandersnatch, т. е., что данную задачу нельзя решить быстрее, тогда выражение $\Omega(P'(n) - P(n))$ должно быть нижним пределом для вычисления Bo-billy. Почему это так? Если можно было бы решить Bo-billy быстрее, то это могло бы нарушить нижний предел при решении Bandersnatch посредством ранее описанного приведения. Отсюда неявно следует, что не может существовать способа решения Bo-billy быстрее, чем заявлено.

В первом случае Стив (т. е. автор этих строк) демонстрирует ловким ударом в челюсть Адама, что все остальные являются слабаками. А во втором описывается подход с участием Чака Норриса, используемый для доказательства сложности задачи. По существу, данное сведение показывает, что задача Bo-billy не легче, чем задача Bandersnatch. Поэтому если задача Bandersnatch сложная, то это означает, что задача Bo-billy также должна быть сложной.

Для иллюстрации мы рассмотрим в этой главе несколько примеров сведения задач.

Подведение итогов

Посредством сведения можно показать, что две задачи являются, по сути, одинаковыми. Наличие эффективного алгоритма для решения одной задачи (или отсутствие такого) подразумевает наличие (или отсутствие) эффективного алгоритма для решения другой.

9.1.2. Задачи разрешимости

При сведении задача одного типа преобразуется в задачу другого типа таким образом, что ответы для всех экземпляров задачи являются идентичными. Задачи отличаются друг от друга *диапазоном* или *типом* возможных ответов. Решение задачи коммивояжера состоит из перестановки вершин, в то время как решения других задач возвращаются в виде чисел, диапазон которых может быть ограничен положительными или целыми числами.

Диапазон решений наиболее интересного класса задач ограничен значениями ИСТИНА или ЛОЖЬ. Эти задачи называются *задачами разрешимости* (decision problems). Удобно сводить одну задачу разрешимости к другой, т. к. допустимыми ответами как для начальной, так и для конечной задачи являются только ИСТИНА или ЛОЖЬ.

К счастью, большинство представляющих интерес задач оптимизации можно сформулировать в виде задачи разрешимости, которая отражает суть вычислений. Например, задачу разрешимости для задачи коммивояжера можно определить таким образом:

Задача. Разрешимость задачи коммивояжера.

Вход. Взвешенный граф G и целое число k.

Выход. Ответ, существует ли маршрут стоимостью $\leq k$.

Формулировка в виде задачи разрешимости отражает саму суть задачи коммивояжера в том смысле, что если существует эффективный алгоритм для решения задачи разрешимости, то с его помощью можно выполнить двоичный поиск для разных значений k и быстро оптимизировать решение. А приложив некоторые усилия, можно по эффективному решению задачи разрешимости восстановить и саму перестановку узлов маршрута.

В дальнейшем мы будем, как правило, говорить о задачах разрешимости, потому что такой подход, во-первых, проще, а во-вторых, обладает достаточной теоретической мощью.

9.2. Сведение для создания новых алгоритмов

На кухне сидят инженер и алгорист. Алгорист просит инженера вскипятить воды для чая. Инженер встает со своего стула, берет со стола чайник, наливает в него воды, ставит его на плиту, зажигает горелку, ждет, пока вода закипит, после чего выключает горелку. Некоторое время спустя инженер просит алгориста вскипятить еще воды. Тот поднимается со стула, берет чайник с плиты, ставит на стол и снова садится. — Сделано, — говорит он. — Я *свел* задачу, требующую решения, к уже решенной задаче.

Пример со сведением задачи кипячения воды иллюстрирует достойный способ создания новых алгоритмов из старых. Если входные данные для задачи, *которую мы хотим решить*, можно преобразовать во входные данные для задачи, *которую мы знаем, как решить*, то процедура преобразования и известное решение образуют алгоритм для решения нашей задачи.

В этом разделе мы рассмотрим несколько примеров сведения, которые позволяют воспользоваться эффективными алгоритмами. Чтобы решить задачу a, мы выполняем сведение экземпляра задачи a к экземпляру задачи b, после чего решаем данный экземпляр, используя эффективный алгоритм для решения задачи b. Общее время исполнения в данном случае равно времени, необходимому для выполнения сведения, плюс время для решения экземпляра задачи b.

9.2.1. Поиск ближайшей пары

В задаче *поиска ближайшей пары* требуется найти в множестве чисел пару чисел с наименьшей разницей между ними. Эту задачу можно преобразовать в задачу разрешимости, задав вопрос, является ли данная разница меньшей, чем некое пороговое значение. А именно:

Вход. Множество S из n чисел и пороговое значение t.

Выход. Решение, существует ли такая пара чисел s_i, $s_j \in S$, для которой $|s_i - s_j| \leq t$.

Задача поиска ближайшей пары решается применением простой сортировки, т. к. после сортировки члены ближайшей пары должны находиться рядом друг с другом. Отсюда следует этот алгоритм:

CloseEnoughPair(S,t)
    Sort S.
    Is $\min_{1 \le i < n} |s_i - s_{i+1}| \le t$?

По поводу этого простого сведения можно сделать несколько замечаний.

1. Сведенная к задаче разрешимости версия задачи сохраняет суть первоначальной задачи, что означает, что решить ее не легче, чем первоначальную задачу поиска ближайшей пары.
2. Временная сложность алгоритма решения зависит от временной сложности сортировки. Например, если применить для сортировки алгоритм с временной сложностью $O(n\log n)$, то время поиска ближайшей пары будет равно $O(n\log n + n)$.
3. Это сведение и факт существования нижнего предела времени исполнения сортировки, равного $\Omega(n\log n)$, *не* доказывают, что в наихудшем случае достаточно близкую пару можно найти за время $\Omega(n\log n)$. Возможно, существует более быстрый алгоритм, который только нужно поискать?
4. С другой стороны, если бы мы знали, что в наихудшем случае достаточно близкую пару можно найти за время $\Omega(n\log n)$, то это сведение было бы достаточным доказательством того, что сортировку нельзя осуществить быстрее, чем за время $\Omega(n\log n)$, поскольку такая более быстрая сортировка подразумевала бы более быстрый алгоритм поиска достаточно близкой пары.

9.2.2. Максимальная возрастающая подпоследовательность

В *главе 8* было продемонстрировано использование динамического программирования для решения разных задач, включая вычисление расстояния редактирования строки (см. *раздел 8.2*) и максимальной возрастающей подпоследовательности (см. *раздел 8.3*). Приведу краткий обзор этих задач.

ЗАДАЧА. Вычислить расстояние редактирования.

Вход. Последовательности целых чисел или символов S и T; стоимость каждой операции вставки обозначена как c_{ins}, удаления — c_{del} и замены — c_{sub}.

Выход. Последовательность операций с минимальной стоимостью для преобразования последовательности S в последовательность T.

ЗАДАЧА. Найти максимальную возрастающую подпоследовательность.

Вход. Последовательность S целых чисел или текстовых символов.

Выход. Такая максимальная последовательность позиций целых чисел $\{p_1, \ldots, p_m\}$, для которой $p_i < p_{i+1}$ и $S_{p_i} < S_{p_{i+1}}$.

В действительности, задачу поиска максимальной возрастающей подпоследовательности можно решить в виде частного случая задачи вычисления расстояния редактирования, как показано в листинге 9.2.

Листинг 9.2. Поиск максимальной возрастающей последовательности

```
Longest Increasing Subsequence(S)
    T = Sort(S)
    c_ins = c_del = 1
    c_sub = ∞
    Return (|S| - EditDistance(S,T,c_ins,c_del,c_del)/2)
```

Почему этот алгоритм дает правильный результат? Создавая вторую последовательность T из элементов последовательности S, отсортированных в возрастающем порядке, мы обеспечиваем, что любая общая подпоследовательность должна быть возрастающей. Если же замены не разрешены ни при каких обстоятельствах (т. к. $c_{sub} = \infty$), то в результате оптимального выравнивания двух последовательностей мы находим их максимальную общую подпоследовательность и удаляем все прочее. Таким образом, для преобразования последовательности {3, 1, 2} в последовательность {1, 2, 3} требуются две операции, а именно удаление и вставка цифры 3, не попавшей в максимальную общую подпоследовательность. Длиной максимальной возрастающей подпоследовательности будет длина последовательности S за вычетом половины стоимости этого преобразования.

Каковы последствия данного сведения? Исполнение самого сведения занимает время $O(n\log n)$. Поскольку вычисление расстояния редактирования занимает время $O(|S|\cdot|T|)$, то временная сложность алгоритма поиска максимальной возрастающей подпоследовательности в S будет квадратичной, т. е. такой же, как и сложность алгоритма, рассмотренного в *разделе 8.3*. В действительности, для поиска максимальной возрастающей подпоследовательности существует более быстрый алгоритм с временной сложностью $O(n\log n)$, основанный на удачно подобранных структурах данных, в то время как алгоритм вычисления расстояния редактирования в худшем случае имеет квадратичную временную сложность. В данном случае в результате сведения задач мы получим простой, но не оптимальный алгоритм с полиномиальной временной сложностью.

9.2.3. Наименьшее общее кратное

При работе с целыми числами часто требуется найти *наименьшее общее кратное* и *наибольший общий делитель*. Говорят, что a кратно b (т. е. $b|a$), если существует такое целое число d, для которого $a = bd$. Тогда постановка этих двух задач выглядит таким образом:

ЗАДАЧА. Найти наименьшее общее кратное (НОК).

Вход. Два целых числа x и y.

Выход. Наименьшее целое число m, которое кратно и числу x, и числу y.

ЗАДАЧА. Найти наибольший общий делитель (НОД).

Вход. Два целых числа x и y.

Выход. Наибольшее целое число d, на которое делится как число x, так и число y.

Например, НОК(24,36) = 72, а НОД(24,36) = 12. Обе задачи можно с легкостью решить, разложив числа x и y на простые множители, но до сих пор не предложен эффективный алгоритм для разложения целых чисел на множители (см. *раздел 13.8*).

К счастью, задачу поиска наибольшего общего делителя можно эффективно решить с помощью алгоритма Евклида, не прибегая при этом к разложению на множители. Этот рекурсивный алгоритм основан на двух наблюдениях. Первое:

если $b|a$, тогда НОД(a, b) = b.

Это должно быть довольно очевидным, т. к. если a кратно b, тогда $a = bk$ для некоторого целого числа k, откуда следует, что НОД(bk, b) = b. Второе:

если $a = bt + r$ для целых чисел t и r, тогда НОД(a, b) = НОД(b, r).

Поскольку $x \cdot y$ является кратным как x, так и y, то НОК(x, y) $\leq xy$. Существование меньшего общего кратного возможно лишь в том случае, если существует какое-либо нетривиальное общее кратное для x и y. Данное свойство в совокупности с алгоритмом Евклида обеспечивает эффективный способ для вычисления наименьшего общего кратного, как показано в следующем псевдокоде:

```
LeastCommonMultiple(x, y)
   Return (xy/НОД(x,y)).
```

Это сведение позволяет нам применить алгоритм Евклида для решения другой задачи.

9.2.4. Выпуклая оболочка (*)

В нашем последнем примере сведения "легкой" задачи (т. е. задачи, которая решается за полиномиальное время) мы используем сортировку для поиска выпуклой оболочки. Многоугольник называется *выпуклым*, если отрезок прямой линии, проведенной между любыми двумя точками внутри многоугольника P, находится полностью внутри многоугольника. Это возможно, когда многоугольник P не содержит зазубрин или вогнутостей, поэтому выпуклые многоугольники имеют аккуратную форму. Выпуклая оболочка предоставляет удобный способ структурирования множества точек. Соответствующие приложения рассматриваются в *разделе 17.2*.

ЗАДАЧА. Найти выпуклую оболочку.

Вход. Множество S, состоящее из n точек, лежащих в плоскости.

Выход. Наименьший выпуклый многоугольник, содержащий все точки множества S.

Данная задача решается с помощью сортировки (рис. 9.1). Это означает, что каждое число преобразуется в точку, для чего значение x отображается на точку (x, x^2), т. е. каждое целое число отображается на точку параболы $y = x^2$. А так как эта парабола является выпуклой, то каждая точка должна находиться на выпуклой оболочке. Кроме этого, так как соседние точки на выпуклой оболочке имеют соседние значения x, то выпуклая оболочка возвращает точки, отсортированные по x-координате, т. е. первоначальные числа. Временная сложность алгоритма для создания и считывания точек, приведенного в листинге 9.3, равна $O(n)$.

Листинг 9.3. Алгоритм создания и считывания точек выпуклой оболочки

```
Sort(S)
Для каждого числа i∈S создаем точку (i, i²).
   Вызываем процедуру выпуклой оболочки для этого набора точек.
   Начиная с самой левой точки оболочки, считываем точки слева направо.
```

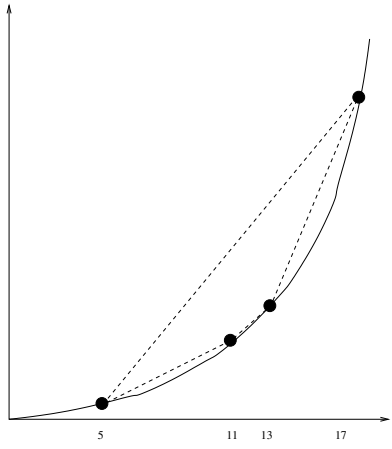


Рис. 9.1. Сведение задачи поиска выпуклой оболочки к задаче сортировки посредством отображения точек на параболе

Что все это означает? Вспомним, что нижний асимптотический предел сортировки равен $\Omega(n\lg n)$. Если бы временная сложность вычисления выпуклой оболочки могла быть лучшей, чем $n\lg n$, то это сведение подразумевает временную сложность сортировки лучшую, чем $\Omega(n\lg n)$, что нарушает наш нижний предел. Соответственно, временная сложность вычисления выпуклой оболочки также должна быть равной $\Omega(n\lg n)$! Обратите внимание, что любой алгоритм для вычисления выпуклой оболочки, имеющий временную сложность $O(n\lg n)$, также предоставляет нам сложный, но правильный алгоритм сортировки с временем исполнения $O(n\lg n)$.

9.3. Простые примеры сведения сложных задач

Сведения в предыдущем разделе демонстрируют преобразования между задачами, для решения которых существуют эффективные алгоритмы. Но нас, в основном, интересует использование механизма сведения для доказательства сложности задачи путем демонстрации того, что существование быстрого алгоритма решения задачи Bandersnatch влечет за собой существование алгоритма, невозможного для задачи Bo-billy.

На данном этапе вам нужно просто поверить мне на слово, что задачи поиска гамильтонова цикла и вершинного покрытия являются сложными. Вся картина сведения (рис. 9.2) станет ясной по завершении изучения этой главы.

9.3.1. Гамильтонов цикл

Задача поиска гамильтонова цикла является одной из наиболее знаменитых задач в теории графов. В ней требуется найти маршрут, который проходит через каждую вершину данного графа ровно один раз. Данная задача имеет долгую историю и множество применений, как рассматривается в *разделе 16.5*. Формальное определение задачи следующее:

ЗАДАЧА. Найти гамильтонов цикл.

Вход. Невзвешенный граф G.

Выход. Существует ли простой маршрут, который проходит через каждую вершину графа G ровно один раз?

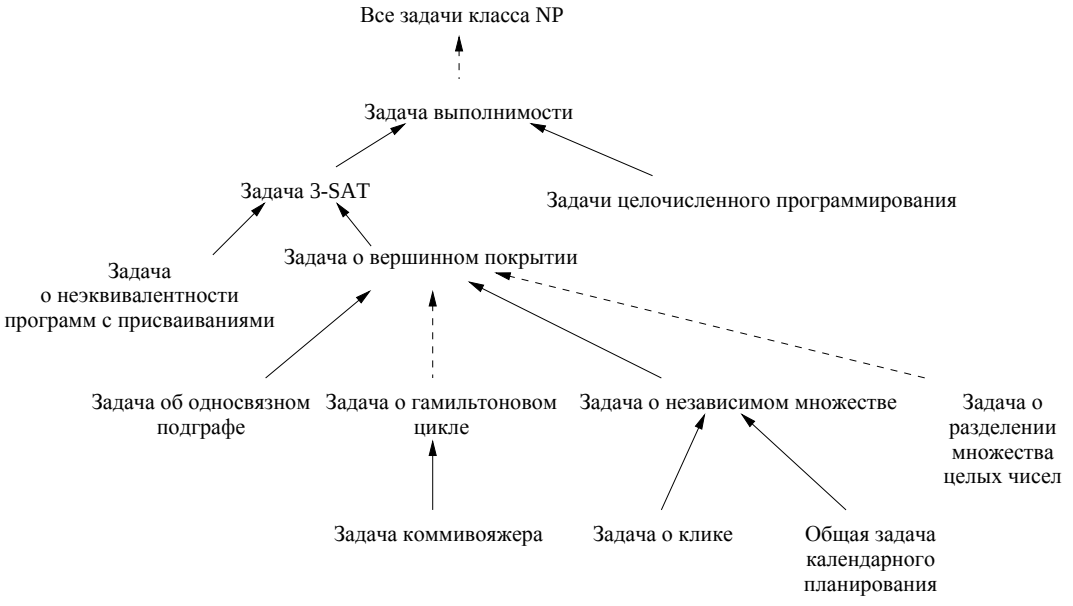


Рис. 9.2. Часть дерева сводимости для NP-полных задач. Сплошные линии обозначают сводимости, рассматриваемые в этой главе

Задача поиска гамильтонова цикла имеет очевидное сходство с задачей коммивояжера. В каждой задаче требуется найти маршрут, который проходит через каждую вершину ровно один раз. Но между этими двумя задачами также имеются различия. Задача коммивояжера решается для взвешенных графов, а гамильтонова задача — для невзвешенных. Из показанного в листинге 9.4 сведения задачи гамильтонова цикла к задаче коммивояжера можно видеть, что сходств между этими задачами больше, чем различий.

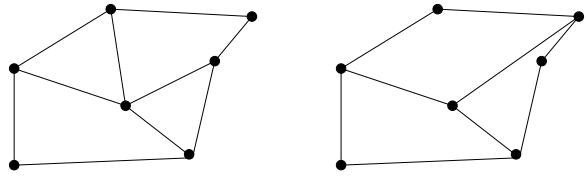
Листинг 9.4. Алгоритм сведения задачи поиска гамильтонова цикла к задаче коммивояжера

```
HamiltonianCycle(G = (V,E))
  Создаем полный взвешенный граф G' = (V', E'), где V' = V.
  n=|V|
  for i = 1 to n do
    for j = 1 to n do
      if (i, j) ∈ E then w(i, j) = 1 else w(i, j) = 2
  Решаем задачу Traveling-Salesman-Decision-Problem(G', n)
```

Само сведение не несет в себе ничего сложного, причем преобразование из невзвешенного во взвешенный граф осуществляется за время $O(n^2)$. Кроме этого, данное преобра-

зование подобрано таким образом, чтобы была обеспечена идентичность ответов для обеих задач.

Если граф G содержит гамильтонов цикл $\{v_1, ..., v_n\}$, тогда этот же маршрут будет соответствовать n ребрам в наборе ребер E', вес каждого из которых равен единице. Это дает нам маршрут для задачи коммивояжера в графе G' весом ровно в n единиц. Если граф G не содержит гамильтонов цикл, тогда граф G' не может содержать такого маршрута коммивояжера, т. к. единственный способ получения в графе G маршрута стоимостью в n единиц требовал бы использования ребер, каждое весом в единицу, что подразумевает наличие гамильтонова цикла в графе G. На рис. 9.3, *а* представлен пример графа, содержащего гамильтонов цикл, а на рис. 9.3, *б* — графа, не содержащего гамильтонов цикл.



а) *б)*

Рис. 9.3. Примеры графов

Такое сведение является эффективным и сохраняет истинность. Наличие эффективного алгоритма для решения задачи коммивояжера подразумевало бы наличие эффективного алгоритма для решения задачи поиска гамильтонова цикла, в то время как доказательство сложности задачи поиска гамильтонова цикла также подразумевало бы сложность задачи коммивояжера. Поскольку в данном случае имеет место второй вариант, это сведение показывает, что задача коммивояжера является сложной, по меньшей мере, в той же степени, что и задача поиска гамильтонова цикла.

9.3.2. Независимое множество и вершинное покрытие

Задача о вершинном покрытии, которая рассматривается более подробно в *разделе 16.3*, заключается в поиске минимального набора вершин, в который входит хотя бы один конец каждого ребра графа. Формальное определение задачи следующее:

ЗАДАЧА. Найти вершинное покрытие.

Вход. Граф $G = (V, E)$ и целое число $k \leq |V|$.

Выход. Существует ли такое подмножество S, содержащее, самое большее, k вершин, для которого каждое ребро $e \in E$ содержит, по крайней мере, одну вершину в множестве S?

Поиск "обычного" вершинного покрытия графа, т. е. покрытия, содержащего все вершины, представляет собой тривиальную задачу. Более сложной является задача охвата всех ребер с использованием минимально возможного набора вершин. Для графа на рис. 9.4 вершинное покрытие образовано четырьмя из восьми вершин графа.

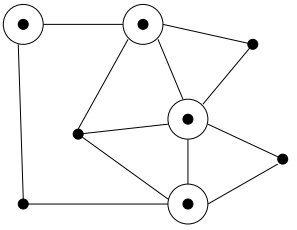


Рис. 9.4. Обведенные кружками вершины составляют вершинное покрытие, а остальные вершины — независимое множество

Множество вершин S графа G является *независимым*, если не существует ребер (x, y), для которых как вершина x, так и вершина y являются элементами множества вершин S. Это означает, что ни одна из любой пары вершин независимого множества не соединена ребром. Как будет рассказано в *разделе 16.2*, независимое множество возникает при решении задач размещения точек обслуживания.

Формальная постановка задачи выглядит таким образом:

ЗАДАЧА. Найти независимое множество.

Вход. Граф G и целое число $k \leq |V|$.

Выход. Существует ли в графе G независимое множество из k вершин?

Как в задаче о вершинном покрытии, так и в задаче о независимом множестве нужно найти особое подмножество вершин, содержащее хотя бы один конец каждого ребра в первом случае и не содержащее концы ребер во втором. Если множество S является вершинным покрытием графа G, то остальные вершины $S–V$ должны составлять независимое множество, т. к. если бы обе вершины ребра находились в подмножестве $S–V$, то тогда множество S не могло бы быть вершинным покрытием. Это позволяет нам выполнить сведение между этими двумя задачами (листинг 9.5).

Листинг 9.5. Алгоритм для сведения задачи о независимом множестве и задачи о вершинном покрытии

```
VertexCover(G, k)
    G' = G
    k'= |V| - k
    Решаем задачу IndependentSet(G', k ')
```

Опять же, простое сведение показывает, что обе эти задачи являются идентичными. Обратите внимание на то, что это преобразование осуществляется без какой бы то ни было информации об ответе. Преобразованию подвергается *вход*, а не решение. Это сведение демонстрирует, что из сложности задачи о вершинном покрытии вытекает сложность задачи о независимом множестве. В данном конкретном случае преобразуемые задачи можно с легкостью поменять местами, и это доказывает, что они одинаково сложны.

Остановка для размышлений. Сложность общей задачи календарного планирования

Докажите, что *общая* задача календарного планирования является NP-полной, сведя к этой задаче задачу о независимом множестве.

ЗАДАЧА. Общая задача календарного планирования.

Вход. Набор I, состоящий из n наборов линейных интервалов, целое число k.

Выход. Можно ли из множества I выбрать подмножество из, по крайней мере, k непересекающихся наборов линейных интервалов?

Решение. Вспомните задачу календарного планирования съемок в фильмах (см. *раздел 1.2*). Там каждому фильму соответствовал один временной интервал, во время которого осуществлялись съемки фильма. Требовалось найти наибольший возможный набор неконфликтующих фильмов, т. е. фильмов, периоды съемок которых не пересекаются.

Общий вариант календарного планирования позволяет разбить выполнение одного задания на несколько временных интервалов. Например, задание A, с периодами исполнения январь–март и май–июнь, не конфликтует с заданием B, с периодами исполнения апрель–август, но конфликтует с заданием C, с периодом исполнения июнь–июль.

Если мы хотим доказать сложность задачи календарного планирования на основе задачи о независимом множестве, то какая задача должна играть роль Bandersnatch, а какая роль Bo-billy? Нам необходимо показать, как преобразовать все задачи о независимом множестве в экземпляры задачи календарного планирования, т. е. наборов непересекающихся линейных интервалов.

Какое соответствие имеется между этими двумя задачами? В обоих случаях требуется выбрать наибольшее возможное подмножество — вершин в задаче о независимом множестве и проектов в задаче календарного планирования. Это обстоятельство наводит на мысль, что нужно преобразовать вершины в проекты. Более того, в обеих задачах требуется, чтобы выбранные элементы не конфликтовали, т. е. вершины не имели общее ребро, а временные интервалы проектов не пересекались. Соответствующее сведение показано в листинге 9.6.

Листинг 9.6. Алгоритм для сведения задачи о независимом множестве к задаче о календарном планировании

```
IndependentSet(G, k)
   I = 0
   Для i-го ребра (x, y), 1 ≤ i ≤ m
      Добавляем интервал [i, i + 0,5] для проекта x до I
      Добавляем интервал [i, i + 0,5] для проекта y до I
      Решаем задачу GeneralProjectScheduling(I, k)
```

Мое решение строится таким образом. Для каждого из m ребер графа на линии создается интервал. Соответствующий каждой вершине проект будет содержать интервалы для смежных с ним ребер, как показано на рис. 9.5.

Каждая пара вершин, имеющая общее ребро (что не допускается в независимом множестве), определяет пару проектов, имеющих общий временной интервал (что не допускается в расписании съемок актера). Таким образом, наибольшие подмножества, удовлетворяющие условиям обеих задач, являются одинаковыми, а эффективный алгоритм для решения общей задачи календарного планирования дает нам быстрый алго-

ритм для решения задачи о независимом множестве. Соответственно, общая задача календарного планирования должна быть такой же сложной, как и задача о независимом множестве. ■

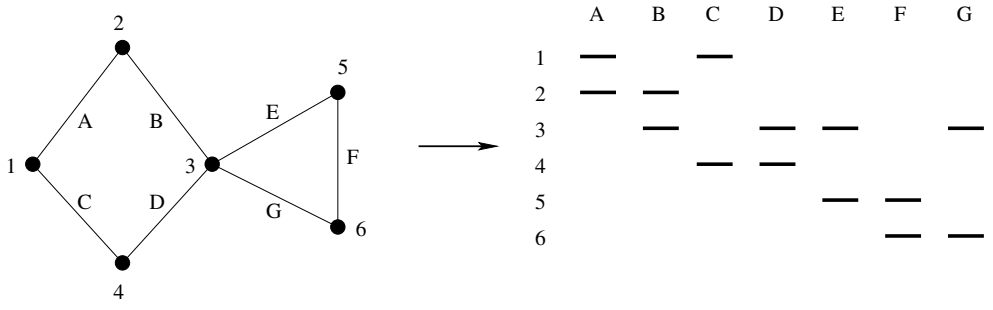


Рис. 9.5. Сведение задачи о независимом множестве к задаче календарного планирования (вершины обозначены цифрами, а ребра — буквами)

9.3.3. Задача о клике

Социальной кликой (clique) является группа друзей, которые вместе проводят время. В теории графов *кликой* называется полный подграф, в котором каждая пара вершин соединена ребром. Клики являются наиболее плотными подграфами. Формальная постановка задачи о клике имеет такой вид:

ЗАДАЧА. Найти максимальную клику.

Вход. Граф $G = (V, E)$ и целое число $k \leq |V|$.

Выход. Содержит ли граф клику из k вершин, т. е., существует ли такое подмножество $S \subset V$, где $|S| \leq k$, для которого каждая пара вершин в S определяет ребро в G?

На рис. 9.6 показан граф, содержащий клику из пяти вершин.

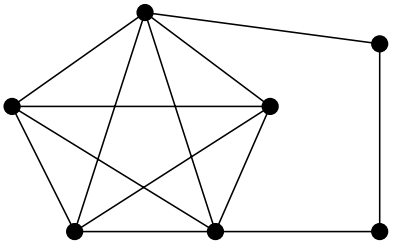


Рис. 9.6. Пример графа

Можно ожидать, что граф дружеских отношений будет содержать большие клики, соответствующие отношениям на работе, между соседями, в церковных приходах, учебных заведениях и т. п. Применение клик рассматривается более подробно в *разделе 16.1*.

В случае с задачей о независимом множестве нам нужно было найти такое подмножество вершин S, не являющихся концами общих ребер. Задача о клике отличается от этой задачи тем, что любая пара вершин должна быть соединена ребром. Для сведения

этих задач мы выполняем операцию *дополнения* графа, т. е. производим обмен ролями между фактом наличия ребра и фактом его отсутствия (листинг 9.7).

Листинг 9.7. Сведение задачи о независимом множестве к задаче о клике

```
IndependentSet(G, k)
      Создаем граф G' = (V', E'), где V = V, и
         For all (i, j) not in E, add (i, j) to E'
      Решаем задачу Clique(G', k)
```

Последние два сведения предоставляют цепочку, связывающую три разные задачи. Сложность задачи о клике следует из сложности задачи о независимом множестве, которая, в свою очередь, следует из сложности задачи о вершинном покрытии. Выстраивая сведения в цепочку, мы связываем вместе пары задач в соответствии с импликацией сложности. Нашу работу можно считать выполненной, если все эти цепочки начинаются с заведомо сложной задачи. В данной цепочке первым звеном является задача выполнимости.

9.4. Задача выполнимости булевых формул

Для демонстрации сложности задач с помощью сведений нам нужно начать с одной заведомо сложной задачи. Самой сложной изо всех NP-полных задач является логическая задача, называющаяся задачей *выполнимости* (satisfiability) булевых формул.

Далее приводится формальная постановка этой задачи.

ЗАДАЧА. Задача выполнимости булевых формул.

Вход. Набор булевых переменных V и набор дизъюнкций (clause) C над V.

Выход. Существует ли выполнимый набор значений истинности для дизъюнкций из C, т. е. способ присвоить набору переменных v_1, ..., v_n значение ИСТИНА или ЛОЖЬ таким образом, чтобы каждая дизъюнкция содержала, по крайней мере, один истинный литерал?

Для ясности приведем два примера. Допустим, что имеется набор $C = \{\{v_1, \bar{v}_2\},\{\bar{v}_1, v_2\}\}$ над булевыми переменными $V = \{v_1, v_2\}$. Символом $\bar{v}_i$ обозначается дополнение переменной v_i. Это позволяет нам обеспечить выполнимость дизъюнкции, содержащей v_i, если v_i = ИСТИНА, или дизъюнкции, содержащей $\bar{v}_i$, если v_i = ЛОЖЬ. Следовательно, выполнимость некоторого набора дизъюнкций включает в себя принятие последовательности из n решений ИСТИНА или ЛОЖЬ в попытке найти такие булевы значения, при которых выполняются все дизъюнкции.

Дизъюнкции $C = \{\{v_1, \bar{v}_2\},\{\bar{v}_1, v_2\}\}$ можно выполнить, присвоив значения $v_1 = v_2 =$ ИСТИНА или $v_1 = v_2 =$ ЛОЖЬ. Рассмотрим теперь набор дизъюнкций $C = \{\{v_1, v_2\}, \{v_1, \bar{v}_2\},\{\bar{v}_1\}\}$. В данном случае не существует удовлетворительного набора значений, т. к. для выполнения третьей дизъюнкции значение переменной v_1 должно быть ЛОЖЬ, откуда следует, что для выполнения второй дизъюнкции значение переменной v_2 должно быть ЛОЖЬ, но тогда не выполняется первое выражение. И сколько бы мы ни старались, получить требуемую выполнимость нам не удастся.

По ряду общественных и технических причин существует устоявшееся мнение, что задача выполнимости булевых формул является сложной, для решения которой не существует алгоритмов с полиномиальным временем исполнения в худшем случае. Буквально все эксперты в области разработки алгоритмов (и бесчисленное множество просто хороших профессионалов) пытались непосредственно или косвенно разработать эффективный алгоритм для выяснения выполнимости данного набора дизъюнкций. Все они потерпели неудачу. Кроме этого, было доказано, что если бы существовал эффективный алгоритм для решения задачи выполнимости, то в области вычислительной сложности были бы возможными многие странные и невероятные вещи. Задача выполнимости является сложной, и принятие этого факта не должно доставлять нам никаких неудобств. Дополнительную информацию по задаче выполнимости и ее применении см. в *разделе 14.10*.

9.4.1. Задача выполнимости в 3-конъюнктивной нормальной форме

Занимаемое задачей выполнимости первое место среди NP-полных задач обусловило тот факт, что ее трудно решить для наихудшего случая. Но некоторые экземпляры частных случаев задачи не обязательно такие сложные. Допустим, что каждая дизъюнкция содержит ровно один литерал. Чтобы выполнить данную дизъюнкцию, этому литералу необходимо присвоить правильное значение. Эту процедуру можно повторить для каждой дизъюнкции данного экземпляра задачи. Таким образом, этот набор будет невыполнимым только в том случае, если имеется две дизъюнкции, которые прямо противоречат одна другой, например, такие как $C = \{\{v_1\},\{\bar{v}_1\}\}$.

Так как наборы дизъюнкций, каждая из которых содержит только один литерал, легко поддаются выполнению, нас интересуют чуть более длинные дизъюнкции. Сколько требуется литералов в каждой дизъюнкции, чтобы превратить задачу из решаемой за полиномиальное время в труднорешаемую? Это происходит, когда каждая дизъюнкция содержит три литерала. Формальное определение выглядит таким образом:

ЗАДАЧА. Задача выполнимости в 3-конъюнктивной нормальной форме (3-SAT).

Вход. Коллекция дизъюнкций C, каждая из которых содержит ровно 3 литерала, над набором булевых переменных V.

Выход. Существует ли набор значений истинности для переменных V, при котором выполняется каждая дизъюнкция?

Поскольку это частный случай задачи выполнимости, сложность задачи 3-SAT влечет за собой сложность общей задачи выполнимости. Обратное утверждение неверно, т. к. сложность общей задачи выполнимости может зависеть от длин дизъюнкций. Сложность задачи 3-SAT можно показать с помощью сведения, которое преобразует каждый экземпляр задачи выполнимости в экземпляр задачи 3-SAT, не нарушая при этом его выполнимость.

Данное сведение преобразует каждую дизъюнкцию отдельно, в зависимости от ее длины, путем добавления новых дизъюнкции и булевых переменных. Допустим, что дизъюнкция C_i содержит k литералов:

- $k = 1$ означает, что $C_i = \{z_1\}$. Создадим две новые переменные v_1, v_2 и четыре новые дизъюнкции по 3 литерала каждая: $\{v_1, v_2, z_1\}$, $\{v_1, \bar{v}_2, z_1\}$, $\{\bar{v}_1, v_2, z_1\}$ и $\{\bar{v}_1, \bar{v}_2, z_1\}$.

Обратите внимание, что все эти четыре дизъюнкции могут быть одновременно выполнены только в том случае, если значение z_1 = ИСТИНА, что также означает выполнимость первоначальной дизъюнкции C_i;

- $k = 2$ означает, что $C_i = \{z_1, z_2\}$. Создадим одну новую переменную v_1 и две новые дизъюнкции по 3 литерала: $\{v_1, z_1, z_2\}$ и $\{\bar{v}_1, z_1, z_2\}$. Так же, как и в предыдущем случае, эти две дизъюнкции могут быть одновременно выполнены только в том случае, если, по крайней мере, значение одной из переменных z_1 и z_2 равно ИСТИНА, что также означает выполнимость C_i;
- $k = 3$ означает, что $C_i = \{z_1, z_2, z_2\}$. Дизъюнкция C_i копируется в экземпляр задачи 3-SAT без изменений: $\{z_1, z_2, z_2\}$;
- $k > 3$ означает, что $C_i = \{z_1, z_2, ..., z_n\}$. Создадим последовательно $n - 3$ новых переменных и $n - 2$ новых дизъюнкций, таких, что $C_{i,j} = \{v_{i,j-1}, z_{j+1}, \bar{v}_{i,j}\}$, $C_{i,1} = \{z_1, z_2, \bar{v}_{i,1}\}$ и $C_{i,n-2} = \{v_{i,n-3}, z_{n-1}, z_n\}$ для $2 \le j \le n - 2$.

Наиболее сложным является случай с длинными дизъюнкциями. Если ни один из первоначальных литералов не имеет значение ИСТИНА, то тогда не будет достаточного количества новых переменных, чтобы можно было выполнить все новые дизъюнкции. Дизъюнкцию $C_{i,1}$ можно выполнить, присвоив литералу $v_{i,1}$ значение ЛОЖЬ, но это заставит литерал $v_{i,2}$ принять значение ЛОЖЬ, и т. д. пока не окажется, что дизъюнкция $C_{i,n-2}$ не может быть выполнена. Но если значение некоторого литерала z_i равно ИСТИНА, то у нас имеется $n - 3$ свободных переменных и $n - 3$ оставшихся дизъюнкций, так что все они могут быть выполнены.

Это преобразование выполняется за время $O(m + n)$, если экземпляр задачи выполнимости имеет n дизъюнкций и m литералов. Так как любое решение задачи SAT является решением экземпляра задачи 3-SAT, а любое решение задачи 3-SAT описывает, как присвоить значения переменных, чтобы получить решение задачи SAT, то преобразованная задача эквивалентна первоначальной.

Обратите внимание, что незначительная модификация этой конструкции может быть использована для доказательста того, что задачи 4-SAT, 5-SAT и, вообще, ($k \ge 3$)-SAT также являются NP-полными. Но эта конструкция неприменима для задачи 2-SAT, т. к. у нас нет способа заполнить цепочку дизъюнкций. Но для решения задачи 2-SAT за линейное время можно использовать алгоритм обхода в глубину на соответствующем графе. Подробности см. в *разделе 14.10*.

9.5. Нестандартные сведения

Поскольку известно, что как общая задача выполнимости, так и задача выполнимости 3-SAT являются труднорешаемыми, в сведениях можно использовать любую из них. Обычно для этого лучше подходит задача 3-SAT, т. к. с ней проще работать. С целью предоставления дополнительных примеров и расширения нашего списка известных труднорешаемых задач мы рассмотрим еще два сложных сведения. Многие сведения довольно сложны, т. к. мы, по сути, программируем одну задачу на языке другой задачи, значительно отличающейся от первой.

Больше всего путаницы возникает при выборе направления сведения. Запомните, что требуется преобразовать каждый экземпляр заведомо NP-полной задачи в экземпляр

задачи, которая нас интересует. Если выполнить сведение в обратном направлении, то мы получим лишь медленный способ решения интересующей нас задачи в виде процедуры с экспоненциальным временем исполнения. Это может сбить с толку начинающего алгориста, т. к. указанное направление сведения кажется обратным требуемому. Обязательно разберитесь с правильным направлением сведения сейчас, и обращайтесь к этому материалу в случае затруднений по данному предмету.

9.5.1. Целочисленное программирование

Рассматриваемое в *разделе 13.6* целочисленное программирование представляет собой фундаментальную задачу комбинаторной оптимизации. Ее лучше всего рассматривать, как задачу линейного программирования, в которой переменные могут принимать только целочисленные значения (а не вещественные). Формальная постановка задачи имеет следующую форму:

ЗАДАЧА. Целочисленное программирование.

Вход. Набор целочисленных переменных V, набор неравенств над V, функция максимизации $f(V)$ и целое число B.

Выход. Существует ли набор целочисленных значений переменных V, для которого выполняются все неравенства и $f(V) \geq B$?

Рассмотрим два примера. Допустим, что

$v_1 \geq 1, v_2 \geq 0$

$v_1 + v_2 \leq 3$

$f(v):2v_2, B = 3$

Решением для данного входного экземпляра было бы $v_1 = 1$, $v_2 = 2$. Но не все задачи обладают реализуемыми решениями. Рассмотрим следующий входной экземпляр для этой же задачи:

$v_1 \geq 1, v_2 \geq 0$

$v_1 + v_2 \leq 3$

$f(v):2v_2, B = 5$

При данных ограничениях максимальное значение функции $f(v)$ равно $2 \times 2 = 4$, поэтому соответствующая задача разрешимости не имеет решения.

Для доказательства труднорешаемости задачи целочисленного программирования мы выполним сведение от задачи 3-SAT. В данном конкретном случае можно было воспользоваться общей задачей выполнимости, но, как правило, использование задачи 3-SAT облегчает сведение.

В каком направлении должно выполняться сведение? Мы хотим доказать труднорешаемость задачи целочисленного программирования, и мы знаем, что задача 3-SAT является труднорешаемой. Если задачу 3-SAT можно было бы решить, используя целочисленное программирование, и задача целочисленного программирования не была бы трудной, то и задача выполнимости не была бы трудной. Отсюда следует направление сведения: задачу 3-SAT нужно преобразовать в задачу целочисленного программирования.

Каким должно быть это преобразование? Каждый экземпляр задачи выполнимости содержит булевы переменные и дизъюнкции. Каждый экземпляр задачи целочисленного программирования содержит целочисленные переменные (т. е. значения, ограниченные множеством 0, 1, 2, ...) и ограничения. Имеет смысл сопоставить целочисленные переменные с булевыми и использовать ограничения в той же роли, какую играют дизъюнкции в исходной задаче.

Преобразованная задача целочисленного программирования будет содержать вдвое больше переменных, чем соответствующий экземпляр задачи SAT — по одной переменной для каждой исходной переменной и для ее дополнения. Для каждой переменной v_i в поставленной задаче добавляются следующие ограничения:

- чтобы каждая переменная V_i целочисленного программирования принимала только значения 0 или 1, добавляются ограничения $1 \geq V_i \geq 0$ и $1 \geq \bar{V}_i \geq 0$. С учетом целочисленности переменных эти значения соответствуют значениям ИСТИНА и ЛОЖЬ;
- для гарантии того, что одна и только одна из двух переменных в задаче целочисленного программирования, связанных с данной переменной в задаче выполнимости, имеет значение ИСТИНА, добавляются ограничения $1 \geq V_i + \bar{V}_i \geq 1$.

Для каждой дизъюнкции 3-SAT $C_i = \{z_1, z_2, z_3\}$ создадим ограничение $V_1 + V_2 + V_3 \geq 1$. Для удовлетворения данного ограничения, по крайней мере, одному из литералов в каждой дизъюнкции нужно присвоить значение 1, чтобы он соответствовал литералу ИСТИНА. Таким образом, выполнимость данного ограничения эквивалентна выполнимости дизъюнкции.

Функция максимизации, а также граница теряют свою актуальность, т. к. мы уже закодировали весь экземпляр задачи 3-SAT. Используя $f(v) = V_1$ и $B = 0$, мы гарантируем, что они не будут противоречить никаким значениям переменных, удовлетворяющим всем неравенствам. Очевидно, что это сведение можно выполнить за полиномиальное время. Чтобы установить, что при данном сведении сохраняется правильность ответа, нам нужно подтвердить следующие два обстоятельства:

- *любое решение задачи выполнимости дает решение задачи целочисленного программирования.* В любом решении задачи выполнимости литерал ИСТИНА соответствует значению 1 в решении задачи целочисленного программирования, т. к. дизъюнкция выполняется. Следовательно, сумма в каждом неравенстве больше или равна 1;
- *любое решение задачи целочисленного программирования дает решение первоначальной задачи выполнимости.* В любом решении данного экземпляра задачи целочисленного программирования всем переменным должно быть присвоено значение 0 или 1. Если $V_i = 1$, тогда литералу z_i присваивается значение ИСТИНА. Если $V_i = 0$, тогда литералу z_i присваивается значение ЛОЖЬ. Это законное присваивание значений, которое должно удовлетворять все дизъюнкции.

Так как сведение выполняется в обоих направлениях, то задача целочисленного программирования должна быть труднорешаемой. Обратите внимание на следующие свойства, которые остаются в силе и при доказательстве NP-полноты:

1. При данном сведении сохраняется структура задачи. Задача *не решается*, а просто преобразуется в другой формат.

2. Возможные экземпляры задачи целочисленного программирования, которые могут получиться в результате данного преобразования, представляют только небольшое подмножество всех возможных экземпляров этой задачи. Но так как некоторые из этих экземпляров являются труднорешаемыми, то и общая задача также должна быть труднорешаемой.
3. Это преобразование отражает суть того, почему задачи целочисленного программирования трудны для решения. Оно не имеет никакого отношения к большим коэффициентам или большим диапазонам переменных, т. к. ограничение множества значений до 0 и 1 является достаточным. Оно также не имеет ничего общего с большим количеством переменных в неравенствах. Задача целочисленного программирования является труднорешаемой потому, что выполнимость набора ограничений является труднорешаемой задачей. Внимательное изучение свойств задачи, необходимых для ее сведения, может многое рассказать нам о самой задаче.

9.5.2. Вершинное покрытие

Алгоритмическая теория графов изобилует сложными задачами. Прототипичной NP-полной задачей теории графов является задача поиска вершинного покрытия, которая была определена в *разделе 9.3.2* таким образом:

ЗАДАЧА. Найти вершинное покрытие.

Вход. Граф $G = (V, E)$ и целое число $k \leq |V|$.

Выход. Существует ли такое подмножество S, содержащее, самое большее, k вершин, для которого, по крайней мере, одна вершина каждого ребра $e \in E$ является элементом S?

Доказать сложность задачи о вершинном покрытии труднее, чем сложность задач в ранее рассмотренных сведениях, из-за различия структур соответствующих задач. Сведение задачи 3-SAT к задаче о вершинном покрытии требует создания графа G и границы k из переменных и дизъюнкций экземпляра задачи выполнимости.

Сначала мы преобразуем переменные задачи 3-SAT. Для каждой булевой переменной v_i мы создадим две вершины v_i и $\bar{v}_i$, которые соединяются ребром. Для покрытия всех ребер потребуется, по крайней мере, n вершин, т. к. ни одна пара этих ребер не будет иметь общей вершины.

Далее мы преобразуем дизъюнкции задачи 3-SAT. Для каждой из c дизъюнкций мы создадим три новые вершины, по одной для каждого литерала в каждой дизъюнкции. Эти три вершины каждой дизъюнкции будут соединены таким образом, чтобы получилось c треугольников. В любое вершинное покрытие этих треугольников должны быть включены, по крайней мере, две вершины каждого треугольника, при этом общее число вершин покрытия будет равным $2c$.

Наконец, мы соединим вместе эти два набора компонентов. Каждый литерал в вершинных компонентах соединяется с вершинами в компонентах дизъюнкций (треугольниках), разделяющих один и тот же литерал. Таким образом, из экземпляра задачи

3-SAT с n переменными и c дизъюнкциями создается граф, имеющий $2n + 3c$ вершин. Полное сведение для задачи 3-SAT $\{\{v_1, \bar{v}_3, \bar{v}_4\}, \{\bar{v}_1, v_2, \bar{v}_4\}\}$ показано на рис. 9.7.

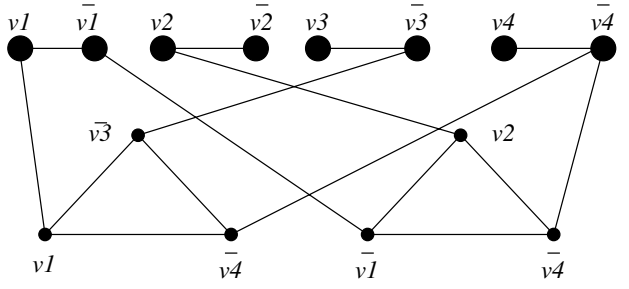


Рис. 9.7. Сведение экземпляра $\{\{v_1, \bar{v}_3, \bar{v}_4\}, \{\bar{v}_1, v_2, \bar{v}_4\}\}$ задачи выполнимости к задаче о вершинном покрытии

Этот граф создан таким образом, чтобы вершинное покрытие размером $n + 2c$ было возможным тогда и только тогда, когда первоначальное выражение является выполнимым. Согласно ранее приведенному анализу, каждое вершинное покрытие должно содержать, по крайней мере, $n + 2c$ вершин, т. к. добавление соединительных ребер в граф G не может уменьшить размер вершинного покрытия до значения, меньшего, чем размер покрытия из разъединенных элементов. Для доказательства правильности нашего сведения нам нужно продемонстрировать, что верны следующие утверждения.

- *Каждый выполняющий набор значений истинности дает вершинное покрытие.* Для данного выполняющего набора значений истинности для дизъюнкций выберите в качестве членов вершинного покрытия n вершин из вершинных компонентов, которые соответствуют истинным литералам. Так как это определяет выполняющий набор значений истинности, то истинный литерал из каждой дизъюнкции должен покрывать, по крайней мере, одно из трех поперечных ребер, соединяющих каждую вершину треугольника с компонентом вершин. Соответственно, выбирая две другие вершины каждого треугольника дизъюнкции, мы также выбираем все оставшиеся поперечные ребра.
- *Каждое вершинное покрытие дает выполняющий набор значений истинности.* В любом вершинном покрытии C размером $n + 2c$ ровно n вершин должны принадлежать вершинным компонентам. Пусть эти вершины первого этапа определяют набор значений истинности, в то время как остальные $2c$ вершины покрытия нужно распределить по две на каждый компонент дизъюнкции. В противном случае ребро компонента дизъюнкции должно остаться непокрытым. Эти вершины компонентов дизъюнкции могут покрывать только два из трех соединяющих перекрестных ребер для каждой дизъюнкции. Поэтому если C обеспечивает вершинное покрытие, то, по крайней мере, одно поперечное ребро в каждой дизъюнкции должно быть покрыто, а это означает, что соответствующий набор значений истинности является выполняющим для всех дизъюнкций.

Данное доказательство сложности задачи о вершинном покрытии, объединенное в цепочку со сведениями задачи о клике и задачи о независимом множестве из *раздела 9.3.2*, составляет библиотеку сложных задач теории графов, которые мы можем использовать для облегчения доказательства сложности других задач.

Подведение итогов

Небольшой набор NP-полных задач (3-SAT, вершинное покрытие, разделение множества целых чисел и гамильтонов цикл) является достаточным для доказательства сложности большинства других сложных задач.

9.6. Искусство доказательства сложности

Доказательство сложности задач требует определенного мастерства. Однако, приобретя навык, вы обнаружите, что выполнение сведений может быть на удивление простым процессом. Малоизвестной особенностью доказательств NP-полноты является тот факт, что их легче создать, чем объяснить, аналогично тому, как часто бывает легче заново написать код, чем разбираться в старом.

Умение оценить вероятность того, что задача трудна для решения, требует опыта. Возможно, что самым быстрым способом набраться такого опыта будет внимательное изучение примеров в каталоге задач. Незначительное изменение формулировки задачи может сделать полиномиальную задачу NP-полной. Задача поиска кратчайшего пути в графе является легкой, в то время как задача поиска самого длинного пути в графе является сложной. Задача построения маршрута, который проходит по всем ребрам графа ровно один раз (цикл Эйлера), является легкой, а вот задача построения маршрута, который проходит через каждую вершину графа (гамильтонов цикл) является сложной.

Если вы подозреваете, что задача является NP-полной, сначала проверьте, не содержится ли она в книге [GJ79], в которой перечислены несколько сотен известных NP-полных задач. Велика вероятность, что вы найдете свою задачу в этой книге.

Если же нет, то вы можете воспользоваться следующими советами по доказательству сложности задач:

- *Примите меры к тому, чтобы ваша исходная задача была как можно проще (т. е. максимально ограничена).*

  Во-первых, никогда не пытайтесь использовать в качестве исходной задачи общую задачу коммивояжера. Вместо этого используйте задачу поиска гамильтонова цикла, в которой все веса равны 1 или ∞. Еще лучше взять задачу поиска гамильтонова пути, чтобы не нужно было беспокоиться о замыкании цикла. Но самый лучший вариант — это задача поиска гамильтонова пути в ориентированном планарном графе, где степень каждой вершины равна 3. Все эти задачи имеют одинаковую сложность, но чем больше ограничений имеет сводимая задача, тем проще будет процесс сведения.

  Во-вторых, никогда не пытайтесь использовать полную задачу выполнимости для доказательства сложности. Начните с задачи выполнимости 3-SAT. На самом деле, вам даже не нужно использовать задачу выполнимости полной 3-SAT. Вместо этого можно использовать задачу планарной 3-SAT, для которой существует способ представления дизъюнкции в виде плоского графа, такого, что все экземпляры одного и того же литерала можно соединить вместе, избегая пересечения ребер. Это свойство полезно при доказательстве сложности геометрических задач. Все эти задачи одинаковой сложности, и поэтому сведения NP-полноты с использованием любой из них являются одинаково убедительными.

- *Сделайте целевую задачу максимально сложной.*

  Не бойтесь добавлять дополнительные ограничения или разрешения, чтобы сделать целевую задачу более общей. Возможно, задачу на неориентированном графе можно обобщить до задачи на ориентированном графе и, таким образом, облегчить доказательство ее сложности. Имея доказательство сложности для общей задачи, можно возвратиться назад и попытаться упростить целевую задачу.

- *Выбирайте исходную задачу на основании веских аргументов.*

  Выбор правильной исходной задачи играет важную роль в доказательстве сложности задачи. Здесь очень легко ошибиться, хотя теоретически одна NP-полная задача подходит так же хорошо, как и любая другая. Пытаясь доказать сложность задачи, некоторые люди просматривают списки с десятками задач в поисках наиболее подходящей. Это непрофессиональный подход. Натолкнувшись на подходящую задачу, они, вероятнее всего, не догадаются, что именно она им и нужна.

  Я использую четыре (и только четыре!) задачи в качестве кандидатов для моих исходных сложных задач. Ограничение количества исходных задач четырьмя означает, что я могу знать многое о каждой из них, например, какие варианты задачи являются сложными, а какие нет. Перечислю свои любимые исходные задачи:

  - *Задача 3-SAT.* Многократно проверенный вариант. Если ни одна из перечисленных ниже задач не кажется подходящей, я возвращаюсь к этой первоначальной исходной задаче.
  - *Задача о разделении множества целых чисел.* Это единственно возможный выбор в тех случаях, когда кажется, что для доказательства сложности требуется использовать большие числа.
  - *Задача о вершинном покрытии.* Это подходящий вариант для любой задачи теории графов, чья сложность зависит от *выбора*. Решение задач поиска хроматического числа, клики и независимого множества содержит элемент выбора правильного подмножества вершин или ребер.
  - *Задача о гамильтоновом цикле.* Подходит для любой задачи на графах, чья сложность зависит от *упорядочивания*. Если вы пытаетесь разработать маршрут или календарный план, то задача о гамильтоновом цикле, вероятнее всего, является вашим средством для решения этой задачи.

- *Повышайте стоимость нежелательного выбора.*

  Многие люди испытывают робость при доказательстве сложности задач. Мы пытаемся преобразовать одну задачу в другую, как можно меньше отступив от оригинальной задачи. Это легче всего делать, применяя строгие штрафные санкции за любое отклонение от целевого решения. Вы должны мыслить примерно следующим образом: "Если выбирается элемент a, то придется выбрать огромное множество S, которое не позволит найти оптимальное решение". Чем дороже последствия нежелательных действий, тем легче будет доказать эквивалентность задач.

- *Разработайте стратегический план, а затем создавайте компоненты для тактических действий.*

Спрашивайте себя:

- Как мне добиться того, чтобы был выбран вариант *A* или *B*, но не оба одновременно?
- Что сделать, чтобы вариант *A* был выбран раньше варианта *B*?
- Как поступить с невыбранными вариантами?

После того, как вы получите общее представление о действиях, выполняемых компонентами, вы можете задуматься о том, как создать эти компоненты.

♦ *Если вы испытываете трудности на каком-то этапе, переключайтесь с поиска алгоритма на поиск сведения и обратно.*

Иногда сложность задачи нельзя доказать по той причине, что для ее решения существует эффективный алгоритм! Применение таких методов, как динамическое программирование или сведение задач к сложным, но полиномиальным задачам на графах, например, к задаче о сопоставлении пар или задаче потоков в сети, может привести к неожиданным результатам. Каждый раз, когда вы не можете доказать сложность задачи, имеет смысл попытаться пересмотреть свое мнение.

9.7. История из жизни. Наперегонки со временем

Аудитория стремительно теряла внимание к лекции. У некоторых студентов слипались глаза, а другие уже клевали носом.

До конца моей лекции по NP-полноте оставалось двадцать минут, и студентов нельзя было винить за их реакцию. Мы уже рассмотрели несколько сведений, аналогичных представленным в предыдущих разделах. Но сведения NP-полных задач легче создать, чем объяснить или понять. Было необходимо продемонстрировать студентам сведение в процессе его создания, чтобы они могли понять, как оно работает.

Я потянулся за книгой "Computers and Intractability" [GJ79], выручавшей меня в трудные моменты. Приложение к ней содержит список из более чем четырех сотен разных известных NP-полных задач.

— Итак! — сказал я достаточно громко, чтобы заставить вздрогнуть дремавших в задних рядах. — Доказательства NP-полноты настолько рутинны, что их можно создавать по требованию. Мне нужен помощник. Есть добровольцы?

В передних рядах поднялось несколько рук. Я вызвал к доске одного из студентов.

— Выбери произвольную задачу из списка в приложении в этой книге. Я могу доказать сложность любой из этих задач в течение оставшихся семнадцати минут лекции.

Теперь я определенно овладел их вниманием. Но это было все равно, что жонглировать работающими бензопилами. Я должен был выдать результат, не изрезав себя в клочья.

Студент выбрал задачу. — Хорошо, докажите сложность *задачи о неэквивалентности программ с присваиваниями*, — сказал он.

— Никогда раньше не слышал об этой задаче. Прочитай мне условие, чтобы я мог написать его на доске.

Задача определялась таким образом:

ЗАДАЧА. Неэквивалентность программ с присваиваниями (Inequivalence of Programs with Assignments).

Вход. Конечное множество переменных X, множество их значений V и две программы P_1 и P_2, каждая из которых состоит из последовательности присваиваний вида:

$$x_0 \leftarrow \text{if } (x_1 = x_2) \text{ then } x_3 \text{ else } x_4,$$

где x_i является элементом X.

Выход. Возможно ли первоначально присвоить каждой переменной из множества X значение из множества V таким образом, чтобы две данные программы выдавали разные конечные значения для некой переменной из множества X?

Я посмотрел на часы. Пятнадцать минут до конца лекции. Мне предстояло решить языковую задачу. Входом задачи были две программы с переменными, и нужно было проверить, всегда ли они выдают одинаковый результат.

— В первую очередь, самое важное. Нам нужно выбрать исходную задачу для нашего сведения. Какая задача подойдет для этого? Задача разделения множества целых чисел? Задача 3-SAT? Задача о вершинном покрытии или о гамильтоновом пути?

Овладев вниманием аудитории, я рассуждал вслух. — Так как наша целевая задача не является задачей на графах или численной задачей, давайте рассмотрим старую добрую задачу 3-SAT. По-видимому, эти две задачи похожи в некоторых аспектах. Задача 3-SAT имеет переменные и данная задача тоже. Чтобы сделать нашу задачу еще более похожей на задачу 3-SAT, можно попробовать ограничить ее переменные двумя значения, т. е. V = {ИСТИНА, ЛОЖЬ}. Да, это удобно.

До конца лекции оставалось 14 минут. — В каком направлении должно идти наше сведение? От задачи 3-SAT к языковой задаче или от языковой задачи к задаче 3-SAT?

Кто-то из первого ряда пробормотал, что от задачи 3-SAT к языковой задаче.

— Совершенно верно. Поэтому нам нужно преобразовать наш набор дизъюнкций в две программы. Как мы можем это сделать? Можно попробовать разделить дизъюнкции на два набора и написать отдельные программы для каждого из них. Но как их разделить? Я не вижу, как это можно сделать каким-либо естественным способом, т. к. удаление любой дизъюнкции из программы может неожиданно сделать невыполнимую формулу выполнимой, таким образом полностью меняя ответ. Давайте попробуем что-нибудь другое. Мы можем преобразовать все дизъюнкции в одну программу, а потом сделать вторую программу тривиальной. Например, вторая программа может игнорировать вход и всегда выводить или только значение ИСТИНА, или только значение ЛОЖЬ. Это выглядит *намного* лучше.

Я продолжал рассуждать вслух. В этом не было ничего особенного. Но я заставил студентов слушать меня!

— Как нам превратить набор дизъюнкций в программу? Мы хотим знать, может ли этот набор дизъюнкций быть выполнен, т. е. существует ли набор значений переменных, делающий его значение истинным. Допустим, что мы создали программу для проверки выполнимости $c_1 = (x_1, \bar{x}_2, x_3)$.

Несколько минут я водил мелом по доске, пока у меня не получилась правильная программа для эмулирования дизъюнкции. Я предположил, что у нас есть доступ к константам для значений ИСТИНА и ЛОЖЬ:

c_1 = if (x_1 = ИСТИНА) then ИСТИНА else ЛОЖЬ

c_1 = if (x_2 = ЛОЖЬ) then ИСТИНА else c_1

c_1 = if (x_2 = ИСТИНА) then ИСТИНА else c_1

— Теперь у меня есть метод для оценки истинности каждой дизъюнкции. Я могу сделать то же самое для оценки выполнимости всех дизъюнкций.

sat = if (c_1 = ИСТИНА) then ИСТИНА else ЛОЖЬ

sat = if (c_2 = ИСТИНА) then *sat* else ЛОЖЬ

...

sat = if (c_n = ИСТИНА) then *sat* else ЛОЖЬ

В задних рядах возникло оживление. Они увидели луч надежды покинуть аудиторию вовремя. До звонка оставалось две минуты.

— Итак, у нас имеется программа, которая возвращает истинное значение тогда и только тогда, когда переменным можно присвоить значения так, чтобы выполнялись все дизъюнкции. Нам нужна вторая программа, чтобы закончить доказательство. А если мы напишем *sat* = ЛОЖЬ. Да, это все что нам нужно. В нашей языковой задаче спрашивается, выводят ли две программы всегда одинаковый результат, независимо от значений, присвоенных переменным. Если дизъюнкции являются выполнимыми, то это означает, что можно присвоить переменным значения таким образом, что длинная программа будет выводить истинное значение. Проверка эквивалентности программ это то же самое, что и проверка дизъюнкций на выполнимость.

Я победно вскинул руки. — Итак, задача является NP-полной. — Как только я сказал последнее слово, прозвучал звонок.

9.8. История из жизни. Полный провал

Этот педагогический прием с выбором произвольной NP-полной задачи из 400 с лишним задач в книге [GJ79] и доказательством ее сложности на лету мне так понравился, что я начал пользоваться им постоянно. Прием удавался мне восемь раз подряд. Однако наступил день, когда я потерпел фиаско.

На этот раз группа проголосовала за сведение из раздела по теории графов, а студент-доброволец выбрал задачу № 30. Постановка задачи GT30 выглядит таким образом:

ЗАДАЧА. Найти односвязный подграф.

Вход. Ориентированный граф $G = (V, A)$, положительное целое число $k \leq |A|$.

Выход. Существует ли такое подмножество дуг $A' \in A$, где $|A'| \geq k$, для которого в графе $G' = (V, A')$ существует, самое большее, один ориентированный путь между любой парой вершин?

— Это задача о выборе, — определил я, как только задача была озвучена. Ведь нам нужно было выбрать такое максимально возможное подмножество дуг, которое не со-

держало бы пары вершин, соединенных множественными путями. А это означало, что предпочтительной задачей для сведения является задача вершинного покрытия.

Я немного поразмыслил о том, каким образом эти две задачи были похожими друг на друга. В обеих задачах нужно было найти некоторые подмножества, хотя в задаче о вершинном покрытии требовалось найти подмножества вершин, а в задаче об односвязном подграфе нужно было найти подмножества ребер. Кроме этого, в задаче о вершинном покрытии требовалось найти минимально возможное подмножество, в то время как в задаче об односвязном подграфе нужно было определить максимально возможное подмножество. В исходной задаче были неориентированные ребра, а в целевой — ориентированные дуги, так что мне нужно было добавить в сведение ориентацию ребер.

Я должен был каким-то образом придать ориентацию ребрам графа вершинного покрытия. Можно было попробовать заменить каждое неориентированное ребро (x, y) дугой, идущей, скажем, от y к x. Но в зависимости от выбранного направления дуги получились бы совсем разные ориентированные графы. Поиск "правильной" ориентации ребер мог оказаться трудной задачей, слишком трудной для использования на этапе преобразования.

Я понимал, что ребра можно ориентировать так, чтобы получившийся граф стал бесконтурным орграфом. Но что это дает? В бесконтурных орграфах пары вершин могут быть связаны большим количеством ориентированных путей.

В качестве варианта можно было попробовать заменить каждое неориентированное ребро (x, y) двумя дугами — от y к x и от x к y. Теперь не нужно было выбирать правильные дуги для моего сведения, но граф стал очень сложным. Я никак не мог найти способ предотвратить множественные нежелательные пути между парами вершин.

Время лекции подходило к концу. В последние десять минут меня охватила паника, т. к. я понял, что на этот раз я не смогу предоставить доказательство.

Нет худшего чувства, чем чувство профессора, завалившего лекцию. Ты стоишь у доски, что-то бормоча, и ясно видишь, что студенты не понимают, о чем идет речь, но догадываются, что ты сам не понимаешь, о чем говоришь. Прозвучал звонок и студенты стали покидать аудиторию, одни с сочувствующим выражением на лице, другие с ехидными ухмылками.

Я пообещал им предоставить решение на следующем занятии, но каждый раз, когда я думал об этом, я застревал в одном и том же месте. Я даже попробовал поступить нечестно и найти доказательство в научном журнале, указанном в ссылке в книге [GJ79]. Но ссылка была на неопубликованный доклад 30-летней давности, который нельзя было найти ни в Интернете, ни в библиотеке.

Мысль о следующем занятии, последней лекции семестра, приводила меня в ужас. Но в ночь перед лекцией решение явилось ко мне во сне. — *Раздели каждое ребро пополам*, — сказал мне голос. Я вздрогнул, проснулся и посмотрел на часы. Было три часа ночи.

Вскочил с кровати и набросал доказательство. Допустим, я заменю каждое неориентированное ребро (x, y) конструкцией, состоящей из новой центральной вершины v_{xy} с исходящими из нее дугами к вершинам x и y соответственно. Неплохо. Между каки-

ми вершинами могут существовать множественные пути? Новые вершины имели только исходящие ребра, так что только они могли служить источником множественных путей. Старые вершины имели только входящие ребра. Существует максимум один способ попасть из новой вершины-истока в любую из оригинальных вершин графа вершинного покрытия, так что старые вершины не могли дать множественных путей.

Теперь добавим узел стока *s* и направим в него ребра из всех оригинальных вершин. От каждой новой вершины к этому стоку будет идти ровно два пути — по одному через каждую из первоначальных вершин, смежных с ней. Один из этих путей нужно разорвать, чтобы создать односвязный подграф. Как это сделать? Для разъединения можно выбрать одну из двух вершин, удалив дугу (x, s) или (y, s) новой вершины v_{xy}. Чтобы получить подграф максимального размера, мы ищем наименьшее количество дуг, подлежащих удалению. Удалим исходящие дуги хотя бы у одной из двух вершин, определяющих первоначальное ребро. *Но ведь это же самое, что и поиск вершинного покрытия в этом графе!* Полученное сведение показано на рис. 9.8.

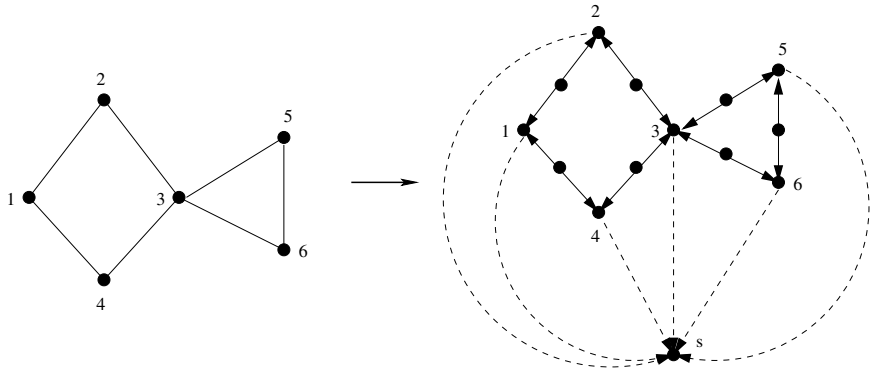


Рис. 9.8. Сведение задачи о вершинном покрытии к задаче односвязного подграфа посредством разделения ребер и добавления узла стока

Представление этого доказательства на следующем занятии несколько подняло мою самооценку, но гораздо важнее тот факт, что оно подтвердило правильность провозглашенных мною принципов доказательства сложности задач. Обратите внимание, что в конечном счете сведение не было таким уж сложным: достаточно разорвать ребра и добавить узел стока. Выполнение сведения NP-полных задач нередко удивляет своей простотой, нужно лишь подойти к нему с правильной стороны.

9.9. Сравнение классов сложности P и NP

Теория NP-полноты основана на строгих определениях из теории автоматов и формальных языков. Новички, не обладающие базовыми знаниями, обычно находят применяемую терминологию трудной для понимания или употребляют ее неправильно. Знание терминологии не является чем-то действительно необходимым при решении практических вопросов. Но, несмотря на все сказанное, вопрос "Верно ли, что P = NP?" представляет собой важнейшую задачу в теории вычислительных систем, и каждый образованный алгорист должен понимать, о чем идет речь.

9.9.1. Верификация решения и поиск решения

В сравнении классов сложности P и NP главным вопросом является, действительно ли *верификация* решения представляет более легкую задачу, чем первоначальный *поиск* решения. Допустим, что при сдаче экзамена вы "случайно" заметили ответ у студента, сидящего рядом с вами. Принесет ли это вам какую-либо пользу? Вы бы не рискнули сдать этот ответ, не проверив его, т. к. вы способный студент и могли бы решить данную задачу самостоятельно, если бы уделили ей достаточно времени. Но при этом важно, можете ли вы проверить правильность подсмотренного ответа на задачу быстрее, чем решить саму задачу самостоятельно.

В случае NP-полных задач разрешимости, которые обсуждались в этой главе, ситуация *кажется* очевидной. Рассмотрим примеры.

- Можно ли проверить, что граф содержит маршрут коммивояжера с наибольшим весом k, если дан порядок вершин маршрута? Да. Просто сложим вместе веса ребер маршрута и покажем, что общий вес равен, самое большее, k. Это ведь легче, чем найти сам путь, *не так ли*?
- Можно ли проверить, что данный набор значений истинности представляет решение для данной задачи выполнимости? Да. Просто проверим каждую дизъюнкцию и убедимся, что она содержит, по крайней мере, один истинный литерал из данного набора значений истинности. Это ведь легче, чем найти выполняющий набор значений истинности, *не так ли*?
- Можно ли проверить, что граф G содержит вершинное покрытие, состоящее из, самое большее, k вершин, если известно подмножество S, состоящее из, самое большее, k вершин, составляющих данное покрытие? Да. Просто обходим каждое ребро (u, v) графа G и проверяем, что или u или v является элементом S. Это ведь легче, чем найти само вершинное покрытие, *не так ли*?

На первый взгляд, все просто. Данные решения можно проверить за линейное время для всех трех задач, однако для решения любой из них неизвестно никакого алгоритма, кроме полного перебора. Но проблема в том, что у нас нет строгого *доказательства* нижней границы, препятствующей существованию эффективного алгоритма для решения этих задач. Возможно, в действительности существуют полиномиальные алгоритмы (скажем, с временем исполнения $O(n^7)$), но мы просто недостаточно хорошо их искали.

9.9.2. Классы сложности P и NP

Каждая четко определенная алгоритмическая задача должна иметь максимально быстрый алгоритм решения, точнее говоря, максимально быстрый в терминах "O-большое" для наихудшего случая.

Класс P можно рассматривать как закрытый клуб алгоритмических задач, членом которого задача может стать, только продемонстрировав, что для ее решения существует алгоритм с полиномиальным временем исполнения. Полноправными членами этого клуба P являются, например, задача поиска кратчайшего пути, задача поиска минимального остовного дерева и задача календарного планирования. Сокращение P означает *полиномиальное* (polynomial) *время исполнения*.

Менее престижный клуб открывает свои двери любой алгоритмической задаче, решение которой можно *проверить* за полиномиальное время. Как было показано ранее, членами этого клуба являются задача коммивояжера, задача выполнимости и задача о вершинном покрытии, ни одна из которых в настоящее время не квалифицирована должным образом, чтобы стать членом клуба P. Но все члены закрытого клуба P автоматически являются членами этого второго, не столь престижного, клуба. Если задачу можно решить с самого начала за полиномиальное время, то ее решение определенно можно проверить столь же быстро: просто решить ее сначала и посмотреть, совпадает ли полученное вами решение с тем, которое было заявлено.

Этот менее престижный клуб называется NP. Это сокращение можно расшифровать как "not-necessarily polynomial-time" (не обязательно полиномиальное время исполнения)[1].

Важнейший вопрос состоит в том, действительно ли класс NP содержит задачи, которые не могут быть членами класса P. Если такой задачи не существует, то классы должны быть одинаковыми и P = NP. Но если существует хотя бы одна такая задача, то классы разные и P ≠ NP. Большинство алгористов и теоретиков сложности вычислений придерживаются мнения, что классы разные, т. е. что P ≠ NP, но для него требуется гораздо более строгое доказательство, чем заявление: "Я не могу найти достаточно быстрый алгоритм".

9.9.3. Почему задача выполнимости является самой сложной из всех сложных задач?

Существует громадное дерево сведений задач NP-полноты, которое всецело основано на сложности задачи выполнимости. Часть задач из этого дерева, рассмотренных в этой главе (и доказанных в других источниках), показана на рис. 9.2.

Но здесь возникает одна тонкость. Каковы были бы последствия, если бы кто-то действительно нашел алгоритм с полиномиальным временем исполнения для решения задачи выполнимости? Существование эффективного алгоритма для любой NP-полной задачи (скажем, задачи коммивояжера) подразумевает существование эффективных алгоритмов для решения всех задач на отрезке пути в дереве сведений между задачей коммивояжера и задачей выполнимости (задача о гамильтоновом цикле, задача о вершинном покрытии и задача 3-SAT). Но существование эффективного алгоритма для решения задачи выполнимости не дает нам ничего сейчас же, т. к. путь сведений от задачи SAT к задаче SAT не содержит никаких других задач.

Не будем впадать в панику. Существует замечательное сведение, называемое теоремой Кука, которое сводит все задачи класса NP к задаче выполнимости. Таким образом, если доказать, что задача выполнимости, или любая NP-полная задача, является членом класса P, то за ней последуют *все* другие задачи в классе NP, из чего будет следовать, что P = NP. Так как, в сущности, каждая упомянутая в данной книге задача явля-

[1] В действительности это сокращение означает "nondetenninistic polynomial-time" (недетерминированное полиномиальное время. исполнения) и является термином из области теории недетерминированных авто матов.

ется членом класса NP, то результаты такого развития событий были бы весьма значительными и удивительными.

Теорема Кука доказывает, что задача выполнимости является такой же сложной, как и любая другая задача из класса NP. Кроме этого, она также доказывает, что каждая NP-полная задача такая же сложная, как и любая другая. Здесь уместно вспомнить про эффект домино. Но то обстоятельство, что мы не можем найти эффективного алгоритма ни для одной из этих задач, дает веское основание полагать, что все они действительно сложные и что, вероятно, $P \neq NP$.

9.9.4. NP-сложность по сравнению с NP-полнотой

Сейчас мы обсудим различие между NP-сложностью задачи и ее NP-полнотой. Я имею склонность к несколько вольному употреблению терминологии, а между этими двумя понятиями существует тонкая (обычно несущественная) разница.

Говорят, что задача является *NP-сложной*, если, подобно задаче выполнимости, она, по крайней мере, такая же сложная, как любая другая задача класса NP. Говорят, что задача является *NP-полной*, если она NP-сложная и также является членом класса NP. Так как класс NP является таким большим классом задач, большинство NP-сложных задач, с которыми вам придется столкнуться, в действительности будут NP-полными, и вы всегда можете предоставить стратегию проверки решения задачи (обычно достаточно простую). Все рассмотренные в этой книге NP-сложные задачи также являются NP-полными.

Тем не менее, существуют задачи, которые кажутся NP-сложными, но при этом не являются членами класса NP. Такие задачи могут быть *даже более сложными*, чем NP-полные задачи! В качестве примера сложной задачи, не являющейся членом класса NP, можно привести игру для двух участников, такую, как шахматы. Представьте себе, что вы сели играть в шахматы с самоуверенным игроком, который играет белыми. Он начинает игру ходом центральной пешки на два поля и объявляет вам мат. Единственным способом доказать его правоту будет создание полного дерева всех ваших возможных ходов и его лучших ответных ходов и демонстрация невозможности вашего выигрыша в текущей позиции. Количество узлов этого полного дерева будет экспоненциально зависеть от его высоты, равной количеству ходов, которые вы сможете сделать, перед тем как проиграть, применяя максимально эффективную защиту.

Ясно, что это дерево нельзя создать и проанализировать за полиномиальное время, поэтому задача не является членом класса NP.

9.10. Решение NP-полных задач

Человек практичный никогда не останавливается на доказательстве того, что задача является NP-полной. Очевидно, у него были какие-то причины решить ее. Эти причины не исчезли, когда он узнал, что для решения задачи не существует алгоритма с полиномиальным временем исполнения. Ему все равно нужна программа для решения этой задачи. Все, что известно, так это лишь невозможность создания программы для быстрого оптимального решения задачи в наихудшем случае.

Однако для достижения цели остаются три варианта:

- *алгоритмы, эффективные для средних случаев задачи.* В качестве примеров таких алгоритмов можно назвать алгоритмы поиска с возвратом, в которых выполняются значительные отсечения;
- *эвристические алгоритмы.* Эвристические методы, такие как имитация отжига или жадные алгоритмы, можно использовать, чтобы быстро найти решение, но без гарантии, что это решение будет наилучшим;
- *аппроксимирующие алгоритмы.* Теория NP-полноты только оговаривает сложность получения решения задачи. Но используя специализированные, ориентированные на конкретную задачу эвристические алгоритмы, скорее всего можно получить близкое к оптимальному решение для всех возможных экземпляров задачи.

Аппроксимирующие алгоритмы обеспечивают решение с гарантией того, что оптимальное решение не будет намного лучше. Таким образом, используя аппроксимирующий алгоритм для решения NP-полной задачи, вы никогда не получите крайне неудовлетворительный ответ. Независимо от входного экземпляра задачи, вы неизбежно будете совершать правильные действия. Кроме того, аппроксимирующие алгоритмы с удачно подобранными границами концептуально просты, работают быстро и легко поддаются программированию.

Однако неясным остается следующее обстоятельство. Насколько решение, полученное с помощью аппроксимирующего алгоритма, сравнимо с решением, которое можно было бы получить с помощью эвристического алгоритма, не дающего никаких гарантий? Вообще говоря, оно может оказаться как лучше, так и хуже. Положив деньги в банк, вы получаете гарантию невысоких процентов прибыли без всякого риска. Вложив эти же деньги в акции, вы, скорее всего, получите более высокую прибыль, но при этом у вас не будет никаких гарантий.

Один из способов получения наилучшего результата от аппроксимирующего и эвристического алгоритмов заключается в том, что вы решаете данный экземпляр задачи с помощью каждого из них и выбираете тот алгоритм, который дает лучшее решение. Таким образом, вы получаете гарантированное решение и дополнительный шанс на получение лучшего решения. При применении эвристических алгоритмов для решения сложных задач вы можете рассчитывать на двойной успех.

9.10.1. Аппроксимация вершинного покрытия

Как мы уже видели, задача поиска минимального вершинного покрытия графа является NP-полной. Но с помощью очень простой процедуры (листинг 9.8) можно быстро найти покрытие, которое, самое большее, вдвое больше оптимального.

Листинг 9.8. Аппроксимирующий алгоритм поиска вершинного покрытия графа

```
VertexCover(G = (V, E))
   While (E ≠ 0) do:
      Выбираем произвольное ребро (u, v) ≤ E
      Добавляем обе вершины u и v к вершинному покрытию
      Удаляем из E все ребра, входящие в u или v
```

Должно быть очевидным, что данная процедура всегда выдает вершинное покрытие, т. к. каждое ребро удаляется только после добавления инцидентной вершины к покрытию. Более интересным является утверждение, что любое другое вершинное покрытие должно содержать, по крайней мере, в два раза меньше вершин данного покрытия. Почему? Рассмотрим только те ребра, выбранные алгоритмом, которые образуют паросочетание в графе (а их не более $n/2$). Ни одна пара этих ребер не может иметь общую вершину. Поэтому любое покрытие, состоящее только из этих ребер, должно включать хотя бы одну вершину на каждое ребро, что делает его, по крайней мере, вдвое меньшим, чем вершинное покрытие, полученное с помощью этого "жадного" алгоритма.

Стоит отметить несколько интересных аспектов этого алгоритма.

- *Хотя процедура проста, она не так глупа.* Производительность многих кажущихся более интеллектуальными эвристических алгоритмов может оказаться намного ниже в наихудшем случае. Например, почему бы не модифицировать эту процедуру, чтобы для получения вершинного покрытия вместо обеих вершин выбирать только одну из них? В конце концов, выбранное ребро будет с тем же успехом покрыто только одной вершиной. Но посмотрим на звездообразный граф на рис. 9.9. Если не выбрать центральную вершину, вершинное покрытие окажется крайне неудачным.

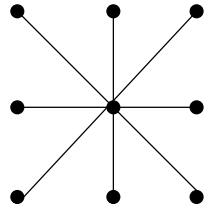


Рис. 9.9. Звездообразный граф

Этот эвристический алгоритм выдаст двухвершинное покрытие, в то время как эвристический алгоритм, выбирающий одну вершину, выдаст покрытие размером до $n - 1$ вершин, если нам не повезет, и алгоритм будет постоянно выбирать листовой узел вместо центрального в качестве вершины покрытия, которую следует сохранить.

- *"Жадный" алгоритм — это не всегда то, что нужно.* Возможно, что самый естественный алгоритм поиска вершинного покрытия будет постоянно выбирать и удалять вершину наивысшей оставшейся степени для данного вершинного покрытия. В конце концов, эта вершина покроет наибольшее количество возможных ребер. Но в случае необходимости выбора между одинаковыми или почти одинаковыми вершинами этот эвристический алгоритм может значительно отклониться от правильного пути. В наихудшем случае он может выдать покрытие больше оптимального в $\Theta(\lg n)$.
- *Усложнение эвристического алгоритма не обязательно улучшает его.* Эвристический алгоритм легко усложнить, вставляя в него дополнительные возможности обработки. Например, в аппроксимирующей процедуре в листинге 9.8 не указывается, какое ребро выбирается следующим. Может показаться разумным следующим вы-

бирать ребро с вершинами наивысшей степени. Но это не сузит границы наихудшего случая и только сделает более трудным анализ работы алгоритма.

- *Корректное завершение алгоритма позволяет улучшить результат.* Дополнительным преимуществом создания простых эвристических алгоритмов является то, что их часто можно модифицировать для получения лучших практических решений, при этом не ослабляя пределы аппроксимации. Например, операция постобработки, которая удаляет все ненужные вершины из покрытия, на практике может только улучшить результат, даже если она не принесет никакой пользы теоретическому пределу наихудшего случая.

Важной характеристикой аппроксимирующих алгоритмов является отношение размера полученного решения к нижней границе оптимального решения. Не следует размышлять о том, насколько хорошим могло бы быть наше решение; мы должны думать о наихудшем случае, т. е. о том, насколько плохим оно могло бы быть.

9.10.2. Задача коммивояжера в евклидовом пространстве

В большинстве естественных приложений задачи коммивояжера прямые маршруты по своей природе короче, чем обходные. Например, если в качестве веса ребер графа выбрать расстояние между городами по прямой, то кратчайший путь от x к y всегда будет проходить по прямой.

Веса ребер, назначаемые в соответствии с евклидовой геометрией, удовлетворяют аксиоме треугольника, т. е. для любой тройки вершин u, v и w выполняется неравенство $d(u, w) \leq d(u, v) + d(v, w)$. Интуитивная очевидность этого условия демонстрируется на рис. 9.10. Аксиома треугольника $d(u, w) \leq d(u, v) + d(v, w)$ обычно справедлива в геометрических задачах и задачах на взвешенных графах.

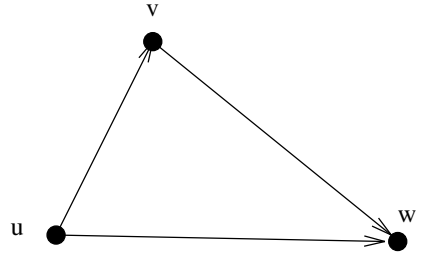


Рис. 9.10. Аксиома треугольника

Стоимость билета на авиарейс является примером функции расстояний, которая нарушает аксиому треугольника, т. к. иногда транзитный рейс обходится дешевле, чем прямой полет до города назначения. Задача коммивояжера остается сложной, когда расстояния являются евклидовыми расстояниями на плоскости.

Оптимальный маршрут коммивояжера можно аппроксимировать, используя минимальные остовные деревья или графы, подчиняющиеся аксиоме треугольника. Прежде всего, обратите внимание на то, что весом минимального остовного дерева является нижняя граница стоимости оптимального маршрута. Почему? Удаление любого ребра

из маршрута оставляет путь, общий вес которого должен быть не большим, чем вес первоначального маршрута. Этот путь не содержит циклов, вследствие чего он является деревом, что означает, что его вес равен, по крайней мере, весу минимального остовного дерева. Таким образом, вес минимального остовного дерева дает нам нижнюю границу оптимального маршрута.

Теперь рассмотрим, что происходит при обходе в глубину остовного дерева. Каждое ребро мы посещаем дважды: один раз спускаясь по дереву при открытии ребра, а второй — возвращаясь наверх после исследования всего поддерева. Например, при обходе в глубину, показанном на рис. 9.11, вершины посещаются в следующем порядке: 1–2–1–3–5–8–5–9–5–3–6–3–1–4–7–10–7–11–7–4–1, т. е. каждое ребро проходится ровно два раза.

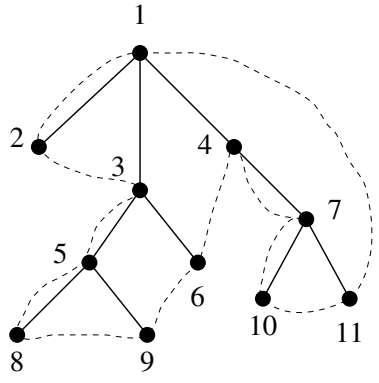


Рис. 9.11. Обход в глубину остовного дерева

Этот маршрут проходит по каждому ребру минимального остовного дерева дважды и, следовательно, сто́ит, самое большее, вдвое дороже, чем оптимальный маршрут.

Но вершины на этом маршруте обхода в глубину будут повторяться. Чтобы удалить лишние вершины, на каждом шаге можно выбирать кратчайший путь к следующей непосещенной вершине. Кратчайший маршрут для дерева на рис. 9.11 проходит через вершины 1–2–3–5–8–9–6–4–7–10–11–1. Так как мы заменили последовательность ребер одним направленным ребром, то аксиома треугольника обеспечивает, что маршрут может стать только короче. Таким образом, этот кратчайший маршрут также находится в пределах допустимого веса и стоит вдвое дороже оптимального маршрута. Что еще лучше, для решения евклидовой версии задачи коммивояжера существуют более сложные алгоритмы, которые рассматриваются в *разделе 16.4*. Среди аппроксимирующих алгоритмов для решения задачи коммивояжера не известен ни один, который не удовлетворяет аксиому треугольника.

9.10.3. Максимальный бесконтурный подграф

Бесконтурные орграфы легче поддаются обработке, чем общие орграфы. Иногда будет полезным упростить данный граф, удалив небольшой набор ребер или вершин, достаточный для разрыва контуров (циклов). Такие задачи о *разрывающем множестве* дуг (feedback set) обсуждаются в *разделе 16.11*.

Здесь мы рассмотрим интересную задачу из этого класса, в которой при разрыве всех ориентированных контуров нужно оставить как можно больше ребер. Постановка задачи выглядит таким образом:

ЗАДАЧА. Найти максимальный бесконтурный ориентированный подграф.

Вход. Ориентированный граф $G = (V, E)$.

Выход. Самое большое подмножество $E' \in E$, для которого $G' = (V, E')$ не содержит контуров.

Существует очень простой алгоритм, который гарантирует решение с количеством ребер равным, по крайней мере, половине оптимального. Я рекомендую вам попробовать разработать такой алгоритм самостоятельно, прежде чем вы прочитаете его описание.

Итак, создаем *любую* перестановку вершин и рассматриваем ее как упорядочивание слева направо, аналогично топологической сортировке. Теперь некоторые ребра будут направлены слева направо, в то время как оставшиеся ребра направлены справа налево.

Размер одного из этих подмножеств ребер должен быть, по крайней мере, таким же, как и другого. Это означает, что оно содержит хотя бы половину ребер. Кроме того, каждое из этих двух подмножеств ребер должно быть бесконтурным, т. к. только бесконтурные орграфы можно подвергать топологической сортировке — контур (цикл) нельзя создать, постоянно двигаясь в одном направлении. Таким образом, подмножество ребер большего размера должно быть бесконтурным и содержать, по крайней мере, половину ребер оптимального решения!

Данный аппроксимирующий алгоритм чрезвычайно прост. Но обратите внимание, что на практике применение эвристики может улучшить его производительность, при этом сохраняя гарантию качества результата. Возможно, имеет смысл попробовать несколько произвольных перестановок и выбрать ту, которая дает наилучший результат. Кроме того, можно попытаться обменивать местами пары вершин в перестановках и оставлять те варианты обмена, которые добавляют большее количество ребер в подмножество большего размера.

9.10.4. Задача о покрытии множества

Предыдущие разделы могут создать ложное представление, что с помощью аппроксимирующих алгоритмов можно получить решение любой задачи с коэффициентом отличия от оптимального равным двум. Однако решение некоторых задач из каталога, например задачи поиска максимальной клики, нельзя аппроксимировать с наперед заданным коэффициентом.

Задача о покрытии множества занимает промежуточную позицию между этими двумя крайностями, поскольку имеет алгоритм, выдающий решение с коэффициентом отличия от оптимального, равным $\Theta(\lg n)$. Задача о покрытии множества представляет собой общий случай задачи о вершинном покрытии. Формальная постановка задачи (рассматриваемой в *разделе 18.1*) выглядит таким образом:

ЗАДАЧА. Найти покрытие множества.

Вход. Семейство подмножеств $S = \{S_1, ..., S_m\}$ универсального множества $U = \{1, ..., n\}$.

Выход. Подмножество наименьшей мощности T семейства S, чье объединение равно универсальному множеству, т. е. $\cup_{i=1}^{|T|} T_i = U$.

Будет естественным применить для решения этой задачи эвристический алгоритм "жадного" типа. В частности, постоянно выбираем подмножество, которое покрывает наибольшее подсемейство непокрытых на данном этапе элементов, пока не получим полное покрытие. Псевдокод соответствующего алгоритма показан в листинге 9.9.

Листинг 9.9. Аппроксимирующий алгоритм поиска покрытия множества

```
SetCover(S)
  While (U ≠ 0) do:
    Определяем подмножество Si, имеющее наибольшее пересечение
    с множеством U
    Выбираем Si для покрытия множества
    U = U – Si
```

Одним из побочных эффектов этого процесса выбора является тот факт, что по мере исполнения алгоритма количество покрытых на каждом шаге элементов образует невозрастающую последовательность. Почему? Потому что в противном случае "жадный" алгоритм выбрал бы более мощное подмножество раньше, если бы оно действительно существовало.

Таким образом, данный эвристический алгоритм можно рассматривать как уменьшение количества непокрытых элементов от n до нуля, причем невозрастающими порциями. Пример трассировки исполнения такого алгоритма показан в табл. 9.1.

Таблица 9.1. Работа "жадного" алгоритма на экземпляре задачи о покрытии множества

Ключевое событие	6	5	4	3	2	1	0
Непокрытые элементы	64	51 40	30 25 22 19 16	13 10 7	4	2	1
Размер выбранного подмножества	13	11 10	5 3 3 3 3	3 3 3	2	1	1

Важное контрольное событие этой трассировки происходит, как только количество оставшихся непокрытых элементов уменьшается в 2^n раз. Очевидно, что таких событий может быть, самое большее, $\lceil \lg n \rceil$.

Пусть w_i обозначает количество подмножеств, выбранных нашим эвристическим алгоритмом для покрытия элементов между контрольными событиями, соответствующими уменьшению в $2^{i+1} - 1$ раз и 2^i раз. Определим максимальную ширину w_i как w, где $0 \le i \le \lg_2 n$. В примере трассировки в табл. 9.1 максимальная ширина представлена пятью подмножествами, необходимыми для перехода от $2^5 - 1$ к 2^4.

Так как существует, самое большее, $\lg n$ таких контрольных событий, то выдаваемое "жадным" эвристическим алгоритмом решение должно содержать, самое большее, $w \lg n$ подмножеств. Но я утверждаю, что оптимальное решение должно содержать, по крайней мере, w подмножеств, так что выдаваемое эвристическим алгоритмом решение хуже оптимального не больше, чем в $\lg n$ раз.

Почему? Рассмотрим среднее количество новых элементов, покрываемых при проходе между контрольными событиями, соответствующими уменьшению в $2^{i+1}-1$ раз и 2^i раз. Для этих 2^i элементов требуется w_i подмножеств, поэтому мощность среднего покрытия равна $\mu_i = 2^i/w_i$. Более того, последнее (наименьшее) из этих подмножеств покрывает, самое большее, μ_i подмножеств. Таким образом, множество S не содержит подмножества, которое может покрыть больше, чем μ_i из оставшихся 2^i элементов. Поэтому, чтобы получить решение, нам нужно, по крайней мере, $2^i/\mu_i = w_i$ подмножеств.

Несколько неожиданным является то обстоятельство, что на самом деле существуют экземпляры покрытия множества, для которых время исполнения этого эвристического алгоритма в $\Omega(\lg n)$ раз больше оптимального. Этот логарифмический множитель является свойством задачи и эвристического алгоритма для ее решения, а не следствием плохо проведенного анализа.

Подведение итогов

Аппроксимирующие алгоритмы гарантируют решения, которые всегда близки к оптимальному. Они могут предоставить практический подход к решению NP-полных задач.

Замечания к главе

Понятие NP-полноты было впервые сформулировано Стивеном Куком (Stephen Cook) в 1971 г. (см. [Coo71]). Задача выполнимости действительно является задачей на один миллион долларов, т. к. институт Clay Mathematical Institute предлагает премию такого размера любому, кто решит вопрос P = NP. Подробности о задаче и призе за ее решение см. по адресу **http://www.claymath.org/millennium/P_vs_NP/**.

В 1972 г. Ричард Карп (Richard Karp) продемонстрировал значимость работы Кука, предоставив сведение от задачи выполнимости к каждой из более чем 20 важных алгоритмических задач. Я рекомендую вам ознакомиться с докладом Карпа ([Kar72]) в силу его исключительного изящества и краткости — он излагает каждое сведение в трех строчках описания, показывающего эквивалентность задач. Работы Кука и Карпа, взятые вместе, предоставляют инструменты для разрешения вопроса сложности буквально сотен важных задач, для решения которых не было известно эффективных алгоритмов.

Наилучшим введением в теорию NP-полноты остается книга "Computers and Intractability" ([GJ79]). В ней дается введение в общую теорию, включая доступное доказательство теоремы Кука ([Coo71]) о том, что сложность задачи выполнимости такая же, как и любой другой задачи класса NP. Книга также содержит важный справочный каталог свыше 300 NP-полных задач, что является хорошим материалом для изучения известных фактов о наиболее интересных задачах. Задачи сведения, упомянутые, но не рассмотренные в этой главе, можно найти в книгах [GJ79] и [CLRS01].

Несколько задач из каталога находятся в состоянии неопределенности, т. е. неизвестно, существует ли эффективный алгоритм для решения конкретной задачи или же она является NP-полной. Наиболее значительными из этих задач являются задачи изоморфизма графов (см. *раздел 16.9*) и разложения целых чисел на множители (см. *раздел 13.8*). Краткость этого списка задач с неопределенным статусом в большой степени является следствием современного уровня развития дисциплины разработки алгорит-

мов. Почти для каждой важной задачи у нас имеется либо эффективный алгоритм, либо веская причина его отсутствия.

Задача об односвязном подграфе, которая рассматривалась в *разделе 9.8*, была впервые доказана в техническом докладе [Mah76].

9.11. Упражнения

Преобразования и выполнимость

1. [2] Предоставьте формулу 3-SAT, которая получается в результате применения сведения задачи SAT к задаче 3-SAT для следующей формулы:

   $(x + y + \overline{z} + w + u + \overline{v}) \cdot (\overline{x} + \overline{y} + z + \overline{w} + u + v) \cdot (x + \overline{y} + \overline{z} + w + u + \overline{v}) \cdot (x + \overline{y})$.

2. [3] Нарисуйте граф, который получается в результате сведения задачи 3-SAT к задаче о вершинном покрытии для выражения

   $$(x + \overline{y} + z) \cdot (\overline{x} + y + \overline{z}) \cdot (\overline{x} + y + z) \cdot (x + \overline{y} + \overline{x}).$$

3. [4] Допустим, что у нас имеется процедура, которая может определить разрешимость задачи коммивояжера из *раздела 9.1.2* за линейное время. Предоставьте эффективный алгоритм поиска самого маршрута, вызывая эту процедуру полиномиальное число раз.

4. [7] Реализуйте процедуру преобразования экземпляров задач выполнимости в эквивалентные экземпляры задачи 3-SAT.

5. [7] Разработайте и реализуйте алгоритм поиска с возвратом для проверки набора формул на выполнимость. Какие критерии можно использовать для отсечения пространства поиска?

6. [8] Реализуйте процедуру сведения задачи о вершинном покрытии к задаче выполнимости и проверьте получившиеся дизъюнкции программой проверки на выполнимость. Насколько такой способ применим на практике для выполнения подобных вычислений?

Базовые сведения

7. [4] Дано: множество X из n элементов, семейство F подмножеств множества X и целое число k. Существует ли k подмножеств семейства F, объединение которых равно X?

   Например, если $X = \{1,2,3,4\}$ и $F = \{\{1,2\},\{2,3\},\{4\},\{2,4\}\}$, для $k = 2$ решения нет, но для $k = 3$ есть (например, $\{1,2\}$, $\{2,3\}$, $\{4\}$). Докажите, что задача о покрытии множества является NP-полной, выполнив сведение с задачи вершинного покрытия.

8. [4] *Задача коллекционера бейсбольных карточек* выглядит таким образом. Имеются пакеты бейсбольных карточек P_1, ..., P_m, каждый из которых содержит подмножество карточек, выпущенных в указанном году. Возможно ли собрать все карточки за этот год, купив не более чем k таких пакетов?

   Например, для игроков {*Aaron*, *Mays*, *Ruth*, *Skiena*} и пакетов

   {{*Aaron*, *Mays*}, {*Mays*, *Ruth*}, {*Skiena*}, {*Mays*, *Skiena*}},

   для $k = 2$ решения нет, но для $k = 3$ есть, такое как

   {*Aaron*, *Mays*}, {*Mays*, *Ruth*}, {*Skiena*}

Докажите, что задача коллекционера бейсбольных карточек является NP-полной, используя для этого сведение к ней от задачи о вершинном покрытии.

9. [4] Приведем формулировку задачи *остовного дерева низкой степени* (low-degree spanning tree problem). Дано: граф G и целое число k. Содержит ли граф G такое остовное дерево, степень всех вершин которого равна, самое большее, k? (Очевидно, что степень определяется только по количеству ребер.) Например, граф, представленный на рис. 9.12, не содержит остовного дерева, степень всех вершин которого меньше чем три.

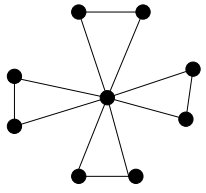


Рис. 9.12. Пример графа

а) Докажите, что задача остовного дерева низкой степени является NP-полной, используя сведение от задачи о гамильтоновом *пути*.

б) Теперь сформулируем *задачу остовного дерева высокой степени* (high-degree spanning tree problem). Дано: граф G и целое число k. Содержит ли граф G остовное дерево, у которого максимальная степень вершины равна, как минимум, k? В предыдущем примере обсуждалось остовное дерево с наивысшей степенью, равной 8. Предоставьте эффективный алгоритм для решения задачи остовного дерева высокой степени и выполните анализ его временной сложности.

10. [4] Докажите, что следующая задача является NP-полной:

ЗАДАЧА. Найти плотный подграф.

Вход. Граф G и два целых числа k и y.

Выход. Содержит ли граф G подграф, имеющий ровно k вершин и, по крайней мере, y ребер?

11. [4] Докажите, что следующая задача является NP-полной:

ЗАДАЧА. Найти клику.

Вход. Неориентированный граф $G = (V, E)$ и целое число k.

Выход. Содержит ли граф G как клику, так и независимое множество размером k?

12. [5] *Эйлеровым циклом* (Eulerian cycle) называется маршрут, который проходит по каждому ребру графа ровно один раз. *Эйлеровым подграфом* (Eulerian subgraph) называется подмножество ребер и вершин графа, которое содержит эйлеров цикл. Докажите, что задача определения количества ребер в наибольшем эйлеровом подграфе графа является NP-сложной. (Подсказка: задача о гамильтоновом цикле является NP-сложной, даже если каждой вершине графа инцидентны три ребра.)

Нестандартные сведения

13. [5] Докажите, что следующая задача является NP-полной:

ЗАДАЧА. Найти минимальное множество представителей (hitting set).

Вход. Коллекция C подмножеств множества S, положительное число k.

Выход. Содержит ли множество S такое подмножество S', для которого $|S'| \leq k$ и каждое подмножество в коллекции C содержит, по крайней мере, один элемент из подмножества S'?

14. [5] Докажите, что следующая задача является NP-полной:

    ЗАДАЧА. Задача о рюкзаке.

    Вход. Множество S, состоящее из n элементов, в котором i-й элемент имеет ценность v_i и вес w_i. Два положительных целых числа: максимальный вес W и требование к ценности груза V.

    Выход. Существует ли такое подмножество $S' \in S$, для которого $\sum_{i \in S'} w_i \leq W$ и $\sum_{i \in S'} v_i \geq V$? (Подсказка: сначала рассмотрите задачу разделения множества целых чисел.)

15. [5] Докажите, что следующая задача является NP-полной:

    ЗАДАЧА. Найти гамильтонов путь.

    Вход. Граф G и вершины s и t.

    Выход. Содержит ли граф G путь с началом в вершине s и концом в вершине t, который проходит через все вершины не более одного раза? (Подсказка: начните с задачи о гамильтоновом цикле.)

16. [5] Докажите, что следующая задача является NP-полной:

    ЗАДАЧА. Найти самый длинный путь.

    Вход. Граф G и положительное целое число k.

    Выход. Содержит ли граф G путь, который соединяет, по крайней мере, k разных вершин, не проходя через каждую из них более одного раза?

17. [6] Докажите, что следующая задача является NP-полной:

    ЗАДАЧА. Найти доминирующее множество.

    Вход. Граф $G = (V, E)$ и положительное целое число k.

    Выход. Существует ли такое подмножество $V' \in V$, где $|V'| \leq k$, в котором для каждой вершины $x \in V$ или $x \in V'$ или существует ребро (x, y), где $y \in V'$.

18. [7] Докажите, что задача о вершинном покрытии (существует ли такое подмножество S, состоящее из k вершин графа G, в котором каждое ребро в графе G инцидентно, по крайней мере, одной вершине в S?) остается NP-полной, даже если степень всех вершин в графе может быть только четной?

19. [7] Докажите, что следующая задача является NP-полной:

    ЗАДАЧА. Задача упаковки множества (set packing problem).

    Вход. Коллекция C подмножеств множества S, положительное число k.

    Выход. Содержит ли множество S, по крайней мере, k непересекающихся подмножеств, т. е. подмножеств, не имеющих общих элементов?

20. [7] Докажите, что следующая задача является NP-полной:

    ЗАДАЧА. Найти разрывающее множество вершин.

    Вход. Ориентированный граф $G = (V, A)$ и положительное целое число k.

Выход. Существует ли такое подмножество $V' \in V$, где $|V'| \leq k$, удаление вершин которого из графа G оставляет бесконтурный орграф?

Алгоритмы для решения частных случаев задач

21. [5] Гамильтонов путь P проходит через каждую вершину ровно один раз. Задача поиска гамильтонова пути в графе G является NP-полной. В отличие от гамильтонова цикла, гамильтонов путь не обязан содержать ребро, соединяющее начальную и конечную вершины пути P.

    Предоставьте алгоритм с временем исполнения $O(n + m)$ для проверки бесконтурного орграфа G на наличие в нем гамильтонова пути. (Подсказка: ваши рассуждения должны идти в направлении топологической сортировки и обхода в глубину.)

22. [8] В задаче 2-SAT требуется выяснить, является ли выполнимой данная булева формула в 2-конъюктивной нормальной форме (КНФ). Задача 2-SAT подобна задаче 3-SAT, только каждая дизъюнкция может содержать всего лишь два литерала. Например, следующая формула записана в 2-КНФ:

    $(x_1 \vee x_2) \wedge (\overline{x}_2 \vee x_3) \wedge (x_1 \vee \overline{x}_3)$

    Предоставьте алгоритм с полиномиальным временем исполнения для решения задачи 2-SAT.

P или NP?

23. [4] Покажите, что следующие задачи принадлежат классу NP:

    - Содержит ли граф G простой путь (т. е. путь, в котором вершины не повторяются) длиной k?
    - Является ли целое число n составным (т. е. не простым)?
    - Содержит ли граф G вершинное покрытие размером k?

24. [7] В течение длительного времени оставался открытым вопрос, возможно ли за полиномиальное время найти решение такой задачи разрешимости: "Является ли целое число n составным, т. е. не простым?" Почему следующий алгоритм не доказывает, что эта задача принадлежит классу P, хотя он и исполняется за время $O(n)$?

```
PrimalityTesting(n)
   composite := false
   for i := 2 to n - 1 do
      if (n mod i) = 0 then
         composite := true
```

Аппроксимирующие алгоритмы

25. [4] В задаче максимальной выполнимости требуется найти набор значений истинности, который является выполняющим для максимального количества дизъюнкций. Предоставьте эвристический алгоритм, выполняющий, как минимум, вдвое меньше дизъюнкций, чем выполняет оптимальное решение.

26. [5] Имеется следующий эвристический алгоритм поиска вершинного покрытия: создается дерево обхода в глубину графа, из которого удаляются все листья; оставшиеся узлы

должны составлять вершинное покрытие графа. Докажите, что размер этого вершинного покрытия превышает оптимальный, самое большее, в два раза.

27. [5] В задаче максимального разреза графа $G = (V, E)$ требуется разделить множество вершин V на два непересекающихся подмножества A и B таким образом, чтобы максимизировать количество ребер $(a, b) \in E$, где $a \in A$, $b \in B$. Рассмотрим следующий эвристический алгоритм поиска максимального разреза. Сначала помещаем вершину v_1 в подмножество A, а вершину v_2 — в подмножество B. Каждую оставшуюся вершину помещаем в подмножество, которое добавляет в разрез наибольшее количество ребер. Докажите, что размер данного разреза равен, по крайней мере, половине оптимального разреза.

28. [5] В задаче об упаковке в контейнеры дается n элементов, веса которых равны w_1, w_2, ..., w_n. Нужно найти наименьшее количество контейнеров, в которые можно упаковать эти n элементов, при этом емкость каждого контейнера не превышает один килограмм.

    В эвристическом алгоритме "первый подходящий" (first-fit) объекты рассматриваются в порядке, в котором они представляются. Каждый объект укладывается в первый же контейнер, в который он помещается. Если такого контейнера нет, объект помещается в новый (пустой) контейнер. Докажите, что количество контейнеров, выдаваемое этим эвристическим алгоритмом, превышает оптимальное не более, чем в два раза.

29. [5] Для только что описанного эвристического алгоритма "первый подходящий" приведите пример экземпляра задачи, решение которого дает количество контейнеров, превышающее оптимальное максимум в 5/3 раз.

30. [5] Целью задачи раскраски вершин графа $G = (V, E)$ является назначение цветов вершинам множества V таким образом, чтобы вершины на концах каждого ребра были окрашены в разные цвета. Предоставьте алгоритм для раскраски графа G не более чем $\Delta + 1$ разными красками, где Δ представляет максимальную степень вершин графа G.

Задачи по программированию

Эти задачи доступны на сайтах **http://www.programming-challenges.comcom** и **http://uva.onlinejudge.org**.

Геометрия

1. The Monocycle. 111202/10047.
2. Dog and Gopher. 10310/111301.
3. Chocolate Chip Cookies. 111304/10136.
4. Birthday Cake. 111305/10167.

Вычислительная геометрия

5. Closest Pair Problem. 111402/10245.
6. Chainsaw Massacre. 111403/10043.
7. Hotter Colder. 111404/10084.
8. Useless Tile Packers. 111405/10065.

> ***Примечание***
>
> Эти задания не имеют непосредственного отношения к теме NP-полноты, но добавлены для полноты картины.

ГЛАВА 10

Как разрабатывать алгоритмы

Разработка правильного алгоритма для определенного приложения в значительной степени является творческой деятельностью. Мы получаем задачу и придумываем решение для нее. Пространство выбора при разработке алгоритмов огромно, и у вас есть много возможностей допустить ошибку.

Цель этой книги состоит в том, чтобы повысить вашу квалификацию разработчика алгоритмов. Представленные в первой части этой книги методы содержат идеи, лежащие в основе всех комбинаторных алгоритмов. А каталог задач во второй части поможет вам моделировать ваши приложения, предоставляя информацию о соответствующих задачах. Однако успешному разработчику алгоритмов требуется нечто большее, чем книжные знания. Для этого нужен определенный склад ума — правильный подход к решению задач. Этому трудно научиться по книгам, но обладание таким качеством является необходимым условием.

Ключом к разработке алгоритмов (или любой другой деятельности, связанной с решением задач) является умение задавать самому себе вопросы, чтобы направлять свой процесс мышления: "Что будет, если я попробую такой способ? А если теперь попробовать другой?" Когда на каком-либо этапе вы столкнетесь с затруднениями, лучше всего перейти к следующему вопросу. Самым полезным участником любой мозговой атаки является тот, кто постоянно задает вопрос: "А почему нельзя применить такой-то способ?", а не тот, кто дает ответ на этот вопрос. В конце концов, первый участник предложит подход, который второй не сможет забраковать.

С этой целью мы предоставляем последовательность вопросов, которая должна направлять вашу деятельность в поиске правильного алгоритма для решения стоящей перед вами задачи. Но чтобы эти вопросы были эффективными, мало их задавать, нужно отвечать на них. Важно тщательно прорабатывать вопросы, записывая их в журнал. Правильным ответом на вопрос: "Можно ли сделать это таким способом?" является не просто: "Нет", а "Нет, потому что...". Четко излагая свои мысли по поводу отрицательного результата, вы сможете проверить, не упустили ли вы вариант, о котором просто не подумали. Поразительно, как часто причиной отсутствия у вас правильного объяснения является ошибочное умозаключение.

В процессе разработки важно осознавать разницу между стратегией и тактикой и постоянно помнить о ней. Стратегия — это поиск общей картины, основы, на которой строится решение. А тактика используется для решения второстепенных вопросов на пути к глобальной цели. В процессе решения задач важно постоянно проверять, на правильном ли уровне находятся ваши рассуждения. Если у вас нет глобального плана (т. е. стратегии) для решения поставленной перед вами задачи, то нет никакого смысла размышлять о правильной тактике. Примером стратегического вопроса явля-

ется: "Могу ли я смоделировать данное приложение в виде задачи создания алгоритма на графах?" А тактическим вопросом может быть: "Какую структуру данных следует использовать для представления моего графа, список смежности или матрицу смежности?" Конечно же, тактические решения критичны для качества конечного решения, но их можно должным образом оценить только в свете успешной стратегии.

У слишком многих людей при необходимости решить задачу пропадают все мысли. Прочитав задачу, они приступают к ее решению и вдруг осознают, что не знают, что делать дальше. Чтобы не попадать в такие ситуации, следуйте списку вопросов, изложенных далее в этой главе и в большинстве разделов каталога задач. Они подскажут вам, что делать дальше.

Очевидно, что чем больше у вас имеется опыта по использованию методов разработки алгоритмов, таких как динамическое программирование, алгоритмы на графах и структуры данных, тем с большим успехом сможете вы следовать этим вопросам. Первая часть этой книги предназначена для развития и укрепления этих технических основ. Но независимо от уровня вашей технической подготовки будет полезно следовать этим вопросам. Самые первые и наиболее важные вопросы в этом списке помогут вам добиться глубокого понимания решаемой задачи и не потребуют никакого специального опыта.

Сам список вопросов был создан под влиянием отрывка из замечательной книги о программе космических исследований США, называющейся "The Right Stuff" ("Правильный материал") ([Wol79]). В данном отрывке описывались радиопереговоры пилотов-испытателей непосредственно перед крушением самолета. Можно было бы ожидать, что в данной ситуации они впадали в панику, но вместо этого пилоты выполняли действия из списка для нештатных ситуаций: "Проверил закрылки. Проверил двигатель. Крылья в порядке. Сбросил..." Эти пилоты обладали "Правильным материалом" и благодаря этому иногда им удавалось избежать гибели.

Я надеюсь, что эта книга и список вопросов помогут вам обрести "Правильный материал" разработчика алгоритмов. Я также надеюсь, что это поможет вам избежать крушений в вашей деятельности.

Список вопросов разработчика алгоритмов

1. Понимаю ли я задачу?
   - Из чего состоит вход?
   - Какой именно требуется результат?
   - Могу ли я создать достаточно небольшой входной экземпляр, чтобы его можно было решить вручную? Что будет, если я попытаюсь решить его?
   - Насколько важно для моего приложения всегда получать оптимальный результат? Устроит ли меня результат, близкий к оптимальному?
   - Каков размер типичного экземпляра решаемой задачи? 10 элементов? 1 000 элементов? 1 000 000 элементов?
   - Насколько важной является скорость работы приложения? Задачу нужно решить за одну секунду, одну минуту, один час или за один день?

- Сколько времени и усилий могу я вложить в реализацию? Буду ли я ограничен простыми алгоритмами, которые можно реализовать в коде за один день, или у меня будет возможность поэкспериментировать с разными подходами и посмотреть, какой из них лучше?
- Задачу какого типа я пытаюсь решить? Численную задачу, задачу теории графов, геометрическую задачу, задачу со строками или задачу на множестве? Какая формулировка выглядит самой легкой?

2. Могу ли я найти простой детерминированный или эвристический алгоритм для решения стоящей передо мной задачи?
   - Можно ли *правильно* решить задачу методом полного перебора всех возможных решений?
     - Если можно, то почему я уверен, что этот алгоритм всегда выдает правильное решение?
     - Как оценить качество созданного решения?
     - Каким будет время исполнения этого простого, но медленного алгоритма — полиномиальным или экспоненциальным? Настолько ли проста моя задача, чтобы ее можно было решить методом полного перебора?
     - Уверен ли я, что моя задача достаточно четко определена, чтобы правильное решение было *возможно*?
   - Можно ли решить данную задачу, постоянно применяя какое-либо простое правило, например, выбирая в первую очередь наибольший элемент, наименьший элемент или произвольный элемент?
     - Если можно, то при каком входе этот эвристический алгоритм работает хорошо? Соответствует ли вход такого типа данным, актуальным для моего приложения?
     - При каком входе этот эвристический алгоритм работает плохо? Если не удастся найти примеры плохих входных экземпляров, то могу ли я показать, что алгоритм всегда работает хорошо?
     - Сколько времени требуется моему эвристическому алгоритму, чтобы выдать ответ? Можно ли его реализовать простым способом?

3. Есть ли моя задача в каталоге задач этой книги?
   - Что известно о данной задаче? Существует ли реализация решения, которую я могу использовать?
   - Искал ли я свою задачу в правильном месте каталога? Просмотрел ли я все рисунки? Проверил ли я все возможные ключевые слова в алфавитном указателе?
   - Существуют ли в Интернете ресурсы, имеющие отношение к моей задаче? Искал ли я в Google и Google Scholar? Посетил ли я веб-сайт этой книги, **http://www.cs.sunysb.edu/~algorith**?

4. Существуют ли специальные случаи стоящей передо мной задачи, которые я знаю, как решить?
   - Можно ли эффективно решить задачу, игнорируя некоторые входные параметры?
   - Станет ли задача проще, если присвоить некоторым входным параметрам тривиальные значения, такие как 0 или 1?
   - Сумею ли я упростить задачу до такой степени, что *смогу* решить ее эффективно?
   - Почему алгоритм для решения этого специального случая нельзя обобщить для более широкого диапазона входных экземпляров?
   - Не является ли моя задача специальным случаем одной из общих задач каталога?
5. Какая из стандартных парадигм разработки алгоритмов наиболее соответствует моей задаче?
   - Существует ли набор элементов, который можно отсортировать по размеру или какому-либо ключу? Способствует ли отсортированный набор данных более легкому решению задачи?
   - Можно ли разбить задачу на две меньших задачи, например, посредством двоичного поиска? Поможет ли метод разделения элементов на большие и маленькие или правые и левые? Может быть, стоит попробовать алгоритм типа "разделяй и властвуй"?
   - Не обладают ли входные объекты или требуемое решение естественным порядком слева направо, как, например, символы в строке, элементы перестановки или листья дерева? Могу ли я применить динамическое программирование, чтобы воспользоваться этим упорядочиванием?
   - Нет ли повторяющихся операций, таких как поиск наименьшего/наибольшего элемента? Могу ли я применить специальные структуры данных для ускорения этих операций, например словари/таблицы хэширования или пирамиды/очереди с приоритетами?
   - Могу ли я применить случайную выборку следующего объекта? Могу ли я создать несколько произвольных конфигураций и выбрать наилучшую из них? Могу ли я применить какой-либо вид управляемой случайности, вроде метода имитации отжига, чтобы сосредоточить усилия на наилучшем решении?
   - Могу ли я сформулировать свою задачу, как линейную программу или как целочисленную программу?
   - Имеет ли моя задача какое-либо сходство с задачей выполнимости, задачей коммивояжера или какой-либо другой NP-полной задачей? Может ли задача быть NP-полной и вследствие этого не иметь эффективного алгоритма решения? Есть ли эта задача в списке из приложения к книге [GJ79]?

6. Я все еще не знаю, что делать?
   - Могу и хочу ли я потратить деньги, чтобы нанять специалиста, чтобы он пояснил мне, что делать? Если можете, то ознакомьтесь со списком профессиональных консалтинговых услуг в *разделе 19.4*.
   - Не пройти ли заново по списку вопросов? Прежние ли ответы даю я при новом проходе по списку?

Решение задач не является точной наукой. Отчасти это искусство, а отчасти — ремесло. Навык решения задач — это один из полезнейших навыков, и его стоит приобрести. Моей любимой книгой по решению задач является книга "How to Solve It" ("Как решить задачу") [P57], содержащая каталог методов решения задач. Я обожаю его просматривать, как с целью решения конкретной задачи, так и просто ради удовольствия.

ЧАСТЬ II

Каталог алгоритмических задач

ГЛАВА 11

Описание каталога

Эта часть книги содержит каталог алгоритмических задач, которые часто возникают на практике. Здесь приводится общая информация по каждой задаче и даются советы, как действовать, если та или иная задача возникнет в вашем приложении.

Как использовать этот каталог? Если вы знаете название вашей задачи, то с помощью алфавитного указателя или оглавления найдите ее описание. Прочитайте весь материал, относящийся к задаче, поскольку он содержит ссылки на связанные с ней задачи. Пролистайте каталог, изучая рисунки и названия задач. Возможно, что-то покажется подходящим для вашего случая. Смело пользуйтесь алфавитным указателем — каждая задача внесена в него несколько раз под разными ключевыми словами. Если вы все еще не можете найти ничего подходящего, значит, либо ваша задача не годится для решения комбинаторными алгоритмами, либо вы не полностью понимаете ее. В любом случае возвращайтесь назад к исходному этапу поиска решения.

Каталог содержит обширную информацию разных типов, которая никогда раньше не собиралась в одном месте.

Каждая задача сопровождается рисунком, представляющим входной экземпляр задачи и результат его решения. Эти стилизованные примеры иллюстрируют задачи лучше, чем формальные описания. Так, пример, касающийся минимального остовного дерева, демонстрирует кластеризацию точек с помощью минимальных остовных деревьев. Мы надеемся, что читатели нашей книги после беглого просмотра рисунков смогут найти то, что им нужно.

Если вы нашли подходящую для вашего приложения задачу, прочитайте *раздел "Обсуждение"*. В нем описаны ситуации, в которых может возникнуть данная задача, и особые случаи входных данных. Также рассказано, какой результат можно ожидать и, что еще более важно, как его добиться. Для каждой задачи дано краткое описание несложного алгоритма, которым можно ограничиться в простых случаях, и ссылки на более мощные алгоритмы, чтобы вы могли ими воспользоваться, если первое решение окажется недостаточно эффективным.

Кроме этого, для каждой задачи перечислены существующие программные реализации. Многие из этих процедур весьма удачны, и их код можно вставлять непосредственно в ваше приложение. Другие могут не отвечать требованиям промышленного применения, но я надеюсь, что они послужат хорошей моделью для вашей собственной реализации. Вообще, программные реализации приведены в порядке убывания их ценности, но при наличии бесспорно лучшего варианта мы отмечаем его. В *главе 19* для многих из этих реализаций дается более подробная информация. Практически для всех реализаций на сайте этой книги (**http://www.cs.sunysb.edu/~algorith**) даются ссылки для их загрузки.

Наконец, в примечаниях к задаче рассказывается ее история и приводятся результаты, представляющие, в основном, теоретический интерес. Мы постарались привести наилучшие известные результаты для каждой задачи и дать ссылки на работы по теоретическому и эмпирическому сравнению алгоритмов, если таковые имеются. Эта информация должна представлять интерес не только для студентов и исследователей, но также и для практиков, не удовлетворившихся рекомендуемыми решениями.

Предостережения

Данный материал является каталогом алгоритмических задач, а не сборником рецептов, поскольку существует слишком много различных задач и вариантов их решений. Моя цель — указать вам правильное направление, чтобы вы могли решать свои задачи самостоятельно. Я попытался очертить круг вопросов, с которыми вы столкнетесь, решая задачи самостоятельно.

- Для каждой задачи я дал рекомендацию, какой алгоритм следует использовать. Эти советы основаны на моем опыте и касаются приложений, которые я считаю типичными. При работе над книгой я полагал, что гораздо важнее дать конкретные рекомендации для типичных случаев, чем пытаться охватить все возможные ситуации. Если вы не согласны с моим советом, поступайте по-своему. Однако вначале постарайтесь понять идеи, лежащие в основе моих советов, и сформулировать причину, по которой ваше приложение не соответствует им.
- Рекомендуемые мною реализации не обязательно являются исчерпывающими решениями вашей задачи. Я предлагаю реализацию, когда считаю, что она может быть полезной в большей степени, чем просто описание алгоритма. Некоторые программы пригодны только в качестве моделей для создания вашего собственного кода. Другие встроены в большие системы и их может быть слишком трудно извлечь и исполнять отдельно. Исходите из предположения, что все решения содержат ошибки. Многие из этих ошибок могут оказаться довольно серьезными, так что будьте начеку.
- Вы обязаны соблюдать условия лицензии любой реализации, которую вы используете в коммерческих целях. Многие из этих программ не являются свободно распространяемыми и имеют лицензионные ограничения. Подробности см. в *разделе 19.1*.
- Мне бы хотелось узнать о результатах ваших действий согласно моим рекомендациям, как о положительных, так и об отрицательных. Особый интерес для меня представляет информация о других реализациях, неизвестных мне, но известных вам. Пишите мне по адресу **feedback@algorist.com**.

ГЛАВА 12

Структуры данных

Структуры данных — это базовые конструкции для построения приложений. Чтобы максимально полно использовать возможности стандартных структур данных, необходимо иметь о них ясное представление. Поэтому мы сначала подробно рассмотрим разные структуры данных, а затем перейдем к обсуждению других задач.

Возможно, что наиболее важным аспектом этого каталога будут предоставленные в нем ссылки на разные программные реализации и библиотеки структур данных. Хорошая реализация многих структур данных — дело далеко не тривиальное, поэтому программы, на которые даются ссылки, будут полезными в качестве моделей, даже если они и не в точности подходят для ваших задач. Некоторые важные структуры данных, такие как kd-деревья и суффиксные деревья, известны не настолько хорошо, насколько они того заслуживают. Будем надеяться, что этот каталог добавит им известности.

Существует большое количество книг по элементарным структурам данных. На мой взгляд, лучшими из них являются следующие:

- [Sed98] — подробное введение в алгоритмы и структуры данных, отличающееся удачными рисунками, показывающими алгоритмы в действии. Имеются версии книги для языков C, C++ и Java;
- [Wei06] — хороший учебник, с упором на структуры данных, в большей степени, чем на алгоритмы. Имеются версии книги для языков Java, C, C++ и Ada;
- [GT05] — версия книги для Java с активным использованием авторской библиотеки структур данных JDSL (Java Data Structures Library);

Книга [MS05] содержит всесторонний обзор современных исследований в области структур данных. Читатель, знакомый лишь с основными понятиями из этой области, будет удивлен объемом проводимых исследований.

12.1. Словарь

Вход. Множество из n записей, каждая из которых идентифицируется одним или несколькими ключевыми полями.

Задача. Создать и поддерживать структуру данных для эффективного поиска, вставки и удаления записи, связанной с ключом запроса q (рис. 12.1).

Обсуждение. Абстрактный тип данных "словарь" является одной из наиболее важных структур в теории вычислительных систем. Для реализации словарей было предложено несколько десятков структур данных, включая хэш-таблицы, списки с пропусками и двоичные деревья поиска. Это означает, что выбрать наилучшую структуру данных не всегда легко. Используемая структура данных может оказать значительное влияние на

производительность приложения. *Однако на практике важнее избежать применения неподходящей структуры данных, чем найти самую лучшую.*

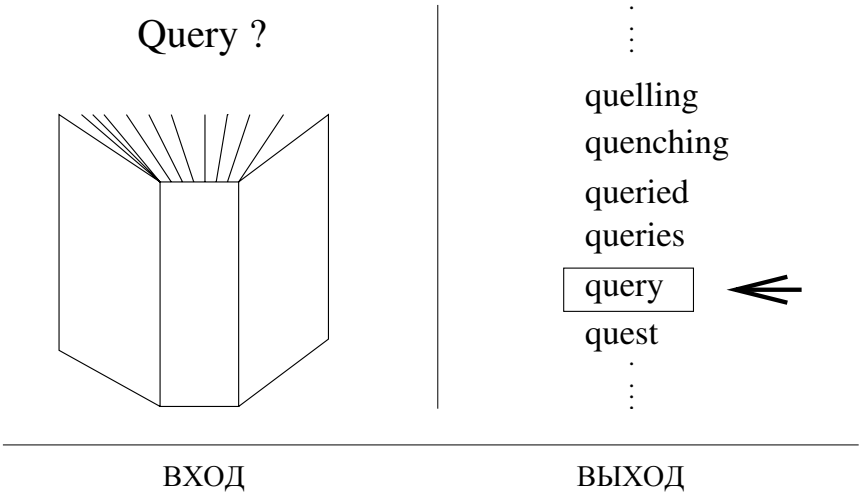


Рис. 11.1. Поиск в словаре

Вы должны тщательно изолировать словарную структуру данных от ее интерфейса. Используйте явные вызовы методов или процедур для инициализации, поиска и модифицировании структуры данных, а не внедряйте их в код структуры. Такой подход не только позволяет получить более аккуратную программу, но также облегчает проведение экспериментов с разными реализациями для оценки их производительности. Не беспокойтесь по поводу неизбежных накладных расходов по вызову процедур. Если время работы вашего приложения является очень важным фактором, то выбор правильной реализации словаря позволит вам минимизировать его.

Выбирая структуру данных для словаря, ответьте на несколько вопросов.

- *Сколько элементов будет содержать структура данных?* Известно ли их количество заранее? Достаточно ли будет для решения вашей задачи простой структуры данных или же размер задачи настолько велик, что следует беспокоиться о нехватке памяти или производительности виртуальной памяти?

- *Известна ли относительная частота операций вставки, удаления и поиска?* Статические структуры данных (например, упорядоченные массивы) достаточны для приложений, в которых структура данных не подвергается изменениям после ее первоначального создания. Полудинамические структуры данных, поддерживающие только операцию вставки и не допускающие удаление данных, реализовать значительно проще, чем полностью динамические.

- *Можно ли предполагать, что обращение к ключам будет единообразным и случайным?* Во многих приложениях поисковые запросы имеют асимметричное распределение обращений, т. е. некоторые элементы пользуются большей популярностью, чем другие. Кроме этого, запросы часто обладают свойством временно́й локальности. Иными словами, запросы периодически приходят группами (кластерами), а не через регулярные интервалы времени. В некоторых структурах данных (таких как

косые деревья) можно воспользоваться асимметричным и кластеризированным пространством.

- *Критична ли скорость отдельных операций или требуется минимизировать только общий объем работы, выполняемый всей программой?* Когда время реакции критично, например, в программе управления аппаратом Искусственного кровообращения, время ожидания выполнения следующего шага не может быть слишком большим. А в программе, выполняющей множество запросов к базе данных, например, для выяснения, кто из нарушителей закона является политическим деятелем, скорость поиска конкретного конгрессмена не является критичной, если удается вы явить их всех с минимальными общим и затратами времени.

Представители современного поколения "объектно-ориентированных" программистов способны написать контейнерный класс не в большей степени, чем отремонтировать двигатель в своей машине. В этом нет ничего плохого — для большинства приложений вполне достаточно стандартных контейнеров. Тем не менее, иногда полезно знать, что находится "под капотом":

- *неупорядоченные списки или массивы*. Для небольших наборов данных самой простой в обслуживании структурой будет неупорядоченный массив. По сравнению с аккуратными и компактными массивам и производительность связных структур может быть крайне неудовлетворительной. Но когда размер словаря превысит 50-100 элементов, линейное время поиска в списке или массиве сведет на нет все его преимущества. Подробности простых реализаций словарей см. в *разделе 3.3*.

  Особенно интересным и полезным вариантом является *самоорганизующийся список* (self-organizing list), в котором каждый вставленный или просмотренный элемент перемещается в начало списка. Таким образом, если в ближайшем будущем к данно му ключу осуществляется повторное обращение, то этот ключ будет находится вблизи начала списка, и его поиск займет очень мало времени. Большинство прило жений демонстрирует как неоднородную частоту обращений, так и пространствен ную локальность, поэтому среднее время успешного поиска в самоорганизующемся списке обычно намного лучше, чем в упорядоченном или неупорядоченном списке. Самоорганизующиеся структуры данных можно создавать из массивов точно так же, как из связных списков и деревьев;

- *упорядоченные списки или массивы*. Обслуживание упорядоченного списка обычно не стоит затрачиваемых усилий, если только вы не пытаетесь избежать создания ду бликатов, т. к. эта структура данных не поддерживает двоичный поиск. Использова ние упорядоченного массива будет уместным только в тех случаях, когда не требу ется выполнять большое количество вставок и/или удалений;

- *хэш-таблицы*. Для приложений, работающих с количеством элементов в диапазоне от умеренного до большого (примерно от 100 до 10 000 000), использование хэш-таблицы будет, скорее всего, правильным выбором. Хэш-функция устанавливает соответствие ключей (будь то строки, числа или что угодно другое) и целых чисел в диапазоне от 0 до m-1. При создании хэш-таблицы сопровождается массив из m корзин, реализованных в виде неупорядоченного связного списка: Хэш-функция немедленно определяет корзину; содержащую данный ключ. Если используемая хэш-

функция распределяет ключи достаточно равномерно и размер хэш-таблицы довольно велик, то каждая корзина будет содержать очень небольшое количество элементов, что делает линейные поиски приемлемыми. Вставка и удаление элемента из хэш-таблицы сводится к вставке и удалению из корзины/списка. Хэширование подробно обсуждалось в *разделе 3.7*.

В большинстве приложений производительность хорошо отлаженной хэш-таблицы будет превосходить производительность упорядоченного массива. Но прежде чем приступить к реализации хэш-таблицы, ответьте на несколько вопросов.

- *Каким образом обрабатывать коллизии?* Использование открытой адресации вместо корзин позволит получить небольшие таблицы с хорошей производительностью, но с повышением коэффициента нагрузки (отношение заполненности к емкости) производительность хэш-таблицы будет понижаться.
- *Каким должен быть размер таблицы?* Если применяются корзины, то значение m должно быть приблизительно равным максимальному количеству элементов, которое вы планируете поместить в таблицу. Если применяется открытая адресация, то значение m должно превышать ожидаемое максимальное количество элементов, по крайней мере, на 30%. Выбор в качестве m простого числа компенсирует возможные недостатки неудачной хэш-функции.
- *Какую хэш-функцию использовать?* Для строк должна работать, например, такая формула $H(S, j) = \sum_{i=0}^{m-1} \alpha^{m-(i+1)} \times char(s_{i+m}) \bmod m$, где α — размер алфавита, $char(x)$ — функция, которая для каждого символа x возвращает его ASCII-код. Для реализации эффективного вычисления этой хэш-функции используйте правило Горнера (или заранее вычисляйте значения α^x), как показано в *разделе 13.9*. Эта хэш-функция имеет интересное свойство, заключающееся в том, что

$$H(S, j+1) = (H(S, j) - \alpha^{m-1} char(s_j))\alpha + s_{j+m}$$

вследствие чего хэш-коды последовательных окон строки размером в m символов можно вычислять за постоянное время, а не за время $O(m)$.

Независимо от того, какую хэш-функцию вы решите использовать, изучите статистические данные относительно распределения ключей по корзинам, чтобы выяснить, насколько она *действительно* является однородной. Вероятно, что самая первая выбранная хэш-функция окажется не самой лучшей. Использование неудачной хэш-функции может заметно снизить производительность любого приложения.

♦ *Двоичные деревья поиска.* Двоичные деревья являются элегантными структурами данных, которые поддерживают быстрое исполнение операций вставки, удаления и запросов. Эти структуры данных рассматриваются в *разделе 3.4*. Разница между типами деревьев проявляется в том, выполняется ли их балансировка явным образом после операций вставки и удаления и в способе ее осуществления. В *произвольных деревьях поиска* узел просто вставляется в некий лист дерева и балансировка не осуществляется. Хотя такие деревья имеют хорошую производительность в случае произвольных вставок, большинство приложений не обеспечивает случайность вставок. В самом деле, несбалансированные деревья поиска, созданные вставками

элементов в отсортированном порядке, никуда не годятся, т. к. их производительность сравнима с производительностью связных списков.

Структура сбалансированных деревьев поиска обновляется с помощью локальных операций *ротации*, которые перемещают удаленные узлы ближе к корню, таким образом поддерживая упорядоченную структуру дерева. В настоящее время такие сбалансированные деревья поиска, как AVL и 2-3, считаются устаревшими, и предпочтение отдается *красно-черным деревьям*. Особенно интересной самоорганизующейся структурой данных являются *косые деревья*. В них ключи, к которым было осуществлено обращение, перемещаются в корень посредством операции ротации. Таким образом, часто используемые или недавно посещенные узлы находятся в верхней части дерева, что позволяет быстрее производить поиск.

Итак, какой тип дерева лучше всего подойдет для вашего приложения? Наверное, тот, для которого у вас имеется наилучшая реализация. Не так важен тип используемого сбалансированного дерева, как профессионализм программиста, реализовавшего его.

- *B-деревья*. Для наборов данных настолько больших, что они не помещаются в оперативную память (скажем, свыше 1 000 000 элементов), наилучшим выбором будет какой-либо тип B-дерева. Когда структуру данных необходимо хранить вне оперативной памяти, время поиска увеличивается на несколько порядков. Подобное падение производительности в меньшем масштабе может наблюдаться в системах с современной архитектурой кэша, т. к. кэш работает намного быстрее, чем оперативная память.

  Идея, лежащая в основе B-дерева, состоит в сворачивании нескольких уровней двоичного дерева поиска в один большой узел, чтобы можно было выполнить операцию, эквивалентную нескольким шагам поиска, прежде чем потребуется новое обращение к диску. С помощью B-деревьев можно получать доступ к огромному количеству ключей, выполняя небольшое количество обращений к диску. Чтобы максимально эффективно использовать B-деревья, нужно понимать, как взаимодействуют внешние запоминающие устройства с виртуальной памятью при учете таких аппаратных характеристик, как размер страницы и виртуальное/реальное адресное пространство. *Нечувствительные к кэшированию алгоритмы* (cache-oblivious algorithms) позволяют уменьшить значение этих факторов.

  Даже в случае наборов данных небольшого размера результатом использования файлов подкачки может быть неожиданно низкая производительность, поэтому необходимо следить за интенсивностью обращений к жесткому диску, чтобы получить дополнительную информацию для принятия решения, стоит ли использовать B-деревья.

- *Списки с пропусками*. Списки с пропусками представляют собой иерархическую структуру упорядоченных связных списков, в которых решение, копировать ли элемент в список более высокого уровня, принимается случайным образом. В структуре задействовано примерно $\lg n$ списков, размер каждого из которых приблизительно вдвое меньше размера списка, расположенного уровнем выше. Поиск начинается в самом коротком списке. Искомый ключ находится в интервале между двумя эле-

ментами и потом ищется в списке большего размера. Каждый интервал поиска содержит ожидаемое постоянное количество элементов в каждом списке, при этом общее ожидаемое время исполнения запроса составляет $O(\lg n)$. Основные достоинства списков с пропусками по сравнению со сбалансированными деревьями — легкость анализа и реализации.

Реализации. Современные языки программирования содержат библиотеки подробных и эффективных реализаций контейнеров. В настоящее время библиотека STL (Standard Template Library) для языка C++ поставляется с большинством компиляторов; Кроме этого эту библиотеку можно загрузить вместе с документацией с веб-сайта **http://www.sgi.com/tech/stl/**. Более подробное руководство по использованию библиотеки STL и стандартной библиотеки C++ можно найти в книгах [Jos99], [Mey01] и [MDS01];

Библиотека LEDA (см. раздел 19.1.1) предоставляет полную коллекцию словарных структур данных, реализованных на языке C++, включая хэш-таблицы, совершенные хэш-таблицы, B-деревья, красно-черные деревья, деревья случайного поиска и списки с пропусками. В результате экспериментов, описанных в книге [MN99], было определено, что наилучшим вариантом для словарей являются хэш-таблицы, а списки с пропусками и (2,4)-деревья (частный случай B-деревьев) являются наиболее эффективными древоподобными структурами.

Небольшая библиотека структур данных Java Collections (JC) входит в стандартный пакет утилит Java java.util (**http://java.sun.com/javase/**). Библиотека Data Structures Library in Java (JDSL) является более обширной. Ее можно загрузить для некоммерческого применения с веб-сайта **http://www.jdsl.org/**. Библиотека JDSL подробно описана в книгах [GTV05] И [GT05].

Примечания

В книге [Knu91] представлено наиболее подробное описание и анализ основных словарных структур данных. Но в ней отсутствуют некоторые современные структуры данных такие как красно-черные и косые деревья. Чтение книг Дональда Кнута послужит хорошим введением в предмет для всех изучающих теорию вычислительных систем.

В книге [MS05] дан всесторонний обзор современных исследований в области структур данных. Другие исследования представлены в книгах [MT90b] и [GBY91]. Хорошими учебниками по словарным структурам данных являются [Sed98], [Wei06] и [GT05]. Мы отсылаем читателя к этим работам, чтобы не приводить здесь ссылки на описания структур данных, обсуждавшихся выше.

Соревнование DIMAGS в 1996 г. было посвящено реализациям элементарных структур данных, включая словари (см. [GJM02]). Наборы данных и код можно загрузить с веб-сайта **http://dimacs.rutgers.edu/Challenges**.

Для многих задач стоимость обмена данными между различными запоминающими устройствами (оперативной памятью, кэшем или диском) превышает стоимость собственно вычислений. В каждой операции по пересылке данных перемещается один блок размером b, которому эффективные алгоритмы стремятся минимизировать количество перемещений блоков. Исследование сложности фундаментальных алгоритмов и структур данных для этой модели внешней памяти представлено в журнале [Vit0l]. *Нечувствительные к кэшированию* структуры данных дают гарантию производительности для такой модели, не требуя при этом явной информации о параметре размера блока b. Таким образом, хорошую производительность можно получить на любой машине, не прибегая к специфическим

ским для данной архитектуры настройкам. Превосходное исследование структур данных, нечувствительных к кэшированию, приведено в [ABF05].

Многие современные структуры данных, такие как косые деревья, изучались с применением *амортизированного анализа*, при котором устанавливается верхняя граница для общего времени исполнения любой последовательности операций. При амортизированном анализе одна операция может быть очень дорогой, но только потому, что ее стоимость компенсируется низкими затратами на другие операции. Структура данных, имеющая амортизированную сложность $O(f(n))$, является менее предпочтительной, чем структура с такой же сложностью в наихудшем случае (т. к. существует вероятность выполнения очень дорогой операции), но она все же лучше структуры с такой сложностью в среднем случае (т. к. амортизированная граница достигнет этого значения при любом входе).

Родственные задачи. Сортировка (см. *раздел 14.1*), поиск (см. *раздел 14.2*).

12.2. Очереди с приоритетами

Вход. Набор записей, ключи которых полностью упорядочены по номерам или каким-либо другим способом.

Задача. Создать и поддерживать структуру данных для обеспечения быстрого доступа к *наименьшему* или *наибольшему* ключу (рис. 12.2).

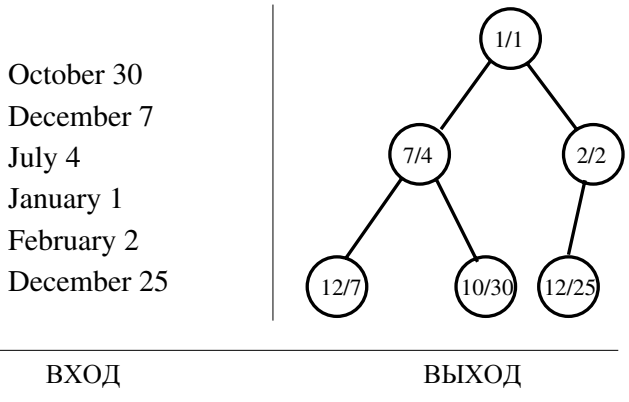


Рис. 12.2. Очередь с приоритетами

Обсуждение. Очереди с приоритетами являются полезными структурами данных при моделировании поведения систем, особенно при сопровождении набора запланированных событий, упорядоченных по времени. Они получили такое название потому, что элементы из них можно извлекать не в зависимости от времени вставки (как в стеке или обычной очереди) и не в результате совпадения ключа (как в словаре), а по их приоритету.

Если после первоначального запроса приложение не будет вставлять новые элементы, то в явно выраженной очереди с приоритетами нет необходимости. Просто отсортируйте записи по приоритетам и выполняйте обработку сверху вниз, поддерживая при этом указатель на последнюю извлеченную запись. Такая ситуация возникает в алгоритме Крускала при построении минимального остовного дерева или при выполнении полностью распланированного сценария развития событий.

Но если операции вставки и удаления чередуются с запросами, понадобится настоящая очередь с приоритетами. Чтобы решить, какой тип очереди следует применить, ответьте на несколько вопросов.

- *Какие еще операции вам потребуются?* Будет ли выполняться поиск произвольных ключей или только наименьшего ключа? Будут ли удаляться произвольные элементы данных или просто будут повторно удаляться верхний или нижний элемент?
- *Известен ли заранее максимальный размер структуры данных?* Здесь вопрос заключается в том, можно ли будет выделить место для структуры данных заранее.
- *Можно ли менять приоритет элементов, уже поставленных в очередь?* Возможность изменять приоритет элементов означает, что, кроме извлечения наибольшего элемента, элементы можно извлекать из очереди по ключу.

Перечислим базовые реализации очереди с приоритетами:

- *упорядоченный массив или список.* Упорядоченный массив очень эффективен как для идентификации наименьшего элемента, так и для его удаления путем уменьшения значения максимального индекса. Но поддержание упорядоченности элементов замедляет вставку новых элементов. Упорядоченные массивы подходят для реализации очереди с приоритетами только тогда, когда в нее будет осуществляться очень небольшое количество вставок. Базовые реализации очередей с приоритетами рассматриваются в *разделе 3.5*;
- *двоичные пирамиды.* Эта простая, элегантная структура данных поддерживает как операцию вставки, так и операцию извлечения наименьшего элемента за время $O(\lg n)$. Пирамиды позволяют поддерживать явную структуру двоичного дерева в массиве, чтобы значение ключа корня любого поддерева было меньшим, чем любого из его потомков. Таким образом, наименьший ключ всегда находится вверху пирамиды. Новые элементы можно добавлять, вставляя элемент в свободный лист дерева и перемещая его вверх до тех пор, пока он не займет правильное место при условии частичной упорядоченности. Реализация двоичной пирамиды на языке C рассматривается в *разделе 4.3.2*, а операция извлечения из нее наименьшего элемента — в *разделе 4.3.3*.

  Двоичные пирамиды являются удачным выбором в том случае, когда известна верхняя граница количества элементов в очереди с приоритетами, т. к. размер массива нужно задавать при его создании. Но даже это ограничение можно обойти с помощью динамических массивов (см. *раздел 3.1.1*);
- *очередь с приоритетами, имеющая ограничение по высоте.* Эта структура данных на основе массива позволяет выполнять операции вставки и поиска наименьшего элемента за постоянное время при ограниченном диапазоне возможных значений ключей. Допустим, мы знаем, что все значения ключей будут целыми числами в диапазоне от 1 до n. Тогда можно создать массив из n связных списков, где i-й список играет роль корзины, содержащей все элементы с ключом i. На наименьший непустой список будем поддерживать указатель *top* на наименьший непустой список. Чтобы вставить в очередь с приоритетами элемент с ключом k, добавляем его к k-й корзине и присваиваем этому указателю значение $top = \min(top,k)$. Чтобы извлечь наименьший элемент, определяем первый элемент в корзине *top*, удаляем его и, если корзина стала пустой, перемещаем указатель *top* вниз.

Очереди с приоритетами, имеющие ограничение по высоте, очень полезны для хранения вершин графа, упорядоченных по их степеням, при том, что такое упорядочивание является одной из основных операций в алгоритмах на графах. Тем не менее, при этом они применяются не так широко, как того заслуживают. Обычно этот тип очереди подходит для случаев небольших, дискретных диапазонов ключей;

- *двоичные деревья поиска.* Из двоичных деревьев поиска получаются эффективные очереди с приоритетами, т. к. самый левый лист всегда содержит наименьший элемент, а самый правый — наибольший. Чтобы найти наименьший (наибольший) элемент, просто выполняется проход по указателям влево (вправо) вниз до тех пор, пока указатель не станет нулевым. Двоичные деревья поиска оказываются наиболее подходящими, когда требуется осуществлять другие словарные операции или в случае неограниченного диапазона ключей, когда максимальный размер очереди с приоритетами неизвестен заранее;
- *Фибоначчиевы и парные (pairing) пирамиды.* Эти сложные очереди с приоритетами предназначены для ускорения выполнения операции *уменьшения ключа*, которая понижает приоритет поставленного в очередь элемента. Такая ситуация возникает, например, в вычислениях кратчайшего пути, при обнаружении более короткого, чем найденный ранее, маршрута к вершине v.

  Если их реализовать и использовать должным образом, то этот тип очереди с приоритетами может улучшить производительность при больших объемах вычислений.

Реализации. Современные языки программирования содержат библиотеки подробных и эффективных реализаций очередей с приоритетами. Методы `push`, `top` и `pop` шаблона `priority_queue` библиотеки STL языка C++ воспроизводят операции с пирамидами `insert`, `findmax` и `deletemax`. Библиотеку STL можно загрузить вместе с документацией с веб-сайта **http://www.sgi.com/tech/stl/**. Более подробное руководство по использованию библиотеки STL можно найти в книгах [Mey01] и [MDS01].

Библиотека LEDA (см. *раздел 19.1.1*) содержит полную коллекцию очередей с приоритетами на языке C++, включая Фибоначчиевы пирамиды, парные пирамиды, деревья Емде Боаса и очереди с приоритетами, имеющие ограничение по высоте. В результате экспериментов, описанных в книге [MN99], было установлено, что простые двоичные пирамиды являются довольно конкурентоспособными в большинстве приложений, при этом в прямом сравнении парные пирамиды показывают лучшую производительность, чем Фибоначчиевы пирамиды.

Класс `PriorityQueue` библиотеки Java Collections (JC) входит в стандартный пакет утилит Java java.util (**http://java.sun.com/javase/**). Библиотека Data Structures Library in Java (JDSL) предоставляет альтернативную реализацию, которую для некоммерческого применения можно загрузить с веб-сайта **http://www.jdsl.org/**. Более подробное руководство по библиотеке JDSL можно найти в книгах [GTV05] и [GT05].

В статье [San00] изложено описание экспериментов, демонстрирующих, что производительность разработанной автором последовательной пирамиды (sequence heap), основанной на k-м слиянии, приблизительно вдвое лучше, чем хорошо реализованной двоичной пирамиды. Реализацию этой пирамиды можно загрузить с веб-страницы **http://www.mpi-inf.mpg.de/~sanders/programs/spq/**.

Примечания

В книге [MS05] приводится всесторонний обзор современного состояния дел в том, что касается очередей с приоритетами. Результаты экспериментальных сравнений структур данных, реализующих очереди с приоритетами, представлены в таких работах, как [CG99], [GBY91], [Jon86] и [San00].

Двусторонние очереди с приоритетами (double-ended priority queue) позволяют расширить набор операций с пирамидами, включив в него одновременно операции `find-min` и `find-max`. Обзор четырех разных реализаций двусторонних очередей с приоритетами см. в [Sah05].

Очереди с приоритетами, имеющие ограничение по высоте, являются подходящими структурами данных для использования на практике, но они не гарантируют высокой производительности в наихудшем случае, когда разрешены произвольные вставки и удаления. Однако очереди с приоритетами Емде Боаса (см. [vEBKZ77]) позволяют выполнять операции вставки, удаления, поиска, определения наибольшего и наименьшего элементов за время $O(\lg\lg n)$, где каждый ключ имеет значение от 1 до n.

Фибоначчиевы пирамиды (см. [FT87]) поддерживают выполнение операций вставки и уменьшения ключа за амортизированное постоянное время, а выполнение операций извлечения наименьшего элемента и удаления — за время $O(\lg n)$. Постоянное время исполнения операции уменьшения ключа позволяет получить более быструю реализацию алгоритмов поиска кратчайшего пути, паросочетаний во взвешенных двудольных графах и минимальных остовных деревьев. Фибоначчиевы пирамиды на практике трудно реализовать, и время исполнения их операций содержит большие постоянные коэффициенты. Парные пирамиды поддерживают такие же пределы, но при меньших накладных расходах. Эксперименты с парными пирамидами описан,ы в журнале [SV87].

Пирамиды поддерживают частичное упорядочивание, которое можно создать за линейное количество операций сравнения. Обычные алгоритмы слияния с линейным временем исполнения для создания пирамид описаны в [Flo64]. В наихудшем случае достаточно $1{,}625n$ сравнений (см. [GM86]), а вообще необходимо от $1{,}5n$ до $O(\lg n)$ сравнений.

Родственные задачи. Словари (см. *раздел 12.1*), сортировка (см. *раздел 14.1*), поиск кратчайшего пути (см. *раздел 15.4*).

12.3. Суффиксные деревья и массивы

Вход. Строка S.

Задача. Создать структуру данных, позволяющую быстро определять местоположение произвольной строки запроса q в строке S (рис. 12.3).

Обсуждение. Суффиксные деревья и массивы являются исключительно полезными структурами данных для элегантного и эффективного решения задач по обработке строк. Правильное применение суффиксных деревьев часто позволяет уменьшить время алгоритмов для обработки строк с $O(n^2)$ до линейного. Суффиксные деревья упоминались в истории из жизни, описанной в *разделе 3.9*.

Простейший экземпляр суффиксного дерева представляет собой простое *нагруженное дерево* (trie) из n суффиксов строки S длиной в n символов. Нагруженном деревом называется древовидная структура, в которой каждое ребро представляет один символ, а корень — нулевую строку. Таким образом, каждый путь из кормя представляет строку,

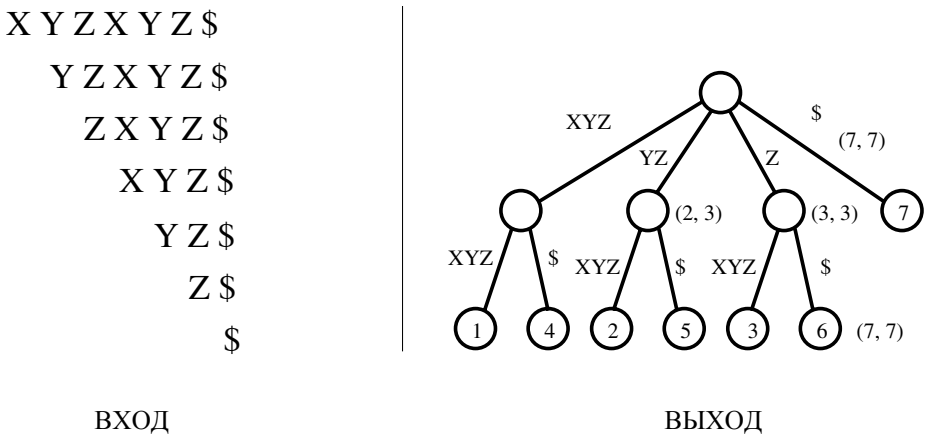


Рис. 12.3. Суффиксное дерево

описываемую символам и, которые маркируют проходимые ребра. Любой конечный набор слов определяет нагруженное дерево, а дерево для двух слов с общим префиксом разветвляется на первом несовпадающем символе. Каждый лист дерева обозначает конец строки. На рис. 12.4 показан пример простого нагруженного дерева.

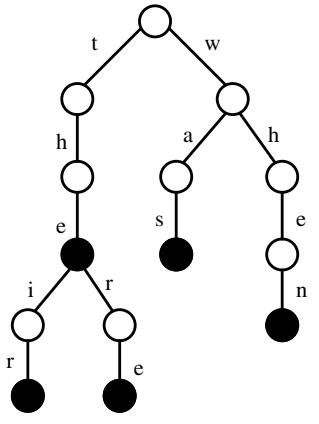


Рис. 12.4. Нагруженное дерево для строк *the*, *their*, *there*, *was* и *when*

Нагруженные деревья полезны для проверки на вхождение данной строки запроса q в набор строк. Обход нагруженного дерева выполняется с корня вдоль ветвей, определяемых последующими символами строки q. Если какая-либо ветвь отсутствует в нагруженном дереве, то тогда строка q не может быть элементом набора строк, определяемых данным нагруженным деревом. В противном случае строка q находится за $|q|$ сравнений символов, независимо от количества строк, содержащихся в нагруженном дереве. Нагруженные деревья очень легко создавать (последовательно вставляя новые строки), и в них быстро выполняется поиск, хотя они могут оказаться дорогими в смысле расхода памяти.

***Суффиксное дерево* — это просто нагруженное дерево всех суффиксов строки S. Суффиксное дерево позволяет проверить, является ли строка запроса q подстрокой строки**

S, т. к. любая подстрока строки S представляет собой префикс какого-либо суффикса. Время поиска — линейное по отношению к длине строки запроса q.

Но проблема здесь заключается в том, что создание полного суффиксного дерева таким образом может занять время $O(n^2)$ и, что еще хуже, такой же объем памяти, т. к. средняя длина n суффиксов равна $n/2$. Но при рациональном подходе для представления полного суффиксного дерева затраты памяти будут линейными. Обратите внимание на то, что большинство узлов нагруженного суффиксного дерева находятся на простых путях между узлами на ветвях дерева. Каждый из этих простых путей соответствует подстроке первоначальной строки. Сохранив первоначальную строку в массиве и свернув каждый такой путь в одно ребро, всю информацию о полном суффиксном дереве можно сохранить в памяти объемом $O(n)$. Маркировка каждого ребра состоит из индексов начала и конца массива, представляющего подстроку. Такое свернутое суффиксное дерево показано на рис. 12.3.

Для создания свернутого дерева существуют алгоритмы, использующие дополнительные указатели, ускоряющие процесс построения. Время исполнения этих алгоритмов равно $O(n)$. Эти дополнительные указатели можно также использовать для ускорения многих приложений, основанных на суффиксных деревьях.

Но для чего можно использовать суффиксные деревья? Далее приводится краткое описание вариантов применения суффиксных деревьев. Подробности см. в книгах [Gus97] и [CR03].

- *Поиск всех вхождений подстроки q в строке S.* Так же, как и в случае с нагруженным деревом, мы можем выполнять обход, начиная с корня дерева по направлению к узлу n_q, связанному с подстрокой q. Позиции всех вхождений подстроки q в строку S представлены потомками узла n_q, которых можно определить посредством поиска в глубину, начиная с узла n_q. В свернутых суффиксных деревьях найти k вхождений подстроки q в строку S можно за время $O(|q| + k)$.
- *Поиск самой длинной подстроки, общей для набора строк.* Создается одно свернутое суффиксное дерево, содержащее все суффиксы всех строк, в котором каждый лист дерева помечен его исходной строкой. В процессе обхода в глубину этого дерева мы можем пометить каждый его узел информацией о длине его префикса и количестве разных строк, являющихся его потомками. По этой информации наилучший узел можно найти за линейное время.
- *Поиск самого длинного палиндрома в строке S. Палиндром* — это строка, которая читается одинаково в обоих направлениях, например *мадам*. Чтобы найти самый длинный палиндром в строке S, создаем суффиксное дерево, содержащее все суффиксы строки S и строки, обратной S. Каждый лист этого дерева будет идентифицироваться его начальной позицией. Палиндром определяется любым узлом этого дерева, у которого из одной позиции выходят потомки, одинаковые в обоих направлениях.

Так как алгоритмы с линейным временем исполнения для создания суффиксных деревьев являются нетривиальными, вместо самостоятельной разработки таких алгоритмов рекомендуется использовать существующие реализации. В качестве альтернативы можно использовать суффиксные массивы.

Суффиксные массивы позволяют делать практически все то, что и суффиксные деревья, но при этом используют приблизительно в четыре раза меньший объем памяти. Кроме этого, их легче реализовать. В принципе, суффиксный массив представляет собой просто массив, содержащий все n суффиксов строки S в отсортированном порядке. Таким образом, двоичного поиска строки q в этом массиве оказывается достаточно для локализации префикса суффикса, совпадающего с подстрокой q, что гарантирует эффективный поиск подстроки за $O(\lg n)$ сравнений строк. После добавления индекса, указывающего общую длину префикса всех граничных суффиксов, для любого запроса потребуется только $\lg n + |q|$ *сравнений символов*, т. к. появляется возможность определять, какой символ нужно проверить следующим при двоичном поиске. Например, если нижней границей диапазона поиска является слово *cowabunga*, а верхней — *cowslip*, то все промежуточные ключи будут совпадать по первым трем символам, поэтому сравнивать с подстрокой q нужно только четвертый символ любого промежуточного ключа. На практике скорость поиска в суффиксных массивах обычно такая же, как в суффиксных деревьях или даже выше.

Кроме этого, суффиксные массивы требуют меньше памяти, чем суффиксные деревья. Каждый суффикс представляется в них уникальной начальной позицией (от 1 до n) и считывается по мере надобности с использованием одной эталонной копии входной строки.

Но для эффективного создания суффиксных массивов нужно соблюдать определенную осторожность, т. к. сортируемые строки содержат $O(n^2)$ символов. Одним из решений — сначала создать суффиксное дерево, после чего выполнить симметричный обход этого дерева, чтобы прочитать строки в отсортированном порядке! Но последние достижения в этой области позволяют создавать эффективные по времени и памяти алгоритмы для построения суффиксных массивов напрямую.

Реализации. В настоящее время существует большое количество реализаций суффиксных массивов. По сути, все последние алгоритмы для построения суффиксных массивов за линейное время были реализованы, а их сравнительная производительность измерена (см. [PST07]). Превосходная реализация суффиксных массивов на языке C описывается в книге [SS07]. Загрузить эту реализацию можно с веб-сайта **http://bibiserv.techfak.uni-bielefeld.de/bpr/**.

А на веб-сайте Pizza&Chili Corputs (**http://pizzachili.di.unipi.it**) представлены восемь разных реализаций на C/C++ для сжатых текстовых индексов. Эти структуры данных позволяют значительно минимизировать расход памяти за счет чрезвычайно эффективного сжатия строки входа. При этом они обеспечивают отличное время исполнения запросов.

Вполне доступны и реализации суффиксных деревьев. Проект с открытым кодом BioJava (**http://www.biojava.org**), предоставляющий инфраструктуру Java для обработки биологических данных, содержит класс `SuffixTree`. Реализацию алгоритма Укконена на языке C (называемую Libstree) можно загрузить с веб-сайта **http://www.icir.org/christian/libstree/**.

А с веб-сайта **http://marknelson.us/1996/08/01/suffix-trees** можно загрузить код Нельсона на языке C++ (см. [Nel96]).

Программы на языке C коллекции strmat реализуют алгоритмы для точного сравнения шаблонов, включая реализацию суффиксных деревьев (см. [Gus97]). Эту коллекцию можно загрузить с веб-сайта **http://www.cs.ucdavis.edu/~gusfield/strmat.html**.

Примечания

Нагруженные деревья (trie) были впервые предложены Е. Фредкином (E. Fredkin) в журнале [Fre62]. В книге [GBY91] представлен обзор основных структур данных нагруженных деревьев, сопровождаемый многочисленными ссылками.

Эффективные алгоритмы для построения суффиксных деревьев были созданы Вейнером (Weiner), МакКрейтом (McCreight) и Укконеном (Ukkonen) (см. [Wei73], [McC76] и [Ukk92] соответственно). Хорошие описания этих алгоритмов даны в журнале [CR03] и в книге [Gus97].

Суффиксные массивы были изобретены Манбером (Manber) и Майерсом (Myers) (см. [MM93]), хотя аналогичная идея Гоннета (Gonnet) и Баеза-Йейтса (Baeza-Yates) была высказана в [GBY91]. Алгоритмы построения суффиксных массивов с линейным временем исполнения были разработаны тремя независимыми командами в 2003 г. (см. [KSPP03], [KA03] и [KSB05]) и с тех пор эта область успешно развивается. Обзор последних разработок представлен в [PST07].

Результатом последних работ является создание сжатых полнотекстовых индексов, которые позволяют реализовать практически все возможности суффиксных деревьев и массивов в структуре данных, размер которой пропорционален размеру *сжатой* текстовой строки. Обзор этих важных структур данных см. в [MN07].

Возможности суффиксных деревьев можно расширить, используя структуру данных для вычисления наименьшего общего предшественника любой пары узлов (x, y) за постоянное время после предварительной обработки дерева за линейное время. Первоначально эта структура данных была предложена Харелом (Harel) и Тарьяном (Tarjan) (см. [HT84]), после чего была упрощена сначала Шибером (Schieber) и Вишкиным (Vishkin) (см. [SV88]), а потом Бендером (Bender) и Фараком (Farach) (см. [BF00]). Ее описание можно найти, например, в [Gus97]. Наименьший общий предшественник двух узлов суффиксного дерева или нагруженного дерева определяет узел, представляющий самый длинный общий префикс двух ассоциированных строк. Удивительно, что осуществление таких запросов происходит за постоянное время, и этот факт делает их пригодными в качестве компонентов многих других алгоритмов.

Родственные задачи. Поиск подстрок (см. *раздел 18.3*), сжатие текста (см. *раздел 18.5*), поиск самой длинной общей подстроки (см. *раздел 18.8*).

12.4. Графы

Вход. Граф G.

Задача. Представить граф G посредством гибкой и эффективной структуры данных (рис. 12.5).

Обсуждение. Двумя основными структурами данных для представления графов являются *матрицы смежности* и *списки смежности*. Подробное описание этих структур данных вместе с реализацией списков смежности на языке *C* дано в *разделе 5.2*. Для большинства задач наиболее подходящей структурой данных являются списки смежности.

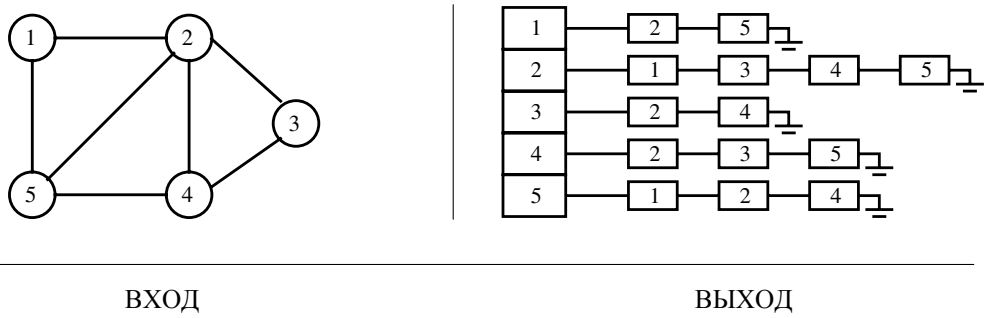


Рис. 12.5. Варианты представления графа

Выбирая структуру данных, ответьте на несколько вопросов.

- *Каким будет размер графа?* Сколько вершин и ребер будет иметь граф как в типичном, так и в наихудшем случае? Для графов с 1 000 вершин требуются матрицы смежности из 1 000 000 элементов, что вряд ли приемлемо. Поэтому матрицы смежности подходят только для представления небольших или очень плотных графов.
- *Какой будет плотность графа?* Если граф очень плотный, т. е. ребра определены для большей части возможных пар вершин, то, пожалуй, у вас нет особой необходимости использовать списки смежности. В любом случае объем требуемой памяти будет $O(n^2)$, т. к. благодаря отсутствию указателей матрицы смежности для полных графов будут иметь меньший размер.
- *Какой алгоритм будет применяться?* В некоторых алгоритмах лучше использовать матрицы смежности (например, в алгоритме поиска кратчайшего пути между всеми парами вершин), а в других — списки смежности (например, в большинстве алгоритмов, основанных на обходе в глубину). Матрицы смежности имеют явное преимущество в алгоритмах, в которых многократно выясняется наличие конкретного ребра в графе. Впрочем, большинство алгоритмов на графах можно разработать так, что подобные запросы в них исключены.
- *Будет ли граф модифицироваться при выполнении приложения?* Если после создания графа в него не будут вставляться и/или удаляться ребра, то можно использовать эффективные реализации статических графов. В действительности, гораздо чаще, чем топология графа, модифицируются *атрибуты* его ребер или вершин — размер, вес, метка или цвет. Атрибуты лучше всего обрабатывать с помощью дополнительных полей в записях ребер или вершин в списках смежности.

Разработать универсальный программный тип графа весьма непросто. Поэтому я рекомендую проверить наличие уже существующей реализации (в особенности в библиотеке LEDA), прежде чем создавать свою собственную. Обратите внимание, что преобразование между матрицей смежности и списком смежности занимает линейное время по отношению к размеру большей структуры данных. Маловероятно, что это преобразование окажется узким местом в каком-либо приложении, поэтому не исключено использование обеих структур данных, если имеется достаточно памяти. Обычно в этом нет необходимости, но такое решение может оказаться самым простым, если не совсем понятно, какую структуру выбрать.

Планарными называются такие графы, которые можно начертить на плоскости без пересечения ребер. Возникающие во многих приложениях графы являются планарными по определению, например карты стран. Другие графы, например деревья, планарны сами по себе. Планарные графы всегда разреженные, т. к. любой планарный граф с n вершинами может иметь, самое большее, $3n–6$ ребер. Поэтому для представления этих графов следует использовать списки смежности. Если планарное представление графа (или укладка) играет важную роль в решаемой задаче, то планарный граф лучше всего представлять геометрическим способом. Информацию об алгоритмах создания укладок графов см. в *разделе 15.12*. В *разделе 17.15* рассматриваются алгоритмы построения графов, неявно возникающих при размещении геометрических объектов, таких как линии и многоугольники.

Гиперграфами называются обобщенные графы, в которых каждое ребро может связывать любое количество вершин. Допустим, что мы хотим представить в виде графа членство конгрессменов в разных комитетах. Такой граф будет гиперграфом, каждая вершина которого представляет отдельного конгрессмена, а каждое гиперребро, соединяющее несколько вершин, представляет один комитет. Такие произвольные наборы подмножеств некоторого множества естественно рассматривать в виде гиперграфов.

При работе с гиперграфами используются такие базовые структуры данных, как:

- *матрицы инцидентности*, которые аналогичны матрицам смежности. Для них требуется объем памяти размером $n \times m$, где m — количество гиперребер. Каждая строка такой матрицы соответствует вершине, а каждый столбец — ребру. Значение ячейки $M[i, j]$ будет ненулевым только в том случае, если вершина i входит в ребро j. Каждый столбец матрицы инцидентности стандартных графов содержит две ненулевые ячейки. Количество ненулевых позиций в каждой строке зависит от степени каждой вершины;
- *двудольные структуры инцидентности*, которые аналогичны спискам смежности и, следовательно, более подходят для использования с разреженными гиперграфами. В структуре инцидентности имеется вершина, связанная с каждым ребром и вершиной гиперграфа. Для представления такой структуры инцидентности обычно используются списки смежности. Естественным способом визуального представления гиперграфа является ассоциированный двудольный граф.

Эффективно представить очень большой граф довольно трудно. Тем не менее графы, содержащие миллионы ребер и вершин, использовались для решения интересных задач. В первую очередь следует сделать структуру данных как можно более экономной, упаковав матрицу смежности в битовый вектор (см. *раздел 12.5*) или удалив ненужные указатели из представления списка смежности. Например, в статическом графе (не поддерживающем вставку или удаление ребер) каждый список ребер можно заменить упакованным массивом идентификаторов вершин, что позволит избавиться от указателей и сэкономить половину памяти.

В случае чрезвычайно больших графов может понадобиться иерархическое представление, в котором группы вершин собираются в подграфы, каждый из которых сжимается в одну вершину. Такие иерархические декомпозиции можно создавать двумя спо-

собами. В первом граф разбивается на компоненты естественным или специфическим для приложения способом. Например, граф дорог и населенных пунктов имеет естественную декомпозицию — разделение карты на населенные пункты, районы и области. При втором подходе выполняется алгоритм разбиения графа, рассматриваемый в *разделе 16.6*. Для NP-полной задачи естественная декомпозиция, скорее всего, даст более приемлемый результат, чем наивный эвристический алгоритм. Если же граф действительно очень большой, вы не сможете позволить себе его алгоритмическое разбиение. Поэтому, прежде чем предпринимать такие решительные действия, убедитесь, что стандартные структуры данных неприменимы к вашей задаче.

Реализации. Самую лучшую на сегодня реализацию структур данных для представления графов на языке C++ содержит библиотека LEDA (см. *раздел 19.1.1*). Хотя она и является коммерческим продуктом, вам следует изучить представленные в ней методы для работы с графами, чтобы увидеть, как правильный уровень абстракции облегчает задачу реализации алгоритмов.

Более доступной является библиотека Boost Graph Library (см. [SLL02]), которую можно загрузить с веб-сайта **http://www.boost.org/libs/graph/doc**. Библиотека включает реализации списков смежности, матриц смежности и списков ребер, а также неплохую коллекцию базовых алгоритмов для решения задач на графах. Ее интерфейс и компоненты являются обобщенными в том же смысле, что и у библиотеки STL на языке C++.

Библиотека графов JUNG (**http://jung.sourceforge.net**) особенно популярна среди разработчиков социальных сетей. Библиотека Data Structures Library in Java (JDSL) предоставляет реализацию разнообразных структур данных с неплохой библиотекой алгоритмов, которую можно загрузить с веб-сайта **http://www.jdsl.org/** для некоммерческого применения. Более подробное описание библиотеки JDSL см. в книгах [GTV05] и [GT05]. Библиотека JGraphT (**http://jgrapht.sourceforge.net**) является более поздней разработкой с аналогичной функциональностью.

Программа Stanford GraphBase (см. *раздел 19.1.8*) предоставляет простую, но гибкую структуру данных для реализации графов, разработанную с использованием методологии CWEB. Поучительно рассмотреть, что Кнут реализовал (и чего не реализовал) в этой базовой структуре данных, хотя в качестве основы для разработок я бы рекомендовал другие реализации.

Среди всех реализаций типов графов на языке C я (субъективно) предпочитаю библиотеку из моей книги "Programming Challenges" (см. [SR03]). Подробности см. в *разделе 19.1.10*. Простые структуры данных для представления графов на языке программирования пакета Mathematica, а также библиотека алгоритмов и процедур вывода имеются в библиотеке Combinatorica (см. [PS03]). Подробности см. в *разделе 19.1.9*.

Примечания

Преимущества списков смежности для работы с графами становятся очевидными при использовании алгоритмов Хопкрофта (Hopcroft) и Тарьяна (Tarjan) с линейным временем исполнения (см. [HT73b], [Tar72]). Базовые структуры данных списков и матриц смежности рассматриваются практически во всех книгах по алгоритмам или структурам данных, включая книги [CLRS01], [AHU83] и [Tar83]. Гиперграфы обсуждаются в книге [Ber89].

Эффективность применения статических графов была показана Неером (Naher) и Злотовским (Zlotowski) в [NZ02], которым удалось повысить скорость исполнения некоторых алгоритмов из библиотеки LEDA в четыре раза, просто используя более компактную структуру для представления графов.

Интересным вопросом является минимизация количества битов, необходимых для представления произвольных графов, состоящих из n вершин, особенно если требуется эффективное исполнение определенных операций. Эта тема обсуждается в книге [vL90b].

Динамические алгоритмы решения задач на графах поддерживают быстрый доступ к инвариантам (таким как минимальное остовное дерево) при вставке или удалении ребер. Общим подходом к созданию динамических алгоритмов является *разрежение* (sparsification) (см. [EGIN92]). Результаты экспериментальных исследований практически всех динамических алгоритмов решения задач на графах изложены в публикациях [ACI92] и [Zar02].

Иерархически определяемые графы часто возникают в задачах разработки микросхем со сверхвысоким уровнем интеграции, т. к. их разработчики активно используют библиотеки схемных элементов (см. [Len90]). Алгоритмы, специфичные для иерархически определяемых графов, включают в себя проверку на планарность (см. [Len89]), проверку на связность (см. [LW88]) и построение минимальных остовных деревьев (см. [Len87a]).

Родственные задачи. Структуры данных множеств (см. *раздел 12.5*), разделение графа (см. *раздел 16.6*).

12.5. Множества

Вход. Универсальное множество элементов $U = \{u_1, ..., u_n\}$, по которому определена коллекция подмножеств $S = \{S_1, ..., S_m\}$.

Задача. Представить каждое подмножество таким образом, чтобы можно было эффективно проверять вхождение элемента u_i в подмножество S_j, вычислять объединение или пересечение подмножеств S_i и S_j, вставлять или удалять члены коллекции S (рис. 12.6).

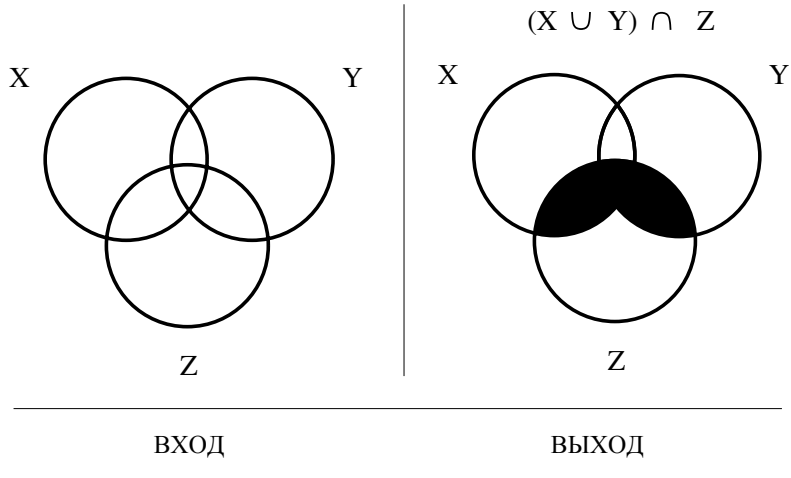


Рис. 12.6. Множества X. Y. Z

Обсуждение. В математике множеством называется неупорядоченная коллекция объéктов из фиксированного универсального множества. Но при практической реализации удобно представлять множества в едином *канонически упорядоченном виде*, обычно отсортированном, чтобы ускорить или упростить разные операции. Упорядоченное множество превращает задачу поиска объединения или пересечения двух подмножеств в операцию с линейным временем исполнения — мы просто перебираем элементы слева направо и выясняем, какие отсутствуют. Кроме этого, в упорядоченном множестве поиск элемента занимает сублинейное время. Однако распечатка элементов канонически упорядоченного множества парадоксальным образом свидетельствует о том, что порядок в действительности не имеет значения.

Множества отличаются от словарей или строк. Коллекцию объектов, составленную *не* из универсального множества постоянного размера, лучше всего рассматривать как *словарь* (см. *раздел 12.1*). Строки являются структурами, в которых порядок элементов имеет значение, т. е., строка $\{A, B, C\}$ не равна строке $\{B, C, A\}$. Структуры данных и алгоритмы для работы со строками рассматриваются в *разделе 12.3* и *главе 18*.

В *мультимножествах* допустимо вхождение одного и того же элемента больше одного раза. Структуры данных для множеств обычно можно расширить для мультимножеств, поддерживая поле счетчика или связный список одинаковых вхождений каждого элемента мультимножества.

Если каждое подмножество содержит ровно два элемента, то эти подмножества можно рассматривать как ребра графа, вершины которого представляют универсальное множество. Система подмножеств без ограничений на количество ее элементов называется *гиперграфом*. Вы должны задать себе вопрос, существует ли для вашей задачи аналогия из теории графов, такая как поиск связных компонент или кратчайшего пути в графе.

Перечислим основные структуры данных для представления произвольных подмножеств:

- *битовые векторы.* Посредством битового вектора или массива из n битов можно представить любое подмножество S из универсального множества U, содержащее n элементов. Если $i \in S$, то значение i-го бита будет 1, а в противном случае — 0. Так как для представления каждого элемента требуется только один бит, то битовые векторы позволяют экономить память даже при очень больших значениях $|U|$. Для вставки или удаления элемента значение соответствующего бита просто меняется на противоположное. Пересечения и объединения вычисляются выполнением логических операций И и ИЛИ. Единственным недостатком битового вектора является его неудовлетворительная производительность на разреженных подмножествах. Например, чтобы явно идентифицировать все члены разреженного (даже пустого) подмножества S, требуется время $O(n)$;
- *контейнеры или словари.* Подмножество можно представить посредством связного списка, массива или словаря, содержащего элементы подмножества. Для такой структуры данных не требуется идея постоянного универсального множества. Для разреженных подмножеств словари могут обеспечить бóльшую экономию памяти и времени, чем битовые векторы; к тому же, с ними легче работать. Для эффективности выполнения операций объединения и пересечения следует поддерживать эле-

менты каждого подмножества в отсортированном порядке, что позволяет идентифицировать все дубликаты проходом по обоим подмножествам за линейное время;

- *фильтры Блума.* При отсутствии фиксированного универсального множества битовый вектор можно эмулировать, хэшируя каждый элемент подмножества в целое число в диапазоне от 0 до n и устанавливая соответствующий бит. Таким образом, если $e \in S$, то биту $H(e)$ будет присвоено значение 1. Однако при использовании этой схемы могут возникнуть коллизии, поскольку разные ключи могут быть хэшированы в одно и то же целое число.

  Чтобы снизить уровень ошибок, в фильтрах Блума используется несколько (скажем, k) разных хэш-функций H_1, ..., H_k, и после вставки ключа e всем k битам $H_i(e)$ присваивается значение 1. Теперь элемент e является членом S только в том случае, когда значения всех битов k равны 1. Таким образом, увеличивая количество хэш-функций и размер таблицы, можно снизить вероятность ошибки. Используя правильно подобранные константы, можно представить каждый элемент подмножества посредством фиксированного количества битов, независимого от размера универсального множества.

  Эта структура, основанная на хэшировании, намного экономичнее словарей в смысле расхода памяти. Она годится для приложений, работающих на статических подмножествах и допускающих небольшую вероятность ошибки. Таких приложений не так уж мало. Например, при проверке правописания не произойдет большой трагедии, если программа иногда пропустит ошибочно написанное слово.

Многие приложения работают с попарно непересекающимися коллекциями подмножеств, в которых каждый отдельный элемент является членом ровно одного подмножества. В качестве примера рассмотрим задачу представления компонент связности графа или принадлежности политика к партии. Здесь каждая вершина (политик) является членом ровно одной компоненты (партии). Такая система подмножеств называется *разбиением множества.* Алгоритмы генерирования разбиений множества рассматриваются в *разделе 14.6.*

Важным моментом при работе с такими структурами данных является сопровождение изменений на протяжении времени, когда в компоненту добавляются или удаляются ребра (в партию вступают новые члены или старые выходят из нее). Типичные вопросы, возникающие при модифицировании множества путем изменения одного элемента, слияния или объединения двух элементов или же разбиения множества на части, сводятся к двум: "В каком множестве находится определенный элемент?" и "Находятся ли оба элемента в одном и том же множестве?".

Нам понадобятся следующие структуры данных:

- *коллекция контейнеров.* Представление каждого подмножества в его собственном контейнере/словаре обеспечивает быстрый доступ ко всем элементам подмножества, что облегчает выполнение операций объединения и пересечения. Но операция проверки членства является дорогой, т. к. требует отдельного поиска в каждой структуре данных до тех пор, пока не будет найден искомый элемент;
- *обобщенный битовый вектор.* Пусть i-й элемент массива содержит номер/имя содержащего его подмножества. Запросы, идентифицирующие множество, и модифи-

кации отдельных элементов можно выполнять за постоянное время. Но для объединения двух подмножеств может потребоваться время, пропорциональное размеру универсального множества, т. к. необходимо идентифицировать каждый элемент в этих двух подмножествах и изменить его имя (по крайней мере, в одном подмножестве);

- *словарь с атрибутом подмножества.* Подобным образом каждый элемент в двоичном дереве можно связать с полем, в котором хранится имя содержащего его подмножества. Запросы для идентификации множества и модифицирования одного элемента можно выполнять за время, требуемое для выполнения поиска в словаре. Однако операции объединения и пересечения по-прежнему выполняются медленно. Необходимость эффективно исполнять операции объединения заставляет нас использовать структуру данных "объединение-поиск";
- *структура данных "объединение-поиск".* Подмножество представляется посредством корневого дерева, в котором каждый узел указывает вместо своего потомка на своего предшественника. Именем каждого подмножества служит имя корневого элемента. Членство определяется с легкостью — мы просто перемещаемся от одного указателя на родительский элемент на другой, пока не дойдем до корня. Операция объединения двух подмножеств также не представляет сложностей. Мы просто присваиваем корню одного из деревьев указатель на другое, после чего *все* элементы получают общий корень и, следовательно, одно и то же имя подмножества.

  В данном случае детали реализации оказывают большое влияние на асимптотическую оценку производительности. Если при выполнении операции слияния всегда выбирать большее (более высокое) дерево в качестве корневого, то в результате гарантированно получатся деревья логарифмической высоты (см. *раздел 6.1.3*). Повторное прохождение пути, проложенного при каждом поиске элемента, и снабжение всех узлов в этом пути явными указателями на корень делает высоту дерева почти постоянной. Структура данных "объединение-поиск" является простой и быстро работающей структурой, с которой должен быть знаком каждый программист. Она не поддерживает разбиение подмножеств, созданных путем объединения, но это обычно не создает проблем.

Реализации. Современные языки программирования содержат библиотеки полных и эффективных реализаций множеств. Библиотека стандартных шаблонов STL языка C++ содержит контейнеры множеств (`set`) и мультимножеств (`multiset`). Стандартный пакет утилит java.util включает в себя библиотеку Java Collections (JC), которая содержит контейнеры `HashSet` и `TreeSet`.

Библиотека LEDA (см. *раздел 19.1.1*) содержит эффективные структуры данных словарей, разреженные массивы и структуры данных "объединение-поиск" для работы с разбиениями множеств, все на языке C++.

Реализация структуры данных "объединение-поиск" лежит в основе любой реализации алгоритма Крускала для построения минимального остовного дерева. Поэтому предполагается, что все библиотеки графов, упоминаемые в *разделе 12.4*, содержат эту реализацию. Реализации минимальных остовных деревьев рассматриваются в *разделе 15.3*.

Система компьютерной алгебры REDUCE (**http://www.reduce-algebra.com**) содержит пакет SETS, который поддерживает теоретико-множественные операции как над яв-

ными, так и над неявными (символическими) множествами. Другие системы компьютерной алгебры могут иметь аналогичную функциональность.

Примечания

Оптимальные алгоритмы для таких операций над множествами, как пересечение и объединение, были представлены в публикации [Rei72]. В книге [Ram05] прекрасно описаны структуры данных для выполнения операций над множествами. Квалифицированно выполненный обзор фильтров Блума представлен в журнале [BM05], а результаты последних экспериментов в этой области изложены в докладе [PSS07].

Некоторые структуры данных сбалансированных деревьев поддерживают операции *merge*, *meld*, *link* и *cut*, что позволяет быстро выполнять операции объединения над непересекающимися подмножествами. Хорошее описание таких структур можно найти в книге [Tar83]. Джекобсон (Jacobson) улучшил структуру данных битового вектора для эффективной поддержки операции выбора (поиска i-го установленного бита) как по времени, так и по памяти.

Обзор структур данных для объединения непересекающихся множеств представлен в работе [GI91]. Верхний предел производительности, равный $O(m\alpha(m, n))$, для m операций поиска и объединения в n-элементном множестве был вычислен Тарьяном (Tarjan) (см. [Tar75]). Им же был найден и соответствующий нижний предел в ограниченной модели вычислений (см. [Tar79]). Так как обратная функция Аккермана $\alpha(m, n)$ возрастает крайне медленно, то производительность приближается к линейной. Интересная связь между наихудшим случаем исполнения операций поиска и объединения и длиной последовательности Дэвенпорта-Шинцеля (Davenport-Schinzel) — комбинаторной структуры, рассматриваемой в вычислительной геометрии — представлена в книге [SA95].

Степенным множеством (power set) множества S называется коллекция всех $2^{|S|}$ подмножеств множества S. Явная манипуляция степенными множествами быстро становится трудной вследствие их большого размера. Для нетривиальных вычислений степенные множества требуется представлять неявно в символической форме. Информацию об алгоритмах работы с символическими представлениями степенных множеств и результатов вычислительных экспериментов с ними можно найти в [BCGR92].

Родственные задачи. Генерирование подмножеств (см. *раздел 14.5*), генерирование разбиений (см. *раздел 14.6*), вершинное покрытие (см. *раздел 18.1*), минимальное остовное дерево (см. *раздел 15.3*).

12.6. Kd-деревья

Вход. Множество S, состоящее из n точек или более сложных геометрических объектов в k-мерном пространстве.

Задача. Создать дерево, которое делит пространство полуплоскостями таким образом, что каждый объект содержится в своей собственной k-мерной прямоугольной области (рис. 12.7).

Обсуждение. Kd-деревья и связанные с ними пространственные структуры данных иерархически разбивают пространство на небольшое количество ячеек, каждая из которых содержит несколько представителей входного множества точек. Это обеспечивает быстрый доступ к любому объекту по его позиции. Мы просто проходим вниз по

иерархической структуре, пока не дойдем до наименьшей ячейки, содержащей требуемый элемент, после чего перебираем все элементы ячейки, чтобы найти нужный.

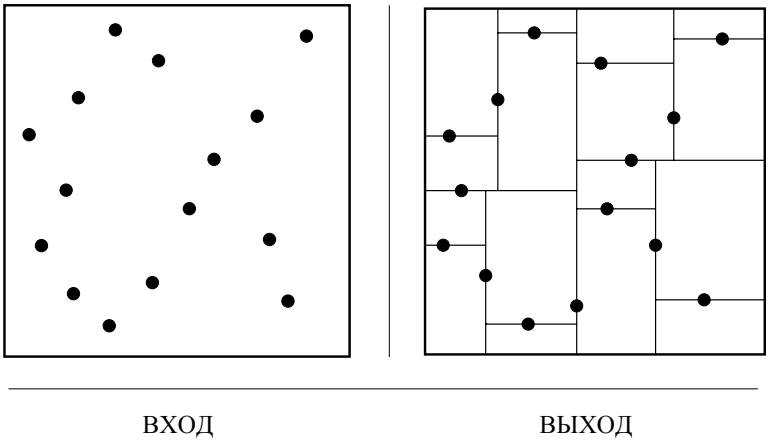


Рис. 12.7. Создание kd-дерева

Обычно алгоритмы создают kd-деревья путем разбиения множества точек. Каждый узел дерева определяется плоскостью, проходящей через одно из измерений. В идеале, эта плоскость разбивает множество точек на равные правое и левое (или верхнее и нижнее) подмножества. Эти подмножества опять разбиваются на равные половины плоскостями, проходящими через другое измерение. Процесс прекращается после $\lg n$ разбиений, при этом каждая точка оказывается в своей собственной листовой ячейке.

Секущие плоскости вдоль любого пути от корня к другому узлу определяют уникальную прямоугольную область пространства. Каждая следующая плоскость разделяет эту область на две меньших. Каждая прямоугольная область определяется $2k$ плоскостями, где k — количество измерений. По мере продвижения вниз по дереву мы продолжаем ограничивать интересующую нас область этими полупространствами.

Тип kd-дерева определяется выбором секущей плоскости. Возможны следующие варианты:

- *циклический проход по измерениям.* Сначала разбивается измерение d_1, потом d_2, ..., d_m, после чего мы возвращаемся к d_1;
- *сечение вдоль самого большого измерения.* Выбираем такое измерение, чтобы форма получившихся в результате его сечения областей была как можно ближе к квадратной. Разбиение точек пополам не обязательно означает, что секущая плоскость проходит точно посередине прямоугольной области, т. к. все точки могут лежать у одного края подлежащей сечению области;
- *квадрадеревья и октадеревья.* Вместо сечения одной плоскостью используются плоскости, параллельные всем координатным осям, которые проходят через данную точку сечения. Это означает, что в двух измерениях создаются четыре дочерние ячейки (квадрадерево), а в трех измерениях — восемь (октадерево). Использование квадрадеревьев особенно популярно при работе с видеоданными, когда листья дерева таковы, что все пикселы в этих областях имеют одинаковый цвет;

- *BSP-деревья.* При двоичном разбиении пространства используются секущие плоскости общего вида (т. е. не обязательно параллельные координатным осям), чтобы разделить пространство на ячейки таким образом, чтобы каждая ячейка содержала только один объект (скажем, многоугольник). Для некоторых наборов объектов такое разделение невозможно осуществить посредством только секущих плоскостей, параллельных координатным осям. Недостатком этого подхода является то обстоятельство, что получившиеся многогранные ячейки труднее поддаются обработке, чем ячейки прямоугольной формы;
- *R-деревья.* Это другая пространственная структура данных, подходящая для геометрических объектов, которые нельзя разделить по прямоугольным областям, ориентированным вдоль координатных осей, не разрезая их на части. На каждом уровне сечения объекты разбиваются на меньшее количество прямоугольных областей (возможно, накладывающихся друг на друга), чтобы создать удобные для поиска иерархические структуры, не разрезая объекты на части.

В идеале, сечения разделяют поровну как пространство (давая в результате большие области правильной формы), так и набор точек (обеспечивая дерево логарифмической высоты); но для определенного экземпляра входа выполнить сечение таким образом может оказаться невозможным. Преимущества ячеек большого размера становятся очевидными во многих применениях kd-деревьев, обсуждаемых далее.

- *Определение местоположения точки.* Для выявления ячейки, в которой находится целевая точка q, выполняется обход дерева, начиная с корня, в процессе которого оба полупространства с каждой стороны секущей плоскости проверяются на наличие в них данной точки. Повторяя этот процесс на соответствующем дочернем узле, мы спускаемся вниз по дереву, пока не найдем листовую ячейку с искомым элементом q. Такой поиск занимает время, пропорциональное высоте дерева. Дополнительную информацию по вопросу определения местоположения точки см. в *разделе 17.7*.
- *Поиск ближайшей точки.* Чтобы в множестве S найти точку, ближайшую к точке запроса q, выполняется процедура определения местоположения ячейки c, содержащей точку q. Так как на границе ячейки c находится какая-либо точка p, то можно вычислить расстояние $d(p, q)$ от точки p до точки q. Точка p, скорее всего, расположена вблизи точки q, но она может быть не единственной точкой, близкой к ней. Почему? Допустим, что точка q находится на границе ячейки. Тогда точка, ближайшая к точке q, может быть расположена сразу же слева от этой границы в другой ячейке. Значит, нам нужно обойти все ячейки, которые находятся от ячейки c на расстоянии не превышающем $d(p, q)$, и проверить, что ни одна из них не содержит более близких точек. В дереве из крупных ячеек правильной формы проверить нужно будет очень небольшое количество ячеек. Поиск ближайшей точки подробно обсуждается в *разделе 17.5*.
- *Поиск в диапазоне.* Как определить, какие точки находятся в целевой области? Начиная с корня дерева, проверяем целевую область на пересечение с ячейкой (или на вхождение в нее этой ячейки), определяющей текущий узел. При положительном результате проверки выполняем такую же проверку на потомках; в противном случае никакая из листовых ячеек ниже данного узла уже не будет нас интересовать.

Таким образом быстро отсекаются не имеющие отношения к делу области пространства. Поиск точки в указанном диапазоне подробно обсуждается в *разделе 17.6*.

- *Поиск по частичному ключу.* Допустим, что нам нужно найти точку p в множестве S, но мы не располагаем подробной информацией об этой точке. Предположим, мы ищем в 3-мерном дереве, имеющем измерения для возраста, роста и веса, узел, представляющий человека в возрасте 35 лет ростом 175 сантиметров и с неизвестным весом. Начиная путь от корня, мы можем найти подходящий потомок по всем измерениям, кроме веса. Чтобы иметь уверенность, что найдена нужная точка, мы должны искать *оба потомка* этих узлов. Чем больше полей (измерений) известно, тем лучше, но такой поиск точки по частичной информации о ней может пройти значительно быстрее, чем сравнение всех точек с искомым ключом.

Kd-деревья лучше всего годятся для приложений с умеренным количеством измерений, в диапазоне от 2 до 20. С увеличением количества измерений они теряют эффективность, в основном из-за того, что объем единичного шара в k-мерном пространстве уменьшается экспоненциально, в отличие от объема единичного куба. Следовательно, в приложениях, связанных, например, с поиском ближайшей точки, экспоненциально возрастает количество ячеек, проверяемых в пределах данного радиуса с центром в точке запроса. Кроме этого, для любой ячейки количество смежных ячеек увеличивается до $2k$ и, в конце концов, становится неуправляемым.

Главный вывод из сказанного состоит в том, что следует избегать работы в многомерных пространствах, возможно, отказываясь от рассмотрения наименее важных измерений.

Реализации. Программа KDTREE 2 содержит реализации kd-деревьев на языках C++ и FORTRAN 95 для эффективного поиска ближайшего соседа во многих измерениях. Подробности см. по адресу **http://arxiv.org/abs/physics/0408067**.

Апплеты Java коллекции Spatial Index Demos (**http://donar.umiacs.umd.edu/quadtree**) демонстрируют многие варианты kd-деревьев. Алгоритмы, реализуемые в апплетах, рассматриваются в книге [Sam06].

Библиотека с открытым кодом на языке C++, носящая имя TerraLib (**http://www.terralib.org**), предназначена для разработки приложений GIS (Geographic Information System, географическая информационная система). Эта библиотека включает в себя реализацию пространственных структур данных.

Соревнование DIMACS в 1999 г. фокусировалось на структурах данных для поиска ближайшего соседа (см. [GJM02]). Наборы данных и коды можно загрузить с веб-сайта **http://dimacs.rutgers.edu/Challenges**.

Примечания

Самым лучшим справочником по kd-деревьям и другим пространственным структурам данных является книга [Sam06]. В ней подробно рассмотрены все основные (и многие второстепенные) варианты этих структур. Книга [Sam05] — краткий обзор этих структур. Принято считать, что kd-деревья разработал Дж. Бентли (J. Bentley) (см. [Ben75]), но история их появления туманна, как и у большинства популярных структур данных.

Большое количество измерений ведет к падению производительности пространственных структур данных. Недавно был представлен простой, но мощный метод понижения раз-

мерности, заключающийся в отображении многомерных пространств на произвольную гиперплоскость меньшей размерности. Как теоретические, так и экспериментальные результаты (см. [IM04] и [BM01] соответственно) указывают на то, что этот метод сохраняет расстояния довольно хорошо.

Новейшей разработкой в области поиска ближайшего соседа в пространственных структурах высокой размерности являются алгоритмы для быстрого поиска точки, которая находится доказуемо близко к точке запроса. На основе набора данных создается разреженный взвешенный граф, после чего поиск ближайшего соседа осуществляется с применением "жадного" обхода от произвольной точки по направлению к точке запроса. Ближайшим соседом считается точка, найденная в результате нескольких таких операций поиска, начатых от произвольной точки. Подобные структуры данных являются перспективными для решения других задач в пространствах с большим количеством измерений. Подробности см. в [AM93, AMN+98].

Родственные задачи. Поиск ближайшего соседа (см. *раздел 17.5*), определение местоположения точки (см. *раздел 17.7*), поиск в диапазоне (см. *раздел 17.6*).

ГЛАВА 13

Численные задачи

Если вам, в основном, приходится иметь дело с численными задачами, значит, вы читаете не ту книгу. В таком случае вам лучше подойдет книга [PFTV07], в которой дается отличный обзор фундаментальных задач в области численных методов, включая линейную алгебру, численное интегрирование, статистику и дифференциальные уравнения. Соответствующие версии этой книги содержат исходный код для всех рассматриваемых в ней алгоритмов на языке C++, FORTRAN и даже Pascal. Комбинаторно-численные задачи, рассматриваемые в этом разделе, обсуждаются в ней не очень подробно, но, тем не менее, следует знать о существовании данной книги. Дополнительную информацию см. на веб-сайте **http://www.nr.com**.

Между численными и комбинаторными алгоритмами существуют, по крайней мере, два различия:

- *по вопросам точности.* Численные алгоритмы обычно выполняют повторяющиеся вычисления с плавающей точкой, и с каждой операцией накапливается ошибка, пока результаты в конце концов не теряют всякий смысл. Моим любимым примером такого накопления ошибки вычислений является случай с индексом Ванкуверской фондовой биржи, в котором за 22 месяца накопилась достаточно большая ошибка округления, чтобы индекс уменьшился до значения 574,081, в то время как правильное значение должно быть 1098,982.

  Существует простой и надежный способ проверки погрешностей округления в численных программах. Для этого программа исполняется на данных как с одинарной, так и с двойной точностью, после чего результаты сравниваются;

- *по наличию библиотек кода.* Начиная с 1960-х годов, для численных алгоритмов создано большое количество высококачественных библиотек процедур, каких еще нет для комбинаторных алгоритмов. Причин этого несколько, включая:

  - раннее становление языка FORTRAN в качестве стандарта для численных расчетов;
  - независимость численных расчетов от приложений, в которых они применяются;
  - существование крупных научных сообществ, нуждающихся в библиотеках общего назначения для решения численных задач.

  Скорее всего, у вас нет никаких причин для самостоятельной реализации алгоритмов для решения любой задачи, описанной в этом разделе. Я рекомендую начинать поиски готового кода с библиотеки NetLib (см. *раздел 19.1.5*).

Представления многих ученых и инженеров об алгоритмах основаны на методах программирования и численных методах языка FORTRAN. А программисты изучают в студенческие годы работу с указателями и применение рекурсии, поэтому они чувст-

вуют себя свободно с более сложными структурами данных, используемыми в комбинаторных алгоритмах. Оба эти сообщества могут и должны учиться друг у друга, поскольку для моделирования таких задач, как распознавание образов, можно применять как численный, так и комбинаторный подход.

Существует множество книг, посвященных численным алгоритмам. В частности, в дополнение к уже упомянутой выше книге я рекомендую для ознакомления и такие:

- [CC05] — современный лидер продаж среди руководств по численному анализу;
- [Mak02] — это хорошо написанное учебное пособие, в котором язык Java вводится в мир численных расчетов. В книгу включен исходный код примеров;
- [Ham87] — это старое, но не устаревшее, пособие содержит четкое и понятное рассмотрение фундаментальных методов численных расчетов;
- [SK00] — интересное обсуждение базовых численных методов, в котором нет слишком подробного описания алгоритмов благодаря использованию системы вычислительной алгебры Mathematica. На мой взгляд, это очень хорошая книга;
- [CK07] — традиционный учебник по численному анализу, написанный с использованием языка FORTRAN и включающий в себя обсуждение методов оптимизации и метода Монте-Карло в дополнение к таким традиционным темам, как извлечение корня, численное интегрирование, системы линейных уравнений, сплайны и дифференциальные уравнения;
- [BT92] — всестороннее, не привязанное ни к какому конкретному языку программирования, обсуждение всех стандартных тем, включая параллельные алгоритмы. Это наиболее полное изо всех упомянутых здесь руководств.

13.1. Решение системы линейных уравнений

Вход. Матрица A размером $m \times n$ и вектор b размером $m \times 1$, совместно представляющие m линейных уравнений с n переменными (рис. 13.1).

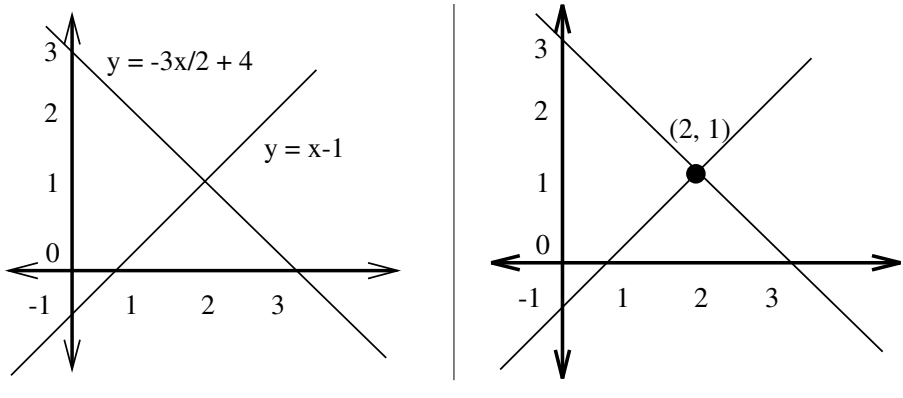


Рис. 13.1. Решение системы линейных уравнений

Задача. Найти такой вектор x, для которого $A \cdot x = b$.

Обсуждение. Потребность в решении систем линейных уравнений возникает в приблизительно 75% всех научных вычислительных задач (см. [DB74]). Например, применение закона Кирхгофа для анализа электрических схем порождает систему уравнений, решение которой определяет прохождение тока через каждую ветвь схемы. При анализе сил, действующих на натянутый трос, возникает аналогичная система уравнений. Даже задача поиска точки пересечения двух или больше линий сводится к решению небольшой системы линейных уравнений.

Но не все системы уравнений имеют решения, например:

$$\begin{cases} 2x + 3y = 5 \\ 2x + 3y = 6 \end{cases}$$

А некоторые системы уравнений, наоборот, имеют несколько решений, например:

$$\begin{cases} 2x + 3y = 5 \\ 4x + 6y = 10 \end{cases}$$

Такие системы уравнений называются *вырожденными*, и их можно распознать, проверив определитель матрицы коэффициентов на равенство нулю.

Задача решения систем линейных уравнений настолько важна в научных и экономических приложениях, что для ее решения существует достаточное количество превосходных программных реализаций. Нет никакого смысла создавать собственную реализацию, хотя базовый алгоритм (метод Гаусса) и изучается в старших классах средней школы. Особенно это справедливо в отношении систем с большим количеством неизвестных.

Метод Гаусса основан на том обстоятельстве, что при умножении уравнения на константу его решение не меняется (если $x = y$, то $2x = 2y$) и при сложении уравнений системы ее решение не меняется (т. е. решение системы уравнений $x = y$ и $w = z$ такое же, как и решение системы $x = y$ и $x + w = y + z$). При решении методом Гаусса уравнения умножаются на константу и складываются так, чтобы можно было последовательно исключать переменные из уравнений и привести систему к ступенчатому виду.

Временная сложность метода Гаусса для системы из $n \times n$ уравнений равна $O(n^3)$, т. к. для того, чтобы исключить i-ю переменную, мы добавляем умноженную на константу копию n-го члена i-й строки к каждому из оставшихся $n - 1$ уравнений. В этой задаче константы играют важную роль. Алгоритмы, которые для получения решения выполняют только частичную обработку матрицы коэффициентов, а затем производят обратную замену, используют на 50% меньше операций с плавающей точкой, чем простые алгоритмы.

Прежде чем приступать к решению системы линейных уравнений, вам нужно ответить на следующие вопросы:

- *Влияют ли на решение ошибки округления?* Реализация метода Гаусса была бы совсем простым делом, если бы не ошибки округления. Они накапливаются с каждой операцией и вскоре могут полностью исказить решение, особенно для *почти* вырожденных матриц.

Чтобы устранить влияние ошибок округления, следует подставлять решение в каждое из первоначальных уравнений и проверять, насколько оно близко к искомому значению. Повысить точность решений систем линейных уравнений можно с помощью *итерационных методов* решения. Хорошие пакеты для решения систем линейных уравнений должны содержать такие процедуры.

Ключом к минимизации ошибок округления в решениях методом Гаусса является выбор правильных опорных уравнений и переменных и умножение уравнений на константы для исключения больших множителей. Это особое искусство, и вам следует использовать тщательно разработанные библиотечные процедуры, описанные ниже.

- *Какую библиотечную процедуру нужно использовать?* Умение выбрать правильный код также является своего рода искусством. Если вы следуете советам из этой книги, то начните с неспециализированных пакетов для решения систем линейных уравнений. Очень вероятно, что их будет достаточно для ваших задач. Однако вам следует поискать в руководстве пользователя эффективные процедуры для решения особых типов систем линейных уравнений. Если окажется, что ваша матрица принадлежит к одному из специальных типов, поддерживаемых пакетом, то время ее решения может быть уменьшено с кубического до квадратичного или даже линейногно.
- *Является ли решаемая система разреженной?* Если в матрице A окажется небольшое количество ненулевых элементов, то матрица является *разреженной* (sparse), и нам повезло. Если эти несколько ненулевых элементов сосредоточены возле диагонали, то матрица является *ленточной* (banded), и нам повезло еще больше. Алгоритмы для уменьшения ширины ленты матрицы рассматриваются в *разделе 13.2*. Для решения можно воспользоваться многими другими стандартными образцами разреженных матриц, подробности о которых можно узнать в руководстве пользователя выбранного вами программного пакета или в книге, целиком посвященной численному анализу.
- *Будет ли использоваться одна и та же матрица коэффициентов для решения нескольких систем?* В таких приложениях, как аппроксимация кривой методом наименьших квадратов, уравнение $A \cdot x = b$ требуется решать несколько раз с разными векторами b. Чтобы облегчить этот процесс, матрицу A можно подвергнуть предварительной обработке. *LU-разложение* — это представление матрицы A с помощью нижней (L) и верхней (U) треугольных матриц, для которых $L \cdot U = A$. LU-разложение можно использовать для решения уравнения $A \cdot x = b$, т. к. $A \cdot x = (L \cdot U) \cdot x = L \cdot (U \cdot x) = b$.

  Это эффективное решение, т. к. обратная подстановка позволяет решить треугольную систему уравнений за квадратичное время. После выполнения LU-разложения за время $O(n^3)$ решение уравнения $L \cdot y = b$, а потом уравнения $U \cdot x = y$ дает нам решение для x за два шага с временем исполнения каждого $O(n^2)$, вместо одного шага с временем исполнения $O(n^3)$.

Задача решения системы линейных уравнений эквивалентна задаче обращения матрицы, т. к. $Ax = B \leftrightarrow A^{-1}Ax = A^{-1}B$, где I — единичная матрица ($I = A^{-1}A$). Но этого подхо-

да следует избегать, т. к. он в три раза медленнее, чем метод Гаусса. LU-разложение оказывается полезным как для обращения матриц, так и для вычисления определителей (см. *раздел 13.4*).

Реализации. Самой лучшей библиотекой для решения систем линейных уравнений повидимому является библиотека LAPACK, более поздняя версия библиотеки LINPACK (см. [DMBS79]). Обе эти библиотеки, написанные на языке FORTRAN, являются частью библиотеки NetLib. Подробности см. в *разделе 19.1.5*.

Версии библиотеки LAPACK имеются и для других языков, в частности для C — CLAPACK и для C++ — LAPACK++. Для этих процедур имеется интерфейс на языке C++, Template Numerical Toolkits, который можно загрузить с веб-сайта **http://math.nist.gov/tnt**.

Универсальная библиотека для научных расчетов JScience содержит обширный пакет инструментов для линейной алгебры (включая определители). Еще одним пакетом для работы с матрицами на языке Java является библиотека JAMA. Ссылки на эти и другие подобные библиотеки см. на сайте **http://math.nist.gov/javanumerics**.

Еще одним источником руководства и процедур для решения систем линейных уравнений является книга [PFTV07] (**www.nr.com**). Наиболее убедительной причиной для использования этих реализаций вместо бесплатных будет отсутствие у разработчика уверенности при работе с численными методами.

Примечания

Стандартным справочником по алгоритмам для работы с системами линейных уравнений является книга [GL96]. Хорошие описания алгоритмов для метода Гаусса и•LU-разложения можно найти в книге [CLRS01]. Существует множество пособий 110 численному анализу, таких как [BT92], [CK07], [SKOO]. Обзор структур данных для систем линейных уравнений представлен в книге [PT05].

Параллельные алгоритмы для систем линейных уравнений обсуждаются в публикациях [Gal90], [KSV97] и [Ort88]. Решение систем линейных уравнений является одним из наиболее важных приложений, в которых на практике широко применяются параллельные архитектуры.

Обращение матриц и, соответственно, решение систем линейных уравнений можно выполнить за время, требуемое для умножения матриц, с помощью алгоритма Штрассена с последующим сведением. Среди хороших описаний равноценных задач можно назвать [AHU74) и [CLRS01].

Родственные задачи. Умножение матриц (см. *раздел 13.3*), определители и перманенты (см. раздел. 13. 4).

13.2. Уменьшение ширины ленты матрицы

Вход. Граф $G = (V, E)$, представляющий матрицу M размером $n \times n$.

Задача. Найти такую перестановку вершин p, которая минимизирует самое длинное ребро, когда вершины выстроены в линию, т. е. минимизирует $\max_{(i,j) \in E} |p(i) - p(j)|$ (рис. 13.2).

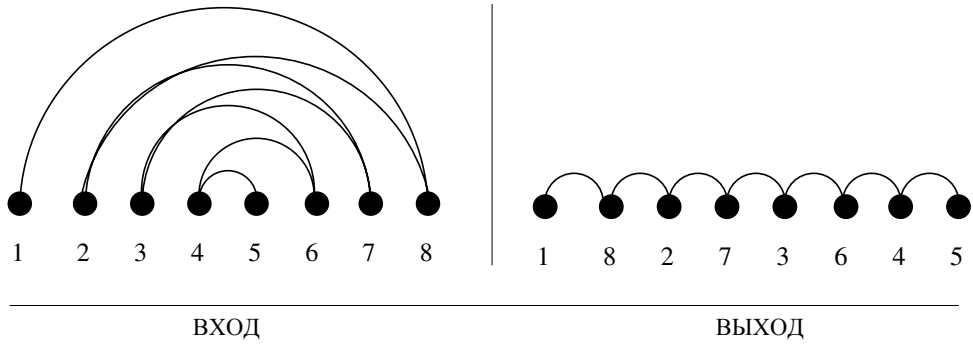


Рис. 13.2. Уменьшение ширины ленты матрицы

Обсуждение. Уменьшение ширины ленты представляет собой незаметную на первый взгляд, но очень важную задачу, как для графов, так и для матриц. Применительно к матрицам, уменьшение ширины ленты заключается в перестановке строк и столбцов разреженной матрицы, чтобы минимизировать расстояние b любого ненулевого элемента от центральной диагонали. Это важная операция в решении систем линейных уравнений, т. к. на матрицах с лентой шириной b метод Гаусса исполняется за время $O(nb^2)$, что является большим улучшением общего времени исполнения $O(n^3)$ алгоритма, когда $b \ll n$.

Уменьшение ширины ленты в графах проявляется не столь очевидным образом. В качестве одного примера задачи уменьшения ширины ленты в графах можно привести задачу упорядочивания n элементов электрической схемы в линию таким образом, чтобы минимизировать длину самого длинного проводника (и таким образом минимизировать временную задержку). В данном случае каждая вершина графа представляет элемент схемы, а каждое ребро — проводник, соединяющий два элемента. Рассмотрим другой пример: гипертекстовое приложение, где нам нужно сохранять большие объекты (скажем, изображения) на магнитной ленте. Каждое изображение имеет набор указателей на набор изображений, к которым можно обратиться на следующем шаге (т. е. гиперссылки). Мы хотим поместить связанные изображения на магнитной ленте как можно ближе друг к другу, чтобы минимизировать время поиска. Это как раз и является задачей уменьшения ширины ленты. Более общие задачи, такие как компоновка прямоугольных схем и магнитные диски, наследуют сложность и эвристические классы линейных версий.

В задаче уменьшения ширины ленты требуется упорядочить вершины в линию таким образом, чтобы минимизировать длину самого длинного ребра, но существует несколько вариантов задачи. В задаче *линейного расположения* (linear arrangement) требуется минимизировать сумму длин всех ребер. Эту задачу можно применить для компоновки схем, в которой требуется расположить микросхемы таким образом, чтобы минимизировать общую длину проводников. В задаче *минимизации профиля* требуется минимизировать сумму длин однонаправленных ребер (т. е. для каждой вершины v минимизировать длину самого большого ребра, вторая вершина которого находится слева от вершины v).

К сожалению, задача уменьшения ширины ленты во всех ее вариантах является NP-полной. Она остается NP-полной, даже если графом входа является дерево с мак-

симальной степенью вершины, равной 3, что является самым строгим ограничением, которое я когда-либо встречал. Поэтому единственными способами ее решения являются метод полного перебора решений и эвристический подход.

К счастью, эвристические методы специального назначения хорошо изучены и для наилучших из них имеются высококачественные реализации. В основе этих реализаций лежит обход в ширину, начинающийся с определенной вершины v, где эта вершина помещается в самую левую позицию упорядочения. Все вершины на расстоянии 1 от вершины v помещаются непосредственно справа от нее, за ними следуют вершины на расстоянии 2 и т. д., пока не будут обработаны все вершины графа. Популярные эвристические алгоритмы различаются по количеству рассматриваемых начальных вершин и по способу упорядочивания равноудаленных вершин. В последнем случае хорошим методом, по-видимому, является выбор самой левой из равнозначных вершин низкой степени.

Реализации наиболее популярных эвристических алгоритмов — Катхилла-Макки (Cuthill-McKee) и Гиббса-Пула-Стокмейера (Gibbs-Poole-Stockmeyer) — рассматриваются в *подразделе "Реализации"*. Время исполнения в наихудшем случае для алгоритма Гиббса-Пула-Стокмейера равно $O(n^3)$, что свело бы на нет любую возможную экономию при решении систем линейных уравнений, но его практическая производительность близка к линейной.

Программы полного перебора могут точно найти минимальную полосу ленты посредством поиска с возвратом в наборе из $n!$ возможных перестановок вершин. Пространство поиска можно значительно разредить с помощью отсечения; для этого нужно начать с хорошего эвристического решения задачи уменьшения ширины ленты и поочередно добавлять вершины в крайние слева и справа свободные позиции в частичной перестановке.

Реализации. Код, написанный Дель Корсо (Del Corso) и Манзини (Manzini), для поиска точных решений задач минимизации ширины ленты (см. [CM99]) можно загрузить с веб-сайта **http://www.mfn.unipmn.it/~manzini/bandmin**. Улучшенные способы решения задачи на основе целочисленного программирования были разработаны Капрарой (Caprara) и Салазар-Гонзалесом (Salazar-Gonzalez), см. [CSG05]. Их реализацию алгоритма метода ветвей и границ на языке C можно загрузить с сайта **http://joc.pubs.informs.org/Supplements/Caprara-2**.

Библиотека NetLib (см. *раздел 19.1.5*) содержит реализации на языке FORTRAN алгоритмов Катхилла–Макки (см. [CGPS76], [Gib76], [CM69]) и Гиббса-Пула-Стокмейера (см. [Lew82], [GPS76]). Обзор экспериментальных оценочных тестов этих и других алгоритмов на наборе из 30 матриц рассматривается в публикации [Eve79b]. Согласно этим тестам, бессменным лидером является алгоритм Гиббса-Пула-Стокмейера.

В докладе [Pet03] предоставляются результаты всестороннего изучения эвристических алгоритмов для решения задачи минимального линейного размещения.

Примечания

Отличный обзор современных алгоритмов решения задачи минимизации ширины ленты и родственных задач на графах представлен в [DPS02]. Обзор решений задачи минимизации ширины ленты для графов и матриц вплоть до 1981 г. можно найти в [CCDG82].

Эвристические алгоритмы специального назначения явились предметом всестороннего изучения, что свидетельствует о их важности в численных расчетах. В докладе [Eve79b] перечислены не менее полусотни различных алгоритмов минимизации ширины ленты. В публикации [CR01] представлены результаты исследований нового класса спектральных эвристических алгоритмов минимизации ширины ленты.

Сложность задачи минимизации ширины ленты была впервые установлена в докладе [Pap76b], а ее сложность на деревьях с максимальной степенью 3 — в докладе [GGJK78]. Для задачи минимизации ленты фиксированной ширины k существуют алгоритмы с полиномиальным временем исполнения (см. [Sax80]). Для общей задачи существуют аппроксимирующие алгоритмы с гарантированным полилогарифмическим временем исполнения (см. [BKRV00]).

Родственные задачи. Решение линейных уравнений (см. *раздел 13.1*), топологическая сортировка (см. *раздел 15.2*).

13.3. Умножение матриц

Вход. Матрица A размером $x \times y$ и матрица B размером $y \times z$.

Задача. Вычислить матрицу $A \times B$ размером $x \times z$ (рис. 13.3).

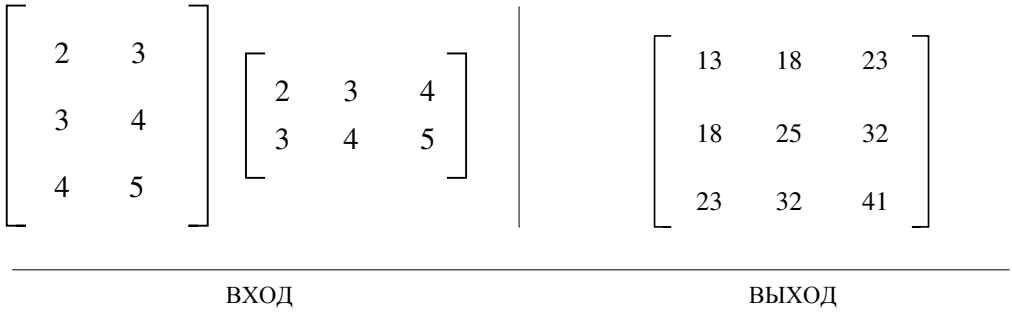


Рис. 13.3. Умножение матриц

Обсуждение. Умножение матриц является фундаментальной задачей линейной алгебры. Значение этой задачи для комбинаторных алгоритмов определяется ее эквивалентностью многим другим задачам, включая транзитивное замыкание и сокращение, синтаксический разбор, решение системы линейных уравнений и обращение матриц. Таким образом, создание быстрого алгоритма умножения матриц влечет за собой появление быстрых алгоритмов для решения всех этих задач. Умножение матриц возникает как самостоятельная задача при вычислении результатов таких координатных преобразований, как масштабирование, вращение и перемещение в робототехнике и компьютерной графике.

В листинге 13.1 представлен псевдокод алгоритма, вычисляющего произведение матрицы A размером $x \times y$ и матрицы B размером $y \times z$ за время $O(xyz)$.

Листинг 13.1. Умножение матриц

```
Инициализируем массив M[i, j] нулями для всех 1 ≤ i ≤ x и i ≤j ≤ z
   for i = 1 to x do
```

```
for j = 1 to z
   for k = 1 to y
      M[i, j] = M[i,j] + A[i, k] — A[k, j]
```

Реализация этого алгоритма на языке C приводится в листинге 2.4 в *разделе 2.5.4*. Казалось бы, что трудно превзойти этот простой алгоритм на практике. Но обратите внимание; что три его цикла можно переставить произвольным образом, не повлияв на конечный результат. После такой перестановки изменятся характеристики обращений к памяти и, вследствие этого, эффективность использования кэша. Можно ожидать, что разброс во времени исполнения между шестью возможными реализациями (перестановками циклов) этого алгоритма достигнет 10-20%, но уверенно предсказать наилучшую из них (обычно *ikj*) нельзя, не выполнив ее на конкретном компьютере с конкретными матрицами.

При умножении матриц с шириной ленты, равной b, когда все ненулевые элементы матриц A и B расположены среди b элементов главных диагоналей, возможно ускорение времени исполнения до $O(xbz)$, т. к. нулевые элементы не участвуют в вычислении произведения матриц.

Использование рекуррентных методов типа "разделяй и властвуй" позволяет получить еще более быстрые алгоритмы для умножения матриц. Но эти алгоритмы трудно программировать, а превзойти тривиальный алгоритм они могут только на очень больших матрицах. Кроме того, они обладают меньшей вычислительной устойчивостью. Наиболее известным из этих алгоритмов является алгоритм Штрассена с временем исполнения $O(n^{2,81})$. Экспериментальные результаты (которые рассматриваются ниже) расходятся в определении критической точки, в которой производительность алгоритма Штрассена начинает превышать производительность простого кубического алгоритма, но можно считать, что это происходит при $n \approx 100$.

Однако при умножении цепочки из более чем двух матриц есть лучший способ сэкономить на вычислениях. Вспомните, что при умножении матрицы размером $x \times y$ на матрицу размером $y \times z$ образуется матрица размером $x \times z$. Таким образом, при умножении цепочки матриц слева направо могут образовываться большие промежуточные матрицы, вычисление каждой из которых занимает много времени. Умножение матриц не обладает перестановочным свойством, но обладает ассоциативным, поэтому элементы цепочки можно заключить в скобки любым удобным способом, что не повлияет на конечный результат умножения. Оптимальный порядок скобок можно создать с помощью стандартного алгоритма динамического программирования. Но такая оптимизация может быть оправданной только в том случае, если матрицы далеки от квадратных, а их перемножение выполняется достаточно часто. Обратите внимание, что оптимизации подвергаются размеры измерений в цепочке, а не сами матрицы. Если же все матрицы одинакового размера, то такая оптимизация невозможна.

Умножение матриц имеет особенно интересную трактовку в задаче подсчета количества путей между двумя вершинами графа. Пусть A представляет матрицу смежности графа G, т. е. $A[i,j] = 1$, если вершины i и j соединены ребром, и $A[i,j] = 0$ в противном случае. Теперь рассмотрим квадрат этой матрицы, $A^2 = A \times A$. Если $A^2[i,j] \geq 1$, то это означает, что должна существовать такая вершина k, для которой $A[i,k] = A[k,j]$, поэтому длина пути от вершины i к вершине k и к вершине j равна 2. В более общем

смысле, $A^k[i, j]$ подсчитывает количество путей длиной ровно k между вершинами i и j. При этом подсчете учитываются непростые пути, т. е. пути с повторяющимися вершинами, например путь, проходящий через последовательность вершин i, k, i и j.

Реализации. В работе [DN07] описывается очень эффективный код для умножения матриц, в котором в оптимальной точке выполняется переключение с алгоритма Штрассена на кубический алгоритм. Эту реализацию можно загрузить с веб-сайта **http://www.ics.uci.edu/~fastmm/**. В предшествующих экспериментах было установлено значение точки пересечения $n \approx 128$, в которой производительность алгоритма Штрассена начинает превосходить производительность кубического алгоритма (см. [BLS91], [CR76]).

Таким образом, кубический алгоритм будет, вероятнее всего, самым лучшим выбором для матриц не очень большого размера. Лучшей библиотекой процедур линейной алгебры является LAPACK, более поздняя версия библиотеки LINPACK (см. [DMBS79]), которая содержит ряд процедур для умножения матриц. Эти коды на языке FORTRAN являются частью библиотеки NetLib (см. *раздел 19.1.5*).

Алгоритм 601 (см. [McN83]) из коллекции алгоритмов ассоциации ACM (**http://calgo.acm.org/**) представляет собой пакет на языке FORTRAN для работы с разреженными матрицами и содержит процедуры для умножения любых комбинаций разреженных и плотных матриц. Подробности см. в *разделе 19.1.5*.

Примечания

Алгоритм Винограда (Winograd) для быстрого умножения матриц уменьшает количество операций умножения наполовину по сравнению с обычным алгоритмом. Хотя этот алгоритм поддается реализации, необходимые дополнительные накладные расходы ставят под сомнение его превосходство. Описания алгоритма Винограда (см. [Win68]) можно найти в [CLRS01], [Man89] и [Win80].

По моему мнению, история теоретической разработки алгоритмов берет начало с публикации алгоритма Штрассена (см. [Str69]) для умножения матриц с временем исполнения $O(n^{2,81})$. Впервые улучшение алгоритма в асимптотическом отношении стало целью, заслуживающей самостоятельного исследования. Последующие улучшения алгоритма Штрассена становились все менее практичными. В настоящее время наилучшим алгоритмом для умножения матриц является алгоритм Копперсмита-Винограда (см. [CW90]) с временем исполнения $O(n^{2,376})$, но высказываются догадки, что можно добиться производительности $\Theta(n^2)$. Описание альтернативного подхода, позволившего получить алгоритм с временем исполнения $O(n^{2,41})$, см. в [CKSU05].

Создание эффективных алгоритмов для умножения матриц требует тщательного управления кэшем. Исследования по этому вопросу см. в [BDN0I] и [HUW02].

Интерес к квадратам графов возник не только в связи с задачей подсчета путей. В работе [Fle74] приведено доказательство, что квадрат любого двусвязного графа содержит гамильтонов цикл. Алгоритмы поиска квадратных корней графов, т. е. вычисление матрицы A по матрице A^2, обсуждаются в [LS95].

Задачу умножения булевых матриц можно свести к общей задаче умножения матриц (см. [CLRS01]). В алгоритме четырех русских (four-Russians algorithm) для умножения булевых матриц (см. [ADKF70]) применяется предварительная обработка с целью создания всех подмножеств $\lg n$ строк для ускорения доступа при выполнении собственно операции умножения, что позволяет получить время исполнения $O(n^3/\lg n)$. Дополнительная предва-

рительная обработка может улучшить это время до $O(n^3/\lg^2 n)$ (см. [Ryt85]). Описание алгоритма четырех русских, включая процедуру дополнительного ускорения, см. в [Man89].

Хорошее описание алгоритма для умножения цепочки матриц представлено в [BvG99] и [CLRS01] в виде стандартного хрестоматийного примера динамического программирования.

Родственные задачи. Решение линейных уравнений (см. *раздел 13.1*), поиск кратчайшего пути (см. *раздел 15.4*).

13.4. Определители и перманенты

Вход. Матрица M размером $n \times n$.

Задача. Найти определитель $|M|$ или перманент $perm(M)$ матрицы m (рис. 13.4).

$$\det \begin{bmatrix} 2 & 1 & 3 \\ 4 & -2 & 10 \\ 5 & -3 & 13 \end{bmatrix}$$

$$2 * \det \begin{bmatrix} -2 & 10 \\ -3 & 13 \end{bmatrix} + -1 * \det \begin{bmatrix} 4 & 10 \\ 5 & 13 \end{bmatrix} + 3 * \det \begin{bmatrix} 4 & -2 \\ 5 & -3 \end{bmatrix} = 0$$

ВХОД ВЫХОД

Рис. 13.4. Вычисление определителя матрицы

Обсуждение. Определители матриц — это понятие из линейной алгебры, которое можно использовать для решения многих задач. В частности, таких как:

- проверка, является ли матрица вырожденной, т. е. проверка существования матрицы, обратной данной. Матрица M является вырожденной, если $|M| = 0$;
- проверка, располагается ли множество из d точек на плоскости в менее, чем d измерениях. Если да, то определяемая ими система уравнений является вырожденной, поэтому $|M| = 0$;
- проверка, находится ли точка справа или слева от линии или плоскости. Эта задача сводится к проверке знака определителя (см. *раздел 17.1*);
- вычисление площади или объема треугольника, четырехгранника или другого многогранника. Эти величины являются функцией значения определителя (см. *раздел 17.1*).

Определитель матрицы M — это сумма всех $n!$ возможных перестановок π, из n столбцов матрицы:

$$|M| = \sum_{i=1}^{n!} (-1)^{sign(\pi_i)} \prod_{j=1}^{n} M[j, \pi_j]$$

где $sign(\pi_i)$ — количество пар элементов, расположенных не по порядку (называемых инверсиями), в перестановке π_i.

Прямая реализация этого определения дает алгоритм с временем исполнения $O(n!)$ точно так же, как и метод алгебраических дополнений. Алгоритмы с лучшей производительностью для определителей основаны на LU-разложении, рассматриваемом в *разделе 13.1*. При этом подходе определитель матрицы M вычисляется просто как произведение диагональных элементов LU-разложения матрицы, для чего требуется время $O(n^3)$.

В комбинаторных задачах часто встречается похожая на определитель функция, называющейся перманентом. Например, перманент матрицы смежности графа подсчитывает количество совершенных паросочетаний графа. Перманент матрицы M вычисляется таким образом:

$$perm(M) = \sum_{i=1}^{n!} \prod_{j=1}^{n} M[j, \pi_j]$$

Единственное отличие перманента от определителя — то, что все произведения положительные.

Удивительно, задача вычисления перманента является NP-полной, хотя определитель можно с легкостью вычислить за время $O(n^3)$. Фундаментальная разница между этими двумя функциями состоит в том, что $det(AB) = det(A) \times det(B)$, в то время как $perm(AB) \neq perm(A) \times perm(B)$. Существуют алгоритмы вычисления перманента за время $O(n^2 2^n)$, что оказывается значительно быстрее, чем время $O(n!)$. Таким образом, вычисление перманента матрицы размером 20×20 является вполне реальной задачей.

Реализации. Пакет инструментов для линейной алгебры UNPACK содержит множество процедур на языке FORTRAN для вычисления определителей. Этот пакет входит в библиотеку NetLib, информацию о которой см. в *разделе 19.1.5*.

Библиотека для научных расчетов JScience содержит обширный пакет инструментов для линейной алгебры (включая определители). Еще одним пакетом для работы с матрицами на языке Java является библиотека JAMA. Ссылки на эти и многие другие библиотеки см. на сайте **http://math.nist.gov/javanumerics**.

Эффективная процедура для вычисления перманентов матриц приводится в книге [NW78]. Подробности см. в *разделе 19.1.10*. В работе [Cas95] описывается процедура на языке C для вычисления перманентов на основе подсчета структур Кекуле (Kekule structures) в вычислительной химии.

Две разных процедуры для аппроксимации перманента разработаны профессором математики университета Мичигана Александром Барвинком. Первый, на основе работы [BS07], предоставляет коды для аппроксимации перманента и гафниана матрицы, а также количества остовных деревьев графа. Подробности см. на веб-сайте **http://www.math.lsa.umich.edu/~barvinok/manual.html**. Второй, на основе работы [SB01], может выдавать приблизительное значение перманента матрицы размером 200×200 за секунды. Подробности см. по адресу **http://www.math.lsa.umich.edu/~barvinok/code.html**.

Примечания

Правило Крамера сводит задачи обращения матриц и решения систем линейных уравнений к задаче вычисления определителей. Но алгоритмы на основе LU-разложения работают быстрее. Описание правила Крамера см. в [BM53].

Определители можно вычислять за время $O(n^3)$, используя метод быстрого умножения матриц, как показано в книге [AHU83]. Такие алгоритмы обсуждаются в *разделе 13.3*. Разработчиком быстрого алгоритма для вычисления знака определителя является Кларксон (Clarkson); см. [Cla92].

В работе [Val79] было доказано, что задача вычисления перманента является #P-полной, где #P означает класс задач, решаемых на "счетной" машине за полиномиальное время. "Счетная" машина возвращает количество разных решений задачи. Подсчет количества гамильтоновых циклов в графе является #P-полной задачей, очевидно NP-сложной (а, возможно, еще сложнее), т. к. любой ненулевой результат подсчета означает, что граф является гамильтоновым. Задачи подсчета могут быть #P-полными, даже если соответствующую задачу разрешимости можно решить за полиномиальное время с использованием перманента и совершенных паросочетаний.

Основным справочником по перманентам является [Min78]. В книге [NW78] представлена разновидность алгоритма для вычисления перманента за время $O(n^2 2^n)$.

За последнее время были разработаны вероятностные алгоритмы для приблизительного вычисления перманента, вершиной развития которых является полиномиальный рандомизированный алгоритм, обеспечивающий сколь угодно точную аппроксимацию за время, полиномиально зависящее от размера матрицы входа и допускаемого уровня погрешности (см. [JSV01]).

Родственные задачи. Решение линейных уравнений (см. *раздел 13.1*), паросочетание (см. *раздел 15.6*), геометрические примитивы (см. *раздел 17.1*).

13.5. Условная и безусловная оптимизация

Вход. Функция $f(x_1, ..., x_n)$.

Задача. Найти точку $p = (p_1, ..., p_n)$, в которой функция f достигает минимума или максимума (рис. 13.5).

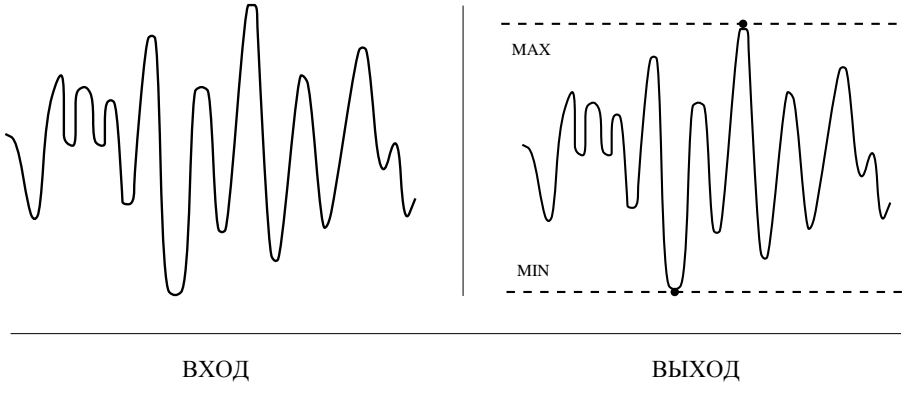


Рис. 13.5. Поиск минимума и максимума функции

Обсуждение. В большей части материала этой книги рассматриваются алгоритмы для оптимизации той или иной величины. В этом разделе мы рассмотрим общую задачу оптимизации функций, для которых по причине отсутствия необходимой структуры или знаний мы не можем применить подогнанные под специфичную задачу алгоритмы, рассматриваемые в других разделах книги.

Задача оптимизации возникает, когда целевую функцию требуется настроить на оптимальную производительность. Допустим, что мы разрабатываем программу для инвестиций на бирже. Здесь мы имеем некоторые финансовые данные, которые можем анализировать, — отношение цены акции к чистой прибыли, процентную ставку и цену акции. Все эти величины являются функцией времени t. Самым сложным здесь является определение коэффициентов следующей формулы:

качество_акций(t) =

$$= c_1 \times \text{цена}(t) + c_2 \times \text{процентная\_ставка}(t) + c_3 \times \text{отношение\_цены\_к\_прибыли}(t)$$

Нам нужно найти такие значения c_1, c_2 и c_3, которые оптимизируют значение функции качества акций. Подобные вопросы возникают при настройке оценочных функций для любой задачи распознавания закономерностей.

Задачи безусловной оптимизации также возникают в научных расчетах. Физические системы, от протеиновых цепочек до галактик, естественно стремятся минимизировать свою "энергию" или "потенциальную функцию". Поэтому в программах, которые пытаются эмулировать природу, потенциальная функция часто определяется путем присваивания некоторого значения каждой возможной конфигурации объектов с последующим выбором из всех этих конфигураций той, у которой потенциал минимален.

Глобальные задачи оптимизации, как правило, сложны, и для их решения существует множество подходов. Чтобы выбрать правильный путь решения таких задач, ответьте на следующие вопросы.

- *Задача какого типа решается (условной или безусловной оптимизации)?* В задачах безусловной оптимизации нет никаких ограничений на значения параметров, кроме условия, чтобы они максимизировали значение функции. Но во многих приложениях требуется накладывать ограничения на эти параметры, вследствие чего некоторые значения, которые в противном случае могли бы дать оптимальное решение, становятся запрещенными. Например, предприятие не может иметь в штате меньше, чем нулевое количество сотрудников, невзирая на то, сколько денег они могли бы сэкономить, имея, скажем, –100 сотрудников. Задачи условной оптимизации обычно требуют подхода математического программирования наподобие линейного программирования, рассматриваемого в *разделе 13.6*.
- *Можно ли оптимизируемую функцию выразить формулой?* Иногда оптимизируемая функция представлена в виде алгебраической формулы, например, требуется найти минимальное значение $f(n) = n^2 - 6n + 2^{n+1}$. В таком случае нужно взять производную этой функции и вычислить, для каких точек p' производная будет равна нулю, т. е. $f'(p') = 0$. Это будут точки локального максимума или минимума, различить которые можно, взяв вторую производную или просто подставив значение p' в функцию f и оценив полученный результат. Такие системы, как Mathematica и Maple, довольно успешно вычисляют производные, хотя эффективное использова-

ние компьютерных приложений вычислительной алгебры в определенной степени сродни шаманству. Тем не менее имеет смысл попробовать эти системы, поскольку с их помощью можно, как минимум, создать диаграмму оптимизируемой функции, позволяющую получить наглядное представление о поведении функции.

- *Насколько затратно вычисление значения функции в определенной точке?* Часто вместо функции приходится иметь дело с программой или процедурой, которая вычисляет значение функции f в данной точке. Мы можем попытаться определить оптимальное значение, проанализировав значения функции в разных точках.

  Возможности нашего поиска зависят от того, насколько эффективно мы вычисляем значение функции. Допустим, что у нас имеется функция $f(x_1, ..., x_n)$ для оценки ситуации на шахматной доске, где x_1 — стоимость пешки, x_2 — стоимость слона и т. д. Чтобы выбрать оптимальные значения набора коэффициентов для этой функции, нам нужно сыграть с ней большое количество игр или протестировать ее на библиотеке известных игр. Очевидно, что такой подход отнимает много времени, поэтому нужно стремиться минимизировать количество вычислений функции, выполняемых при оптимизации коэффициентов.

- *Сколько имеется измерений? Сколько измерений требуется?* Трудность поиска глобального оптимального значения быстро возрастает с увеличением количества измерений (или параметров). По этой причине часто выгодно уменьшить количество измерений, игнорируя некоторые параметры. Это идет против интуиции, и неискушенный программист, скорее всего, вставит в свою оценочную функцию как можно больше переменных. Но оптимизация таких сложных функций является слишком трудной задачей. Намного лучше будет начать с трех до пяти наиболее важных переменных и получить хорошую оптимизацию для них.

- *Насколько гладким является график функции?* Основная опасность, подстерегающая нас при глобальной оптимизации, — это ловушка локального оптимума. Рассмотрим задачу поиска наивысшей точки горного массива. Если данный массив состоит всего лишь из одной горы правильной формы, то наивысшую точку можно найти, просто идя вверх по этой горе. Но если мы имеем дело с горным хребтом, содержащим несколько высоких гор, то найти наивысшую точку будет труднее. *Плавность графика функции* является качеством, позволяющим быстро определить локальный оптимум, двигаясь из данной точки. Когда мы ищем вершину горы, направляясь вверх, мы предполагаем наличие плавности. Если же высота в любой точке определяется полностью случайной функцией, то у нас нет иного способа поиска оптимума, кроме проверки каждой точки.

В наиболее эффективных алгоритмах безусловной глобальной оптимизации для поиска локального оптимума применяются производные и частные производные, позволяющие выяснить направление, в котором следует двигаться из данной точки, чтобы достичь максимального или минимального значения функции. Иногда такие производные можно вычислить аналитическим способом или же значение производной может быть вычислено приблизительно, путем вычисления разности между значениями функции в соседних точках. Для поиска локального оптимума было разработано много разнообразных методов *быстрейшего спуска* (steepest descent) и *сопряженных градиентов* (conjugate gradient), которые во многом подобны числовым алгоритмам поиска корня.

Стоит попробовать несколько разных методов для задачи оптимизации. Прежде чем пытаться реализовать свой собственный метод, поэкспериментируйте с перечисленными далее реализациями. Четкие описания таких алгоритмов можно найти во многих книгах по числовым алгоритмам, в частности, в книге [PFTV07].

В задаче условной оптимизации часто трудно найти точку, удовлетворяющую всем ограничениям. Один из возможных подходов к решению - использование метода без условной оптимизации с добавлением штрафных санкций в зависимости от количества нарушенных ограничивающих условий. Определение правильной функции начисления штрафных "санкций зависит от конкретной задачи, но часто имеет смысл менять штрафные санкции в ходе процесса оптимизации. В конечном счете штрафные санкции должны· быть значительными, чтобы обеспечить удовлетворение всех ограничивающих условий.

Метод имитации отжига является довольно простым и устойчивым к ошибкам подходом к решению задачи условной оптимизации, особенно когда оптимизация выполняется по комбинаторным структурам (перестановкам, графам, подмножествам и т. п.). Технология метода имитации отжига описывается в *разделе 7.5.3*.

Реализации. Предмет безусловной и условной оптимизации является достаточно сложным, вследствие чего было издано несколько руководств в помощь разработчикам. Лучшим из этих руководств является "Decision Tree for Optimization Software", доступное по адресу **http://plato.asu.edu/guide.html**. Также стоит ознакомиться с руководством "Guide to Available Mathematical Software" института NIST, доступным по адресу **http://gams.nist.gov**.

Система NEOS (Network-EnaЬled Optimization System, оптимизирующая система с поддержкой работы в сетевой среде) предоставляет уникальный сервис — возможность решать задачи дистанционно на компьютерах и программном обеспечении Аргоннской национальной лаборатории в США. Поддерживается как линейное программирование, так и безусловная оптимизация. Вместо решения задачи средствами лаборатории можно загрузить программное обеспечение для самостоятельной работы. Подробности см. по адресу **http://www-neos.mcs.anl.gov**.

Коллекция алгоритмов ассоциации ACM содержит несколько реализаций для безусловной оптимизации на языке FORTRAN, из которых особенно заслуживают внимания алгоритм 566 (см. [MGH81]), алгоритм 702 (см. [SF92]) и алгоритм 734 (см. [Buc94]). Алгоритм 744 (см. [Rab95]) из этой же библиотеки реализован на языке Lisp. Все эти алгоритмы являются частью библиотеки NetLib (см. *раздел 19.1.5*).

Также доступны реализации общего назначения метода имитации отжига, которые, скорее всего, будут наилучшей отправной точкой для экспериментов с этой технологией для условной оптимизации. Можете свободно пользоваться моим кодом этого метода (см. *раздел 7.5.3*). Особенно популярной реализацией метода имитации отжига является реализация Adaptive Simulated Annealing на языке C, которую можно загрузить с веб-сайта **http://asa-caltech.sourceforge.net/**.

Как версия Java (**http://jgap.sourceforge.net**), так и версия C (**http://gaul.sourceforge.net**) пакета Genetic Algorithm Utility Library предназначена для оказания помощи в разработке приложений на основе генетических и эволюционных алгоритмов. Лично я скеп-

тически отношусь к генетическим алгоритмам (см. *раздел 7.8*), но многие восхищаются ими.

Примечания

Применение методов быстрейшего спуска для безусловной оптимизации рассматривается в большинстве книг по численным методам, включая [BT92] и [PFTV07]. Сама область безусловной оптимизации также является предметом многих книг, включая [Bre73] и [Fle80].

Метод имитации отжига был разработан в качестве современной версии алгоритма Метрополиса (см. [MRRT53]). В обоих алгоритмах используется метод Монте-Карло для вычисления минимального энергетического уровня системы. Хорошее описание всех разновидностей локального поиска, включая метод имитации отжига, представлено в [AL97].

Генетические алгоритмы были разработаны и популяризированы Холландом (Holland); см. [Hol75] и [Hol92]. Благожелательные описания генетических алгоритмов можно найти в работах [LP02] и [MF00]. Еще одной эвристической процедурой, имеющей преданных последователей, является алгоритм поиска с запретами (tabu search); см. [Glo90].

Родственные задачи. Линейное программирование (см. *раздел 13.6*), выполнимость (см. *раздел 14.10*).

13.6. Линейное программирование

Вход. Набор S из n линейных неравенств с m переменными

$$S_i := \sum_{j=1}^{m} c_{ij} \cdot x_j \geq b_i,\ 1 \leq i \leq n$$

и функция линейной оптимизации $f(X) = \sum_{j=1}^{m} c_j \cdot x_j$.

Задача. Найти такой набор переменных X', который максимизирует целевую функцию f, при этом выполняя все неравенства S (рис. 13.6).

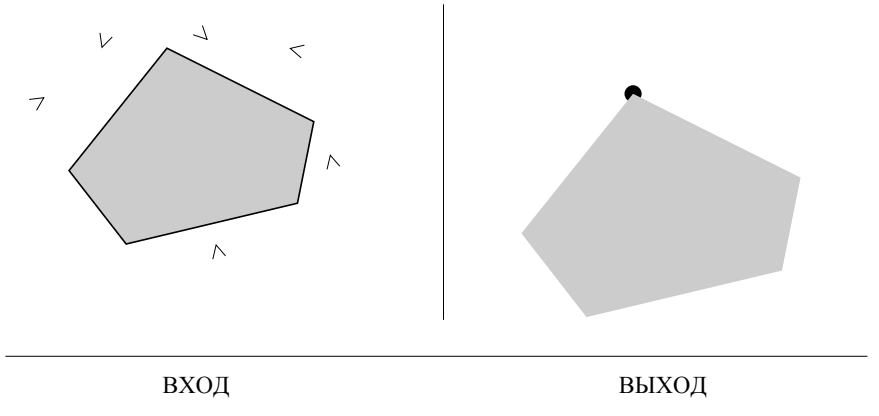


Рис. 13.6. Графическое представление задачи линейного программирования

Обсуждение. Задачи линейного программирования представляют важнейший тип задач из области математической оптимизации и исследования операций.

Линейное программирование имеет следующие приложения:

- *распределение ресурсов.* В данной задаче требуется вложить определенную сумму, чтобы максимизировать полученную прибыль. Часто возможные опции капиталовложения, выплаты и расходы можно выразить в виде системы линейных неравенств таким образом, чтобы максимизировать потенциальный доход при указанных ограничивающих условиях. Крупномасштабные задачи линейного программирования постоянно возникают в авиакомпаниях и других корпорациях;
- *аппроксимация решения несовместных систем уравнений.* Система m линейных уравнений с n неизвестными x_i, $1 \le i \le n$, называется переопределенной (overdetermined), если $m > n$. Переопределенные системы часто являются *несовместными* (inconsistent), т. е. не существует набора значений, которые одновременно удовлетворяют всем уравнениям. Чтобы найти набор значений, наиболее подходящий для решения уравнений, мы можем заменить каждую переменную x_i выражением $x'_i + \varepsilon_i$ и решить новую систему, минимизируя сумму элементов вектора ошибок;
- *алгоритмы на графах.* Многие из рассматриваемых в этой книге задач на графах, включая задачу поиска кратчайшего пути, паросочетания в двудольных графах и потоков в сети, можно решить, как частные случаи задачи линейного программирования. Большинство других задач, включая задачу коммивояжера, покрытие множества и задачу о рюкзаке, можно решить посредством *целочисленного линейного программирования.*

Стандартным алгоритмом линейного программирования является *симплекс-метод.* При использовании этого метода каждое ограничивающее условие задачи отсекает часть пространства возможных решений. Наша цель — найти такую точку в оставшемся после отсечения пространстве решений, которое максимизирует (или минимизирует) функцию $f(X)$. Вращая должным образом пространство решений, оптимальную точку можно всегда сделать наивысшей точкой области. Область (симплекс), полученная в результате пересечения линейных ограничений, является выпуклой, поэтому, если только мы не находимся на самом верху, всегда существует соседняя вершина, расположенная выше. Если мы не можем найти более высокую соседнюю точку, значит, мы нашли оптимальное решение.

В то время как симплексный алгоритм не слишком сложен, создание его эффективной реализации, способной решать задачи линейного программирования на больших входных экземплярах, требует значительного мастерства. Входные экземпляры значительных размеров, как правило, обладают разреженностью (т. е. во многих неравенствах имеется малое количество переменных), что влечет за собой необходимость использования сложных структур данных. Необходимо помнить о важности вопросов вычислительной устойчивости и надежности, а также выбора следующей точки для посещения (так называемые *правила выбора переменных* — pivot rules). Кроме этого, существуют сложные *методы внутренней точки* (interior-point methods), в которых переход к следующей точке осуществляется через внутреннюю область симплекса, а не по поверхности. Во многих приложениях эти алгоритмы превосходят алгоритмы симплекс-метода.

Подводя итоги, можно сказать о линейном программировании следующее: будет намного выгоднее использовать существующие реализации инструментов линейного

программирования, нежели пытаться создать свою собственную программу. Кроме того, надежнее приобрести платное решение, чем искать бесплатное в Интернете. Задача линейного программирования настолько важна в экономических приложениях, что коммерческие реализации превосходят бесплатные во всех отношениях.

Перечислим вопросы, возникающие при решении задач линейного программирования.

- *Ограничены ли какие-либо переменные целочисленными значениями?* Даже если согласно вашей модели максимальную прибыль можно получить, отправляя 6,54 рейсов из Нью-Йорка в Вашингтон ежедневно, реализовать это на практике невозможно. Такие переменные, как количество авиарейсов в приведенном примере, часто обладают естественными целочисленными ограничениями. Задача линейного программирования называется *целочисленной*, если все ее переменные обладают целочисленными ограничениями, и смешанной, если такие ограничения наложены не на все переменные, а только на некоторые из них.

  К сожалению, поиск оптимального решения целочисленной или смешанной задачи является NP-полной задачей. Но существуют методы целочисленного программирования, которые работают достаточно хорошо на практике. Например, методы секущей плоскости сначала решают задачу как линейную, после чего добавляют дополнительные ограничивающие условия, чтобы наложить требование целочисленности в окрестности оптимального решения, и затем решают задачу повторно. После достаточного количества итераций оптимальная точка получившейся задачи линейного программирования совпадает с оптимальной точкой первоначальной целочисленной задачи. Как и в большинстве экспоненциальных алгоритмов, время исполнения алгоритмов целочисленного программирования зависит от сложности конкретного экземпляра задачи, поэтому предсказать его невозможно.

- *Чего в задаче больше — переменных или ограничивающих условий?* Любую задачу линейного программирования с m переменными и n неравенствами можно записать в виде эквивалентной *двойственной* задачи с n переменными и m неравенствами. Важно знать это обстоятельство, т. к. между временами исполнения этих двух вариантов процедуры решения может быть значительная разница. По большому счету, задачи линейного программирования, в которых переменных намного больше, чем ограничивающих условий, следует решать прямым способом. Если же количество ограничивающих условий намного превышает количество переменных, то обычно лучше решать двойственную задачу линейного программирования или (что равносильно) применять двойственный симплекс-метод к первоначальной задаче.

- *Как поступить, если функция оптимизации или ограничивающие условия не являются линейными?* Задача аппроксимации кривой методом наименьших квадратов заключается в поиске прямой, для которой сумма квадратов расстояний между каждой заданной точкой и ею минимальна. При постановке этой задачи естественная целевая функция является квадратичной, а не линейной. Хотя для аппроксимации кривой методом наименьших квадратов существуют быстрые алгоритмы, общая задача *квадратичного программирования* является NP-полной.

  Существуют три возможных подхода к решению задачи нелинейного программирования. Самый лучший — смоделировать ее (если это возможно) каким-либо другим способом, как в задаче аппроксимации кривой методом наименьших квадратов.

Также можно попытаться найти специальные реализации для квадратичного программирования. Наконец, можно смоделировать задачу в виде задачи условной или безусловной оптимизации и попытаться решить ее с помощью реализаций, рассматриваемых в *разделе 13.5*.

- *Что делать, если моя модель не совпадает с форматом входа имеющейся у меня реализации решения задачи линейного программирования?* Многие реализации решений задач линейного программирования принимают модели только в так называемом стандартном формате, где все переменные должны быть неотрицательными, целевая функция должна быть минимизирована, а все ограничивающие условия должны быть сформулированы в виде равенств (а не неравенств).

  В этом требовании нет ничего страшного. Известны преобразования для приведения произвольных задач линейного программирования к стандартному формату. В частности, чтобы преобразовать задачу максимизации в задачу минимизации, нужно просто поменять знак коэффициента целевой функции на противоположный. Остальные несоответствия можно исправить, добавляя в модель *фиктивные переменные* (slack variable). Подробности можно найти в любом учебнике по линейному программированию. Кроме того, вопрос можно решить посредством добавления хорошего интерфейса к программной реализации, воспользовавшись каким-либо языком моделирования, например языком AMPC.

Реализации. Очень полезным ресурсом по решению задач линейного программирования является раздел часто задаваемых вопросов (FAQ) Usenet. В частности, здесь можно найти список имеющихся в наличии реализаций с описанием впечатлений от их применения. Дополнительную информацию см. на веб-сайте **http://www-unix.mcs.anl.gov/otc/Guide/faq/**.

Существуют, по крайней мере, три хорошие бесплатные реализации для решения задач линейного программирования. Приложение lp_solve, написанное Майклом Беркелааром (Michel Berkelaar) на языке ANSI C, работает как с целочисленными, так и со смешанными задачами. Загрузить приложение можно с веб-сайта **http://lpsolve.sourceforge.net/5.5/**. Также существует огромное сообщество пользователей этого приложения. Другая библиотека, CLP, для решения задач линейного программирования симплекс-методом, предоставляется проектом Computational Infrastructure for Operations Research и доступна по адресу **http://www.coin-or.org/**. Наконец, пакет GLPK (GNU Linear Programming Kit) предназначен для решения крупномасштабных задач линейного программирования, целочисленного программирования и других родственных задач. Этот пакет можно загрузить с веб-сайта **http://www.gnu.org/software/glpk/**. В последних тестах открытого кода (**http://plato.asu.edu/bench.html**) библиотека CLP была самой быстрой при решении задач линейного программирования, а библиотека lp_solve — на смешанных задачах.

Система NEOS предоставляет возможность решать задачи дистанционно на компьютерах и программном обеспечении Аргоннской национальной лаборатории (Argonne National Laboratory). Поддерживается как линейное программирование, так и безусловная оптимизация. На эту систему стоит обратить внимание, если вам нужна не программа для решения задачи, а только ответ на задачу. Дополнительную информацию см. на веб-сайте **http://www.mcs.anl.gov/home/otc/Server/**.

Алгоритмы 551 (см. [Abd80]) и 552 (см. [BR80]) симплекс-метода на языке FORTRAN из коллекции алгоритмов ассоциации ACM предназначены для решения переопределенных систем линейных уравнений. Подробности см. в *разделе 19.1.5*.

Примечания

Потребность в оптимизации посредством линейного программирования возникла при решении задач материально-технического снабжения во время Второй мировой войны. Симплексный алгоритм был изобретен Джорджем Данцигом (George Danzig) в 1947 г. (см. [Dan63]). А Кли (Klee) и Минти (Minty) доказали, что симплексный алгоритм имеет экспоненциальное время исполнения в наихудшем случае, но является очень эффективным на практике; см. [KM72] .

При сглаживающем анализе измеряется сложность алгоритмов в предположении, что их вход содержит небольшой объем помех. Аккуратно построенные экземпляры наихудших случаев для многих задач не выдерживают такую проверку. Эффективность симплекс-метода в практическом применении была объяснена с помощью сглаживающего анализа Даниелем Шпильманом (Daniel Spielman) и Шанг-Хуа Тенгом (Shang-Hua Teng); см. [ST04]. А недавно был разработан рандомизированный симплексный алгоритм с полиномиальным временем исполнения (см. [KS05b]).

Полиномиальность задач линейного программирования была впервые доказана в 1979 г. посредством эллиптического алгоритма (см. [Kha79]). Алгоритм Кармаркара (Karmarkar) на основе метода внутренней точки (см. [Kar84]) является как теоретическим, так и практическим улучшением эллиптического алгоритма, а также соперником симплексного метода. Хорошие описания симплексного и эллиптического алгоритмов приведены в [Chv83], [Gas03] и [MG06].

Полуопределенное программирование применяется для решения задач оптимизации с переменными симметрических положительных полуопределенных матриц и линейной функцией стоимости и линейными ограничивающими условиями. Важные частные случаи включают в себя задачи линейного программирования и задачи выпуклого квадратичного программирования с выпуклыми квадратичными ограничивающими условиями. Полуопределенное программирование и его использование для решения комбинаторных задач оптимизации обсуждается в публикациях [Goe97] и [VB96].

Задачи линейного программирования являются P-полными с логарифмической сложностью по памяти (см. [DLR79]). Вследствие этого, создать параллельный алгоритм для решения задач класса NC, скорее всего, невозможно. (Задача принадлежит классу NC тогда и только тогда, когда ее можно решить на машине PRAM за полилогарифмическое время, используя полиномиальное количество процессоров.) Любая P-полная задача по сведению с логарифмической сложностью по памяти не может быть членом класса NC, если только не выполняется условие P = NC. Всестороннее рассмотрение теории P-полноты, включающее в себя обширный список P-полных задач, представлено в книге [GHR95].

Родственные задачи. Условная и безусловная оптимизация (см. *раздел 13.5*), потоки в сети (см. *раздел 15.9*).

13.7. Генерирование случайных чисел

Вход. Либо ничего, либо начальное число (seed).

Задача. Сгенерировать последовательность случайных целых чисел (рис. 13.7).

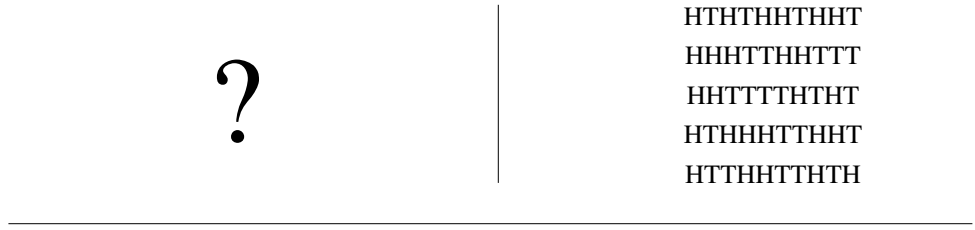


Рис. 13.7. Последовательность случайных символов

Обсуждение. Случайные числа используются в самых разных интересных и важных приложениях. Они лежат в основе метода имитации отжига и родственных эвристических методов оптимизации. Эмуляция дискретных событий выполняется на потоках случайных чисел и используется для моделирования широкого диапазона задач, от транспортных систем до игры в покер. Пароли и ключи шифрования обычно генерируются случайным образом. Рандомизированные алгоритмы для решения задач на графах и геометрических задач коренным образом изменяют эти области и возводят рандомизацию в ранг одного из основополагающих принципов теории вычислительных систем.

К сожалению, задача генерирования случайных чисел выглядит гораздо более легкой, чем она в действительности является. Более того, создать истинно случайное число на любом детерминистическом устройстве, по сути, невозможно. Лучше всего это выразил фон Ньюман (см. [Neu63]): "Любой, кто считает возможным применение арифметических методов для генерирования случайных чисел, несомненно, грешит". Самое лучшее, что мы можем надеяться получить с помощью арифметических методов, это псевдослучайные числа, т. е. последовательность чисел, которые выглядят случайными.

Отсюда вытекают серьезные последствия использования некачественного генератора случайных чисел. В одном получившем известность случае схема шифрования веб-браузера была взломана, т. к. в нем использовалось недостаточно много случайных битов (см. [GW96]). Результаты моделирования постоянно получаются неточными или даже полностью неверными вследствие использования некачественной функции генерирования случайных чисел. Это такая область, в которой не следует заниматься самодеятельностью, но многие обычно переоценивают свои способности.

Перечислим вопросы, касающиеся работы со случайными числами.

- *Можно ли использовать один и тот же набор "случайных" чисел при каждом исполнении программы?* Игра в покер, в которой вам каждый раз сдаются одни и те же карты, быстро надоест. Одним из распространенных решений этой задачи является использование младших битов показаний часов компьютера в качестве начального числа (seed) потока случайных чисел, чтобы при каждом исполнении программы генерировалась другая последовательность.

  Такие методы удовлетворительны для игр, но не для моделирования серьезных ситуаций. Всякий раз, когда вызовы функции генерирования случайных чисел выполняются в цикле, существует вероятность периодичности распределения случайных

чисел. С другой стороны, когда программа выдает разные результаты при каждом ее исполнении, ее отладка серьезно усложняется. В случае сбоя программы вы не сможете отследить ее исполнение, чтобы выяснить причину сбоя. Возможным компромиссом будет использование детерминистического генератора псевдослучайных чисел с сохранением текущего начального числа в файле между исполнениями программы. При отладке в этот файл можно будет записывать фиксированное начальное число.

- *Насколько надежен встроенный генератор случайных чисел моего компилятора?* Если вам требуются равномерно распределенные случайные числа и требования к точности моделирования не являются критическими, то я рекомендую просто использовать генератор, который встроен в компилятор. Здесь очень легко совершить ошибку, неудачно выбрав начальное число, поэтому вы должны внимательно прочитать руководство пользователя.

  Но если точность результатов моделирования *является* критической, то лучше использовать собственный генератор случайных чисел. Но следует иметь в виду, что определить на глазок, действительно ли генерируемые последовательности являются случайными, очень трудно, потому что у людей имеется искаженное представление о поведении генераторов случайных чисел, и они часто видят закономерности, которые в действительности не существуют. Качество генератора случайных чисел следует проверить на нескольких разных тестах и установить статистическую достоверность результатов. Для оценки качества генераторов случайных чисел Национальный институт стандартов и технологий США разработал специальный набор тестов, который рассматривается в *подразделе "Реализации"*.

- *Как реализовать собственный генератор случайных чисел?* Стандартным вариантом реализации генератора случайных чисел является *линейный конгруэнтный генератор* (linear congruential generator). Это простой, быстрый генератор, который дает достаточно удовлетворительные псевдослучайные числа (при условии установления правильных значений констант). Случайное число R_n является функцией $(n-1)$-го сгенерированного случайного числа:

$$R_n = (aR_{n-1} + c) \bmod m$$

  В теории поведение линейного конгруэнтного генератора аналогично поведению рулетки. Длинный путь шарика вдоль окружности колеса (определяемый выражением $aR_{n-1} + c$) заканчивается в одной из немногочисленных лунок. Этот "выбор" весьма чувствителен к длине пути (усеченной с помощью выражения mod m).

  Для выбора значений констант a, c, m и R_0 была разработана специальная теория. Длина периода в значительной степени является функцией от m, значение которого обычно ограничено длиной слова конкретного компьютера.

  Обратите внимание на то, что последовательность чисел, выдаваемая линейным конгруэнтным генератором, начинает повторяться, как только повторяется первое число. Кроме этого, современные компьютеры обладают достаточно высокой скоростью, позволяющей им вызывать функцию генератора 2^{32} раз за несколько минут. Таким образом, для любого линейного конгруэнтного генератора на компьютере с длиной слова в 32 бита существует опасность циклического повторения значений,

что диктует необходимость в генераторах, имеющих значительно более длинные периоды.

- *Что делать, если распределение генерируемых случайных чисел должно быть неравномерным?* Генерирование случайных чисел согласно данной функции неравномерного распределения может быть нелегкой задачей. Самым надежным будет *метод выборки с отклонениями* (acceptance-rejection method). Геометрическая область, из которой требуется делать выборку, заключается в ограничивающее окно, а потом выбирается произвольная точка *p*. Это точку можно получить, независимо генерируя произвольные значения ее координат *x* и *y*. Если полученная точка лежит в интересующей нас области, то мы принимаем ее как случайно выбранную. В противном случае мы отказываемся от точки и процесс повторяется. Можно сказать, что мы бросаем дротики с закрытыми глазами и засчитываем только те, которые попадают в цель.

  В то время как этот метод возвращает правильные результаты, его производительность может быть неудовлетворительной. Если по сравнению с ограничивающим окном площадь представляющей интерес области небольшая, то большинство наших дротиков не будут попадать в цель. Описание эффективных алгоритмов для других специальных распределений, включая гауссово распределение, приводится в *подразделе "Реализации"*.

  Но будьте осторожны с изобретением своих собственных методов, т. к. получить правильное распределение вероятностей очень трудно. Например, генерирование полярных координат посредством случайного выбора с равномерным распределением угла между 0 и 2π и смещением от 0 до *r* будет неправильным способом выбора равномерно распределенных точек из круга радиусом *r*. В данном случае половина сгенерированных точек будет располагаться на расстоянии *r*/2 от центра, в то время как здесь должна находиться только одна четвертая часть их количества! Эта разница достаточно большая, чтобы серьезно исказить результат, и в то же самое время довольно тонкая, что ее легко не заметить.

- *Сколько времени нужно выполнять моделирование по методу Монте-Карло, чтобы получить наилучшие результаты?* Чем больше времени выполняется программа моделирования, тем точнее будет аппроксимация предельного распределения. Но это происходит только до тех пор, пока не будет превышен *период* (длина цикла) генератора случайных чисел, после чего последовательность случайных чисел начнет повторяться, и дальнейшее исполнение не даст никакой дополнительной информации.

Вместо того чтобы устанавливать максимальное значение периода, стоит выполнить программу моделирования с более коротким периодом, но несколько раз (от 10 до 100) и с разными начальными числами, а потом рассмотреть полученный диапазон результатов. Разброс значений даст вам хорошее представление о степени повторяемости полученных результатов. Таким образом вы сможете избавиться от заблуждения, будто моделирование дает "правильный" ответ.

Реализации. Отличный материал по генерированию случайных чисел и стохастическому моделированию можно найти на веб-сайте **http://random.mat.sbg.ac.at**. Там даются ссылки на научные работы и множество реализаций генераторов случайных чисел.

Параллельное моделирование предъявляет особые требования к генераторам случайных чисел. Например, как можно гарантировать независимость случайных потоков на каждой машине? Одно из решений — воспользоваться объектно-ориентированными генераторами, описанными в работе [LSCK02], с длиной периода около 2^{191}. Реализации этих генераторов на языках C, C++ и Java можно загрузить с веб-сайта **http://www.iro.umontreal.ca/~lecuyer/myftp/streams00**. Для работы на многопроцессорных компьютерах поддерживается генерирование независимых потоков случайных чисел. Другой подход — использовать библиотеку SPRNG (см. [MS00]), которую можно загрузить с веб-сайта **http://sprng.cs.fsu.edu**.

Алгоритмы 488 (см. [Bre74]), 599 (см. [AKD83]) и 712 (см. [Lev92]) на языке FORTRAN из коллекции алгоритмов ассоциации ACM предназначены для генерирования случайных чисел с неравномерным распределением согласно нескольким методам распределения вероятностей, включая нормальное, экспоненциальное и пуассоновское. Все эти алгоритмы являются частью библиотеки NetLib (см. *раздел 19.1.5*).

В Национальном институте стандартов США был разработан широкий набор статистических тестов для проверки генераторов случайных чисел (см. [RSN+01]). Это программное обеспечение и описывающий его доклад можно загрузить с веб-сайта **http://csrc.nist.gov/rng**.

Генераторы действительно случайных чисел используют численные представления какого-либо физического процесса. Услуги по предоставлению случайных чисел, сгенерированных таким образом, предоставляются на веб-сайте **http://www.random.org**. Эти числа генерируются на основе атмосферных шумов и проходят проверку статистическими тестами Национального института стандартов и технологий США. Это удачное решение, если вам требуется небольшое количество действительно случайных чисел, например, для проведения лотереи, а не сам генератор таких чисел.

Примечания

Генерирование случайных чисел очень подробно описано в книге Дональда Кнута [Knu97b]. В его книге излагаются теоретические основы нескольких методов генерирования случайных чисел, включая метод серединных квадратов и регистр сдвига, которые не были рассмотрены здесь. Кроме того, подробно обсуждаются статистические тесты для проверки генераторов случайных чисел.

Информация о последних разработках в области генерирования случайных чисел собрана в книге [Gen04]. Генератор случайных чисел, называемый *вихрем Мерсенна* (Mersenne twister; см. [MN98]), имеет период длиной в $2^{19937} - 1$. Другие современные методы генерирования случайных чисел основаны на рекурсии (см. [Den05] и [PLM06]). Обзор методов генерирования случайных чисел с неравномерным распределением представлен в книге [HLD04], а в книге [PM88] — сравнение практического применения разных генераторов случайных чисел.

В прежние времена, когда компьютеры не были так распространены, в большинстве математических учебников печатались таблицы случайных чисел. Наиболее известным является [RC55], в котором приводится один миллион случайных чисел.

Глубинная взаимосвязь между случайностью, информацией и сжимаемостью исследуется в теории колмогоровской сложности, согласно которой о сложности строки можно судить по ее сжимаемости. Строки истинно случайных символов несжимаемы. В соответствии с этой теорией кажущиеся случайными цифры значения π не могут быть случайными, т. к.

вся последовательность определяется любой программой, реализующей разложение в ряд для π. Всестороннее введение в теорию колмогоровской сложности можно найти в книге [LV97].

Родственные задачи. Условная и безусловная оптимизация (см. *раздел 13.5*), генерирование перестановок (см. *раздел 14.4*), генерирование подмножеств (см. *раздел 14.5*), генерирование разбиений (см. *раздел 14.6*).

13.8. Разложение на множители и проверка чисел на простоту

Вход. Целое число n.

Задача. Определить, является ли целое число n простым, а если нет, то найти его множители (рис. 13.8).

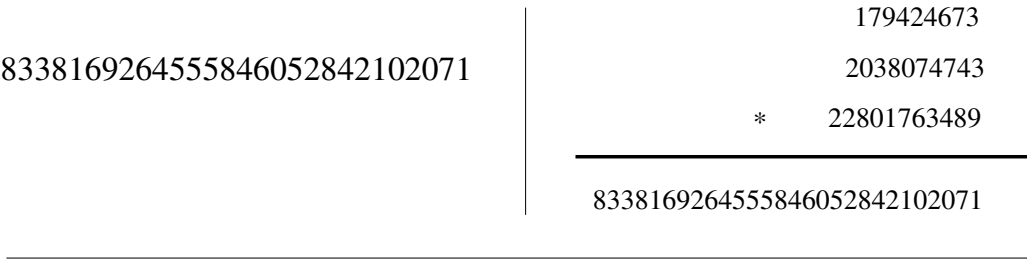


Рис. 13.8. Разложение на множители

Обсуждение. Двойственные друг другу задачи проверки целого числа на простоту и разложения его на множители имеют неожиданно много приложений, если учесть, что долгое время они считались представляющими только математический интерес. В частности, безопасность криптографической системы с открытым ключом RSA (см. *раздел 18.6*) основана на вычислительной неосуществимости решения задачи разложения на множители больших целых чисел. Более скромным приложением задачи проверки целого числа на простоту является улучшение производительности хэш-таблиц путем выбора для них размера, равного простому целому числу. Для этого процедура инициализации хэш-таблицы должна определить простое целое число, близкое к требуемому размеру хэш-таблицы. Наконец, с простыми числами просто интересно экспериментировать. Совсем не случайно, что на UNIX-системах программы для генерирования больших простых чисел часто находятся в папке для игр.

Хотя задача разложения на множители и задача проверки числа на простоту значительно отличаются в алгоритмическом отношении, они являются родственными. Существуют алгоритмы, которые могут показать, что целое число является составным (т. е. не простым), не предоставляя при этом самих множителей. Чтобы убедиться в наличии таких алгоритмов, обратите внимание на тот факт, что можно продемонстрировать составную природу любого нетривиального целого числа, заканчивающегося на 0, 2, 4, 5, 6 или 8, не выполняя для этого самой операции деления.

Самым простым алгоритмом для решения обеих этих задач является проверка делением. Для этого выполняется деление n/i для всех $1 < i \le \sqrt{n}$. Полученные при этом простые множители числа n будут содержать, по крайней мере, один экземпляр такого делителя i, для которого $n / i = \lfloor n / i \rfloor$, если, конечно, n не является простым числом. Но при этом необходимо обеспечить корректную обработку повторяющихся множителей, а также учесть все простые числа, большие $\sqrt{n}$.

Работу таких алгоритмов можно ускорить, используя таблицы заранее вычисленных простых чисел, чтобы не проверять все возможные значения i. Применение битовых векторов (см. *раздел 12.5*) позволяет представить удивительно большое количество простых чисел в неожиданно малом объеме памяти. Для хранения битового вектора, содержащего все нечетные числа до миллиона, требуется меньше, чем 64 Кбайт. Еще более плотную упаковку можно получить, удалив все числа, кратные трем, и другим небольшим простым числам.

Хотя алгоритм проверки делением исполняется за время $O\sqrt{n}$, он *не* является полиномиальным алгоритмом по той причине, что для представления числа n требуется только $\lg_2 n$ битов, поэтому время исполнения алгоритма делением экспоненциально зависит от размера входа. Существуют значительно более быстрые (но с тем же экспоненциальным временем исполнения) алгоритмы, правильность которых основана на теории чисел. Самый быстрый алгоритм, называющийся решетом числового поля *(number field sieve)*, использует случайную последовательность для создания системы конгруэнций, решение которой обычно дает делитель целого числа. С помощью этого метода было выполнено разложение на множители целых чисел длиной в 200 цифр (663 бита), хотя для такой работы потребовалось выполнить громаднейший объем вычислений.

Проверку целых чисел на простоту значительно легче выполнять с помощью рандомизированных алгоритмов. Малая теорема Ферма утверждает, что $a^{n-1} \equiv 1 (\mathrm{mod}\ n)$ для всех a, не делящихся на n, при условии, что n является простым числом. Возьмем случайное значение $1 \le a \le n$ и вычислим остаток $a^{n-1} (\mathrm{mod}\ n)$. Если этот остаток не равен 1, то это доказывает, что n не может быть простым числом. Такие рандомизированные проверки чисел на простоту очень эффективны. С помощью таких проверок программа шифрования PGP (см. *раздел 18.6*) за несколько минут находит простые числа длиной в триста с лишним цифр для использования в качестве ключей шифрования.

Хотя кажется, что простые числа разбросаны среди целых чисел в случайном порядке, их распределение имеет определенную регулярность. Теорема простых чисел утверждает, что количество простых чисел, меньших, чем n (обычно обозначающееся как $\pi(n)$), приблизительно равно $n/\ln n$. Кроме этого, между простыми числами никогда нет больших промежутков, поэтому, вообще говоря, можно ожидать, что для того, чтобы найти первое простое число, большее, чем n, нужно будет проверить $\ln n$ целых чисел. Такое распределение и наличие быстрого рандомизированного теста на простоту объясняет, почему PGP находит большие простые числа с такой скоростью.

Реализации. Существует несколько систем общего назначения для решения задач из вычислительной теории чисел. Система компьютерной алгебры PARI может решать сложные задачи теории чисел с целыми числами произвольной точности (чтобы быть

точным, ограниченными длиной в 80 807 123 цифр на 32-разрядных машинах), а также с действительными, рациональными, сложными и алгебраическими числами и матрицами.

Система написана, в основном, на языке C, а внутренние циклы для основных компьютерных архитектур — на ассемблере, и содержит свыше 200 специализированных математических функций. Систему PARI можно использовать в виде библиотеки, но она также имеет режим калькулятора, предоставляющий немедленный доступ ко всем типам и функциям. Загрузить систему можно с веб-сайта **http://pari.math.u-bordeaux.fr/**.

Библиотека LiDIA на языке C++ (**http://www.cdc.informatik.tu-darmstadt.de/TI/LiDIA/**) реализует несколько современных методов разложения целых чисел на множители.

Высокопроизводительная переносимая библиотека NTL на языке C++ предоставляет структуры данных и алгоритмы для манипулирования целыми числами произвольной длины со знаком, а также для векторов, матриц и многочленов над полем целых чисел и над конечными полями. Библиотеку NTL можно загрузить с веб-сайта **http://www.shoup.net/ntl/**.

Наконец, библиотека MIRACL, написанная на C/C++, реализует шесть разных алгоритмов для разложения целых чисел на множители, включая алгоритм квадратичного решета. Библиотеку MIRACL можно загрузить с веб-сайта **http://www.shamus.ie/**.

Примечания

Книги [CP05] и [Yan03] содержат описание современных алгоритмов проверки целых чисел на простоту и разложения на множители. Более общие сведения по вычислительной теории чисел можно найти в книгах [BS96] и [Sho05].

В 2002 г. Агравал (Agrawal), Каял (Kayal) и Саксена (Saxena) предоставили первый детерминистический алгоритм с полиномиальным временем исполнения, проверяющий, является ли данное целое число составным (см. [AKS04]). При разработке этого незамысловатого алгоритма они выполнили тщательный анализ известных рандомизированных алгоритмов. Существование этого алгоритма является укором исследователям (таким как я), которые побаиваются заниматься классическими нерешенными задачами. Независимая трактовка этого результата приведена в книге [Die04].

Рандомизированный алгоритм Миллера-Рабина (Miller-Rabin algorithm) для проверки (см. [Mil76], [Rab80]) целых чисел на простоту решает проблему чисел Кармайкла (Carmichael numbers), которые являются составными целыми числами, удовлетворяющими условиям теоремы Ферма. Самые лучшие алгоритмы для разложения целых чисел на множители используют метод квадратичного решета (см. [Poni84]) и метод эллиптической кривой (см. [Len87b]).

Механические вычислительные устройства предоставляли самый быстрый способ разложения целых чисел на множители задолго до наступления компьютерной эры. Занимательная история одного из таких устройств, созданного во время Первой мировой войны, изложена в работе [SWM95]. Приводимое в действие вручную вращением рукоятки устройство доказывало простоту числа $2^{31} - 1$ за 15 минут работы.

Важной задачей теории вычислительной сложности является определение истинности выражения $P = NP \cap co\text{-}NP$. Задача разрешимости "является ли n составным числом?" долгое время была наилучшим контрпримером. Предоставляя разложение числа n на множители, она очевидным образом является членом класса NP. Можно доказать, что за-

дача является членом класса co-NP, т. к. для каждого простого числа имеется короткое доказательство его простоты (см. [Pra75]). Но недавнее доказательство, что задача проверки составных чисел является членом класса P, делает эту цепочку рассуждений недействительной. Дополнительную информацию о классах сложности можно найти в [GJ79] и [Joh90].

Число из задачи RSA-129 было разложено на множители в апреле 1994 г. после восьми месяцев вычислений на 1 600 компьютерах. Это событие было особенно примечательным ввиду того обстоятельства, что в исходной работе RSA (см. [RSA78]) предполагалось, что решение этой задачи займет 40 квадрильонов лет (используя технологию 1970-х годов). Текущий рекорд по разложению на множители принадлежит Ф. Бару, М. Боэму, Дж. Франку и Т. Клайнджунгу (F. Bahr, M. Boehm, J. Franke, T. Kleinjung), которые в мае 2005 г. разложили на множители число из задачи RSA-200. Для этого им потребовалось время, эквивалентное 55 годам вычислений на компьютере с одним процессором Opteron 2,2 ГГц.

Родственные задачи. Криптография (см. *раздел 18.6*), арифметические операции высокой точности (см. *раздел 13.9*).

13.9. Арифметика произвольной точности

Вход. Два очень больших целых числа x и y.

Задача. Вычислить $x + y$, $x - y$, $x \times y$ и x/y (рис. 13.9).

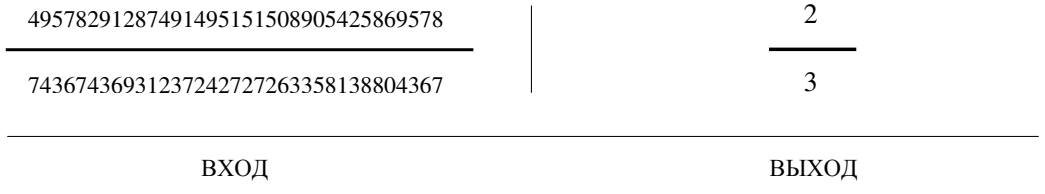


Рис. 13.9. Деление очень больших целых чисел

Обсуждение. Любой язык программирования, имеющий уровень выше ассемблера, поддерживает арифметические операции сложения, вычитания, умножения и деления с целыми и действительными числами с одинарной и, возможно, двойной точностью. А если мы захотим выразить государственный долг США в центах? Для выражения количества центов стоимостью в один триллион долларов потребуется число из 15 десятичных цифр, что намного больше, чем может вместиться в 32-битовое компьютерное слово.

В других приложениях могут возникнуть *намного* бóльшие целые числа. Например, для обеспечения достаточной безопасности в алгоритме RSA для криптографии с открытым ключом рекомендуется использовать целочисленные ключи длиной не менее 1 000 цифр. Исследовательские эксперименты в области теории чисел требуют выполнения операций с большими числами. Однажды я решил незначительную нерешенную задачу (см. [GKP89]), выполнив точные вычисления, результат которых представлен целым числом $\binom{5906}{2953} \approx 9{,}93285 \times 10^{1775}$.

Прежде чем приступить к работе с очень большими целыми числами, ответьте на перечисленные ниже вопросы.

- *Я решаю экземпляр задачи, требующей больших целых чисел, или имею дело со встроенным приложением?* Если вам просто нужен ответ для конкретной задачи с большими целыми числами, как в моем случае с нерешенной задачей, описанном выше, то следует рассмотреть использование системы компьютерной алгебры, такой как Maple или Mathematica. Эти системы по умолчанию предоставляют возможности арифметических вычислений произвольной точности и используют для интерфейса удобные языки программирования наподобие языка Lisp. В таком случае решение вашей задачи будет заключаться в программе из 5–10 строчек.

  Если же вы работаете со встроенным приложением, требующим выполнения арифметических операций с высокой точностью, то следует использовать какую-либо математическую библиотеку, поддерживающую вычисления с произвольной точностью. Эти библиотеки, в дополнение к четырем основным операциям, часто предоставляют дополнительные функции для таких вычислений, как наибольший общий делитель. Подробности см. в *подразделе "Реализации"*.

- *Какой уровень точности требуется?* Иными словами, имеется ли верхняя граница для чисел, с которыми вы работаете, или же требуется произвольная точность представления. От этого зависит, сможете ли вы использовать для представления целых чисел массив фиксированной длины или для этого потребуется применение связных списков. Массив проще для реализации и работы, и в большинстве приложений он не создает никаких ограничений.

- *Какое основание следует использовать?* Возможно, что легче всего будет реализовать свой собственный пакет для выполнения арифметических операций с высокой точностью в десятичной системе, таким образом, представляя каждое целое число в виде строки цифр с основанием 10. Но гораздо эффективнее использовать в качестве основания большое число, в идеале равное квадратному корню из наибольшего целого числа, поддерживаемого аппаратно.

  Почему? Потому, что чем больше основание, тем меньше требуется цифр для представления чисел. Например, сравните количество цифр в одном и том же значении, но представленном в десятичной и двоичной системах счисления — 64 и 1 000 000 соответственно. Так как аппаратная операция сложения обычно выполняется за один тактовый цикл независимо от фактических значений чисел, то наилучшая производительность достигается при использовании максимально возможного основания. Верхняя граница основания определяется значением $b = \sqrt{maxint}$, что позволяет избежать арифметического переполнения при умножении двух таких чисел.

  Основной сложностью использования большого основания является необходимость преобразовывать целые числа в представление с основанием 10 для входа и выхода. Но это преобразование легко выполняется, когда поддерживаются все четыре арифметические операции с высокой точностью.

- *Какой точности будет достаточно?* Сложение аппаратными средствами выполняется намного быстрее, чем программными, поэтому использование арифметических операций с высокой точностью в ситуациях, когда такая точность не требуется,

значительно снижает производительность. Арифметические вычисления с высокой точностью относятся к тем немногочисленным задачам, для которых реализация внутренних циклов на ассемблере оказывается хорошим способом ускорения работы программы. Подобным образом, наложение маски на уровне битов и использование операций сдвига вместо арифметических операций может ускорить выполнение программы; но для этого вы должны хорошо разбираться в особенностях машинного представления целых чисел.

Далее приводятся оптимальные алгоритмы для каждой из пяти основных арифметических операций:

- *сложение.* Простой школьный способ сложения, при котором числа выравниваются по десятичному разделителю, после чего их цифры складываются с переносом справа налево, имеет время исполнения, линейно зависящее от количества цифр. Существуют более сложные параллельные алгоритмы с предсказанием переноса (carry-look-ahead) для реализации низкоуровневого машинного сложения. Скорее всего, они используются в вашем микропроцессоре для выполнения операций сложения с низкой точностью;
- *вычитание.* Изменяя знаковые биты, мы можем превратить операцию вычитания в специальный случай сложения: $(A - (-B)) = (A + B)$. Сложной частью операции вычитания является "заем" из вышестоящего разряда. Для упрощения можно всегда выполнять вычитание из числа с бóльшим абсолютным значением и корректировать знаки позже;
- *умножение.* На больших целых числах выполнение операции умножения повторяющимися операциями сложения занимает экспоненциальное время, поэтому избегайте использования этого метода. Поцифровое умножение школьным методом легко поддается программированию и дает гораздо лучшую производительность, которая, скорее всего, будет достаточной для вашего приложения. Для очень больших целых чисел рекомендуется алгоритм Карацубы (Karatsuba's algorithm) типа "разделяй и властвуй" с временем исполнения $O(n^{1,59})$. Дэн Грейсон (Dan Grayson), разработчик арифметических операций произвольной точности системы компьютерной алгебры Mathematica, обнаружил, что точка перехода к этому алгоритму находится среди чисел, имеющих гораздо менее ста цифр.

  Еще более быстрым является алгоритм на основе преобразований Фурье. Такие алгоритмы рассматриваются в *разделе 13.11*;
- *деление.* Как и в случае с умножением, выполнение деления посредством многократного вычитания занимает экспоненциальное время; поэтому наиболее приемлемым алгоритмом будет "деление в столбик", которому нас учат в школе. Это довольно сложный алгоритм, требующий операции умножения с произвольной точностью и операции вычитания в качестве вызываемых процедур, а также определения правильной цифры в каждой позиции частного методом проб и ошибок.

  Операцию деления целых чисел можно свести к операции умножения, хотя такое сведение далеко не тривиально. Так что, если вы разрабатываете асимптотически быстрое умножение, то результат можно будет использовать и для реализации операции деления;

- *возведение в степень.* Значение a^n можно вычислить, выполнив $n-1$ умножений числа a на само себя, т. е. $a \times a \times ... \times a$. Но существует намного лучший алгоритм типа "разделяй и властвуй", основанный на том обстоятельстве, что $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$.

  Если число n четное, то тогда $a^n = (a^{n/2})^2$. А если n нечетное, то тогда $n^2 = a(a^{\lfloor n/2 \rfloor})^2$. В любом случае, размер показателя степени был уменьшен наполовину, что обошлось нам, самое большее, в две операции умножения. Таким образом, для вычисления конечного значения будет достаточно $O(\lg n)$ операций умножения. Псевдокод соответствующего алгоритма показан в листинге 13.2.

Листинг 13.2. Алгоритм возведения в степень

```
function power(a, n)
   if (n = 0) return(1)
   x = power( a, ⌊n/2⌋ )
   if (n is even) then return(n2)
      else return(a × x2)
```

Арифметические операции высокой, но не произвольной точности, удобно выполнять, используя *китайскую теорему об остатках* и модульную арифметику. Китайская теорема об остатках утверждает, что целое число в диапазоне от 1 до $P = \prod_{i=1}^{k} p_i$ однозначно определяется его набором остатков от деления p_i, где каждая пара p_i, p_j состоит из взаимно простых целых чисел. С помощью таких систем остаточных классов можно выполнять операции сложения, вычитания и умножения (но не деления), причем для манипуляций с большими целыми числами не потребуются сложные структуры данных.

Многие из этих алгоритмов для вычислений с большими целыми числами можно прямо использовать для вычислений с многочленами. Особенно полезным алгоритмом для быстрой оценки многочленов является правило Горнера. Если выражение $P(x) = \sum_{i=0}^{n} c_i \cdot x^i$ вычислять почленно, то нужно будет выполнить $O(n^2)$ операций умножения. Намного лучше будет воспользоваться тем обстоятельством, что $P(x) = c_0 + x(c_1 + x(c_2 + x(c_3 + ...)))$.

Вычисление этого выражения требует только линейного количества операций.

Реализации. Все основные коммерческие системы компьютерной алгебры, включая Maple, Mathematica, Axiom и Macsyma, поддерживают арифметические операции с высокой точностью. Для быстрых результатов в независимом приложении лучшим выбором будет воспользоваться одной из таких систем, если у вас имеется такая возможность. В остальном материале этого подраздела рассматриваются реализации для встроенных приложений.

Ведущей библиотекой на языке C/C++ для быстрого выполнения арифметических операций с высокой точностью является библиотека GMP (GNU Multiple Precision Arithmetic Library, библиотека GNU для арифметических операций с многократно увеличенной точностью), которая поддерживает операции с целыми числами со знаком,

рациональными числами и числами с плавающей запятой. Загрузить ее можно с веб-сайта **http://gmplib.org**.

Класс `BigInteer` пакета java.math предоставляет аналоги операторов с произвольной точностью для всех элементарных операторов Java. Этот класс также предоставляет дополнительные операции для модульной арифметики, вычисления наибольшего общего делителя, проверки на простоту, генерирования простых чисел, манипулирования битами и других операций.

Менее производительная и не так хорошо протестированная моя собственная реализация арифметических операций с высокой точностью содержится в библиотеке из моей книги (см. [SR03]). Подробности см. в *разделе 19.1.10*.

Существует несколько общих систем для вычислительной теории чисел, каждая из которых поддерживает операции с целыми числами с произвольной точностью. Информацию о библиотеках PARI, LiDIA, NTL и MIRACL для работы с задачами теории чисел можно найти в *разделе 13.8*.

Пакет ARPEC предоставляет библиотеку на языке C++ и FORTRAN для арифметических операций с произвольной точностью и сопутствующий интерактивный калькулятор. А пакет MPFUN90 предоставляет подобную функциональность, но исключительно на языке FORTRAN-90. Оба пакета можно загрузить с веб-сайта **http://crd.lbl.gov/~dhbailey/mpdist/**. Алгоритм 693 (см. [Smi91]) из коллекции алгоритмов ACM предоставляет реализацию на языке FORTRAN возможностей арифметических операций с плавающей точкой с многократно увеличенной точностью. Подробности см. в *разделе 19.1.5*.

Примечания

Основным справочником по алгоритмам для всех основных арифметических операций, включая их реализацию на языке ассемблера MIX, является книга Дональда Кнута [Knu97b]. Более современное описание вычислительной теории чисел представлено в книгах [BS96] и [Sho05].

В число работ с описанием алгоритма типа "разделяй и властвуй" для умножения с временем исполнения $O(n^{1.59})$ входят книги [AHU74] и [Man89]. Алгоритм на основе быстрого преобразования Фурье умножает два числа длиной в n бит за время $O(n\lg n\lg\lg n)$ (см. [SS71]). Его описание можно найти, например, в [AHU74] и [Knu97b]. Здесь же имеется описание сведения задачи деления целых чисел к задаче их умножения. Использование быстрого умножения для выполнения других арифметических операций обсуждается в работе [Ber04b].

Хорошие описания алгоритмов модульной арифметики и китайской теоремы об остатках содержатся в [AHU74] и [CLRS01]. А в книге [CLRS01] дано описание алгоритмов для выполнения элементарных арифметических операций.

Самым старым представляющим интерес алгоритмом, вероятно, является алгоритм Евклида для вычисления наибольшего общего делителя двух чисел. Его описание можно найти, например, в [Man89] и [CLRS01].

Родственные задачи. Разложение целых чисел на множители (см. *раздел 13.8*), криптография (см. *раздел 18.6*).

13.10. Задача о рюкзаке

Вход. Множество предметов $S = \{1, ..., n\}$, где размер i-го предмета равен s_i, а значение — v_i. Емкость рюкзака равна C.

Задача. Найти подмножество $S' \subset S$, которое максимизирует значение $\sum_{i \in S'} v_i$, учитывая, что $\sum_{i \in S'} s_i \leq C$, т. е., что все предметы помещаются в рюкзак размером C (рис. 13.10).

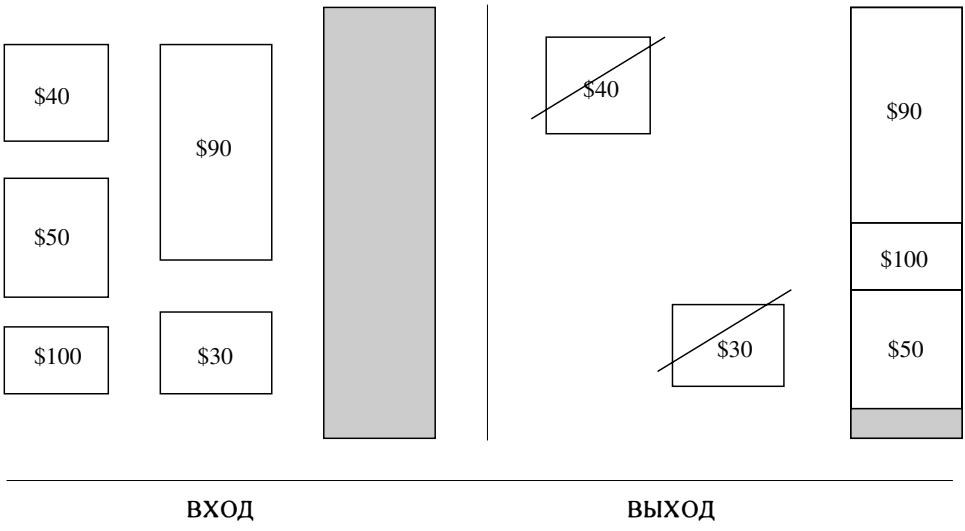


Рис. 13.10. Задача о рюкзаке

Обсуждение. Задача о рюкзаке возникает в ситуациях распределения ресурсов при наличии финансовых ограничений. Например, как выбрать вещи, которые нужно купить на фиксированную сумму? Так как все предметы имеют свою стоимость и значимость, мы стремимся добиться максимальной значимости при данной стоимости. Само название задачи — *задача о рюкзаке* — вызывает в мыслях образ туриста, который из-за ограниченности размера рюкзака старается положить в него лишь самые нужные и компактные вещи.

Самая распространенная постановка задачи имеет ограничение вида "0-1", подразумевающее, что каждый предмет должен быть либо положен в рюкзак целиком, либо не положен вовсе. В большинстве реальных случаев предметы нельзя разламывать на части произвольным образом, поэтому нельзя взять только одну банку лимонада из шестибаночной упаковки или, вообще, часть содержимого одной банки. Вот это ограничение "0-1" и делает задачу о рюкзаке такой сложной, т. к. при возможности разбиения предметов на части оптимальное решение находится "жадным" алгоритмом. Мы просто вычисляем "стоимость килограмма" каждого предмета и вкладываем в рюкзак самый дорогой предмет или наибольшую его часть и повторяем эту операцию с самым дорогим предметом из оставшихся до тех пор, пока не заполним весь рюкзак. Но, к сожалению, в большинстве приложений ограничение "0-1" присутствует.

Перечислим вопросы, которые возникают при выборе наилучшего алгоритма для решения задачи о рюкзаке.

- *Имеют ли все предметы одинаковую стоимость или одинаковый размер?* Когда стоимость всех предметов одинаковая, то, чтобы получить наибольшую стоимость, мы просто берем максимальное количество предметов. Поэтому оптимальное решение — отсортировать все предметы в возрастающем порядке по размеру, после чего вкладывать их в рюкзак в этом порядке до тех пор, пока имеется свободное место. Задача решается подобным образом, когда все предметы одинакового размера, но разной стоимости. Предметы сортируются в возрастающем порядке по стоимости и укладываются в рюкзак в этом порядке. Это легкие частные случаи задачи о рюкзаке.
- *Одинакова ли стоимость килограмма для каждого предмета?* В таком случае мы игнорируем цену и просто пытаемся минимизировать незаполненное пространство рюкзака. К сожалению, даже этот ограниченный случай задачи является NP-полным, поэтому нельзя ожидать найти эффективный алгоритм, который всегда выдает решение. Но не отчаивайтесь, т. к. задача о рюкзаке оказывается "легкой" сложной задачей, которую обычно можно решить с помощью представленных далее алгоритмов.

  Важным частным случаем варианта задачи о рюкзаке с одинаковой стоимостью килограмма каждого предмета является задача *разбиения множества целых чисел*, представленная графически на рис. 13.11.

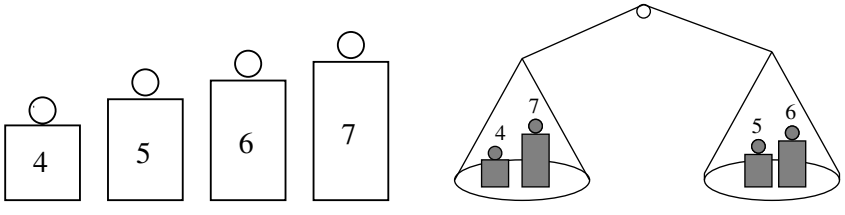


Рис. 13.11. Задача разбиения множества целых чисел

В данном случае нам нужно разделить элементы множества S на два подмножества A и B таким образом, чтобы $\sum_{a \in A} a = \sum_{b \in B} b$, или, хотя бы, чтобы разность была минимальной. Задачу разбиения множества целых чисел можно рассматривать как задачу упаковки двух рюкзаков одинаковой вместимости или одного рюкзака с емкостью, вдвое меньшей, чем исходная, поэтому все три задачи тесно связаны и являются NP-полными.

Вариант задачи с одинаковой стоимостью килограмма для всех элементов часто называется *задачей о сумме подмножества* (subset sum problem), т. к. мы стремимся найти такое подмножество элементов, общая стоимость которых будет равной определенному целевому значению C, т. е. емкости нашего рюкзака.

- *Размеры всех предметов представлены относительно небольшими целыми числами?* Для варианта задачи, когда размеры предметов и емкость рюкзака представлены целыми числами, существует эффективный алгоритм поиска оптимального ре-

шения с временной сложностью $O(nC)$ и сложностью по памяти $O(C)$. Подойдет ли этот алгоритм для решения вашей конкретной задачи, зависит от размера C. Он дает отличные результаты для $C \leq 1\,000$, но не очень хорошие для $C \geq 10\,000\,000$.

Алгоритм работает таким образом. Пусть S' будет набором элементов, а $C[i, S']$ будет иметь значение ИСТИНА тогда и только тогда, когда существует подмножество множества S', общий размер всех элементов которого равен точно i. Отсюда следует, что $C[i, 0]$ имеет значение ЛОЖЬ для всех $1 \leq i \leq C$. Дальше мы по одному добавляем элементы s_j к S' и обновляем затронутые значения $C[i, S']$. Обратите внимание, что $C[i, S' \cup s_j]$ = ИСТИНА тогда и только тогда, когда истинны $C[i, S']$ или $C[i - s_j, S']$, поскольку мы либо используем s_j в получении суммы, либо нет. Мы определяем все суммы, которые можно получить, перебрав n раз все элементы в C — по одному разу для каждого s_j, где $1 \leq j \leq n$, таким образом обновляя массив. Решением задачи является наибольший индекс истинного элемента наибольшего размера. Чтобы воссоздать подмножество, дающее решение, для каждого $1 \leq i \leq C$ мы должны сохранять имя элемента, который изменяет значение $C[i]$ с ЛОЖЬ на ИСТИНА, после чего перебрать элементы массива в обратном направлении.

Эта формулировка в стиле динамического программирования игнорирует значения элементов. Чтобы обобщить этот алгоритм, в каждом элементе массива сохраняется значение наилучшего текущего подмножества, для которого сумма размеров элементов равна i. Теперь обновление выполняется, когда сумма стоимости $C[i - s_j, S']$ и стоимости s_j лучше, чем предыдущая стоимость $C[i]$.

Что делать, если имеется несколько рюкзаков? В таком случае задачу лучше рассматривать, как задачу разложения по контейнерам. Алгоритмы для решения задачи разложения по контейнерам и задачи раскроя (cutting-stock) описаны в *разделе 17.9*. А реализации алгоритмов решения задачи с несколькими рюкзаками рассматриваются в *подразделе "Реализации"*.

Точные решения для рюкзаков большого объема можно найти с помощью целочисленного программирования или поиска с возвратом. Наличие или отсутствие элемента i в оптимальном подмножестве обозначается целочисленной переменной x_i, принимающей значения 0 или 1. Мы максимизируем выражение $\sum_{i=1}^{n} x_i \cdot v_i$ при ограничивающем условии, что $\sum_{i=1}^{n} x_i \cdot s_i \leq C$. Реализации алгоритмов целочисленного программирования рассматриваются в *разделе 13.6*.

Когда получение точного решения оказывается слишком дорогим в вычислительном отношении, то возникает необходимость в использовании эвристических алгоритмов. Простой эвристический "жадный" алгоритм вкладывает предметы в рюкзак согласно ранее рассмотренному правилу максимальной стоимости килограмма. Часто такое эвристическое решение близко к оптимальному, но, в зависимости от конкретного экземпляра задачи, оно также может быть сколь угодно плохим. Правило стоимости килограмма можно использовать, чтобы уменьшить размер задачи в алгоритмах исчерпывающего перебора, чтобы в дальнейшем не рассматривать "дешевые, но тяжелые" предметы.

Другой эвристический алгоритм основан на *масштабировании*. Динамическое программирование хорошо подходит для тех случаев, когда емкость рюкзака выражена

достаточно небольшим целым числом, скажем, C_8. А если нам приходится иметь дело с рюкзаком, емкость которого больше, чем это значение, т. е., $C > C_8$? В таком случае мы уменьшаем размеры всех элементов в C/C_8 раз, округляем полученные размеры до ближайшего целого числа, после чего используем динамическое программирование с этими уменьшенными элементами. Масштабирование хорошо зарекомендовало себя на практике, особенно при небольшом разбросе размеров элементов.

Реализации. Коллекцию реализаций алгоритмов на языке FORTRAN для разных версий задачи о рюкзаке можно загрузить с веб-сайта **http://www.or.deis.unibo.it/kp.html**. Здесь же можно загрузить электронную версию книги [MT90a].

Хорошо организованную коллекцию реализаций алгоритмов на языке C для решения разных видов задачи о рюкзаке и родственных вариантов задачи, таких как разложение по контейнерам и загрузка контейнера, можно загрузить с веб-сайта **http://www.diku.dk/~pisinger/codes.html**. Самый мощный код основан на алгоритме динамического программирования, описание которого дается в работе [MPT99].

Алгоритм 632 на языке FORTRAN из коллекции ACM предназначен для решения задачи о рюкзаке с ограничением вида "0-1". Кроме того, он поддерживает работу с несколькими рюкзаками. Подробности см. в *разделе 19.1.5*.

Примечания

Самым свежим справочником по задаче о рюкзаке и ее вариантах является книга [KPP04]. Книга [MT90a] и обзорная статья [MT87] представляют собой обычные справочные пособия по задаче о рюкзаке, содержащие как теоретические, так и экспериментальные результаты ее решения. Замечательное описание алгоритмов целочисленного программирования для решения задач о рюкзаке представлено в книге [SDK83]. Алгоритмы решения задачи о рюкзаке с ограничением вида "0-1" обсуждаются в работе [MPT00].

Аппроксимирующий метод дает решение, близкое к оптимальному, за время, зависящее полиномиально от размера входа и коэффициента аппроксимации ε.

Такое очень строгое ограничение заставляет искать компромисс между временем исполнения и качеством аппроксимации. Хорошие описания аппроксимирующих методов с полиномиальным временем исполнения для решения задачи о рюкзаке и суммы подмножества можно найти в [IK75], [BvG99], [CLRS0l], [GJ79] и [Man89].

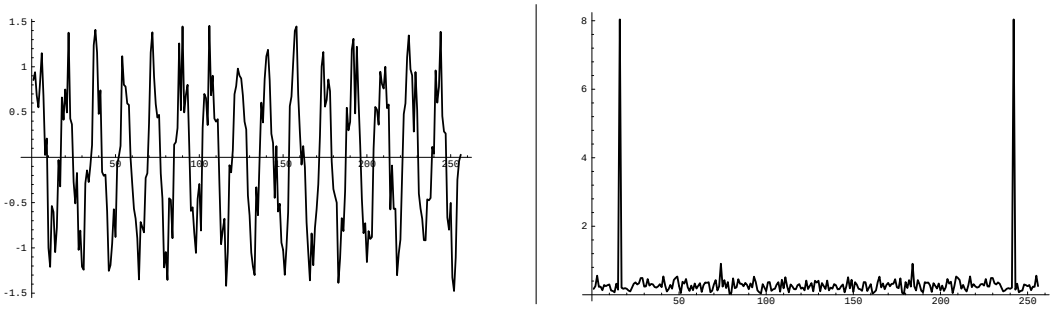
Первый алгоритм для общего метода шифрования с открытым ключом был основан на сложности задачи о рюкзаке. Описание см. в книге [Sch96].

Родственные задачи. Разложение по контейнерам (см. *раздел 17.9*), целочисленное программирование (см. *раздел 13.6*).

13.11. Дискретное преобразование Фурье

Вход. Последовательность из n действительных или комплексных значений h_i функции h, выбранных через одинаковые интервалы.

Задача. Дискретное преобразование Фурье $H_m = \sum_{k=0}^{n-1} h_k e^{2\pi i k m/n}$ для $0 \le m \le n-1$ (рис. 13.12).



ВХОД ВЫХОД

Рис. 13.12. Пример преобразования Фурье

Обсуждение. В то время как программисты обычно плохо разбираются в преобразованиях Фурье, инженеры-электронщики, работающие в области обработки сигналов, обращаются с ними вполне свободно. В функциональном отношении преобразования Фурье предоставляют способ для преобразования выборок стандартного временного ряда в *частотную область*. Таким образом получается двойственное представление функции, при котором определенные операции становятся проще для выполнения. Преобразования Фурье находят следующие применения:

- *фильтрация.* Выполнение преобразования Фурье для функции равносильно представлению ее в виде суммы синусов. Устранив "лишние" высокочастотные и/или низкочастотные компоненты (т. е. исключив некоторые слагаемые) и взяв обратное преобразование Фурье, мы можем отфильтровать шум и другие нежелательные явления. Например, всплеск на графике в рис. 13.12 соответствует периоду одной синусной составляющей и моделирует входные данные. Остальные компоненты являются шумом;
- *сжатие изображений.* Сглаженное, отфильтрованное изображение содержит меньший объем информации, чем исходное изображение, в то же самое время сохраняя похожий внешний вид. Удалив компоненты, вносящие сравнительно небольшой вклад в изображение, мы можем уменьшить размер изображения за счет незначительного понижения его точности;
- *свертка и обращение свертки.* Преобразования Фурье можно использовать для эффективного выполнения свертки двух последовательностей. *Сверткой* (convolution) называется попарное умножение элементов двух разных последовательностей, например, при умножении двух многочленов f и g с n переменными или при сравнении двух текстовых строк. Реализация такой операции напрямую имеет квадратичное время исполнения (т. е. $O(n^2)$), в то время как время исполнения алгоритма на основе быстрого преобразования Фурье равно $O(n\lg n)$.

  Приведем другой пример из области обработки изображений. Так как сканер измеряет уровень освещенности участка изображения, а не отдельной его точки, то отсканированное изображение всегда немного смазано. Исходный сигнал можно восстановить, выполнив обращение свертки входного сигнала посредством гауссовой функции рассеяния точки;

- *вычисление корреляционной функции.* Корреляционная функция двух функций $f(t)$ и $g(t)$ определяется как:

$$z(t) = \int_{-\infty}^{\infty} f(\tau)g(t+\tau)d\tau$$

Эту функцию можно с легкостью вычислить, используя преобразования Фурье. Когда две функции имеют похожую форму, но сдвинуты по отношению друг к другу (например, как функции $f(t) = \sin(t)$ и $g(t) = \cos(t)$), значение $z(t_0)$ будет большим для этого смещения t_0. В качестве примера практического применения допустим, что мы хотим определить, нет ли в нашем генераторе случайных чисел каких-либо нежелательных периодичных последовательностей. Для этого мы можем сгенерировать длинную последовательность случайных чисел, преобразовать их во временную последовательность (в которой i-е число соответствует моменту времени i), после чего взять преобразование Фурье этой последовательности. Любой всплеск будет индикатором возможной периодичности.

Дискретное преобразование Фурье принимает на входе n комплексных чисел h_k, где $0 \le k \le n-1$, соответствующих равномерно распределенным точкам временной последовательности, и выдает на выходе n комплексных чисел H_k, где $0 \le k \le n-1$, каждое из которых описывает синусоидальную функцию данной частоты. Дискретное преобразование Фурье определяется следующей формулой:

$$H_m = \sum_{k=0}^{n-1} h_k e^{-2\pi ikm/n}$$

А обратное преобразование Фурье определяется следующей формулой:

$$h_m = \frac{1}{n}\sum_{k=0}^{n-1} H_k e^{2\pi ikm/n}$$

Таким образом мы можем с легкостью перемещаться между h и H.

Так как выход дискретного преобразования Фурье состоит из n чисел, каждое из которых вычисляется по формуле, содержащей n элементов, то его время исполнения равно $O(n^2)$. А алгоритм быстрого преобразования Фурье вычисляет дискретное преобразование Фурье за время $O(n\log n)$. Вероятно, это самый важный известный алгоритм, т. к. он лежит в основе всей современной обработки сигналов. Под общим названием быстрого преобразования Фурье известно несколько алгоритмов, использующих метод "разделяй и властвуй". По сути, задача вычисления дискретного преобразования Фурье по n точкам сводится к вычислению двух преобразований, каждое по $n/2$ точкам, которые потом применяются рекурсивно.

Алгоритм быстрого преобразования Фурье обычно предполагает, что число n является степенью двойки. Если ваше значение n не отвечает этому условию, то будет лучше дополнить данные нулями, чтобы создать $n = 2^k$ элементов, чем искать более общий код.

Многие системы обработки сигналов имеют строгие условия работы в реальном времени, поэтому алгоритмы быстрого преобразования Фурье часто реализуются аппаратно или, по крайней мере, на языке ассемблера, настроенного под конкретную машину. Учтите это обстоятельство, если используемый вами код окажется слишком медленным.

Реализации. На первом месте среди открытых кодов алгоритма быстрого преобразования Фурье стоит библиотека FFTW. Это библиотека процедур на языке C для вычисления дискретного преобразования Фурье в одном или нескольких измерениях, поддерживающая вход произвольного размера и данные, как действительного, так и комплексного типа. Результаты обширного тестирования доказывают, что это самое быстрое известное преобразование Фурье. Библиотека снабжена интерфейсами для FORTRAN и C++. В 1999 г. библиотеке был присужден приз Дж. Х. Вилкинсона в области численного программного обеспечения (J. H. Wilkinson Prize for Numerical Software). Библиотеку FFTW можно загрузить с веб-сайта **http://www.fftw.org/**.

Пакет FFTPACK содержит процедуры на языке FORTRAN для вычисления быстрого преобразования Фурье периодических и других симметрических последовательностей. Пакет включает комплексные, действительные, синусоидальные и четвертьволновые преобразования. Загрузить его можно с веб-сайта **http://www.netlib.org/fftpack**. Научная библиотека GNU для C/C++ предоставляет версию библиотеки FFTPACK. Подробности см. по адресу **http://www.gnu.org/software/gsl/**.

Алгоритм 545 (см. [Fra79]) из коллекции ACM является реализацией на языке FORTRAN быстрого преобразования Фурье, оптимизирующей производительность виртуальной памяти. Дополнительную информацию см. в *разделе 19.1.5*.

Примечания

Хорошим введением в предмет преобразований Фурье и быстрых преобразований Фурье являются книги [Bra99] и [Bri88]. Описание преобразования можно найти в книге [PFTV07]. Изобретение быстрого преобразования Фурье обычно приписывают Кули (Cooley) и Туки (Tukey); см. [CT65]. Подробную историю вопроса можно найти в [Bri88].

Нечувствительный к кэшированию алгоритм быстрого преобразования Фурье приведен в докладе [FLPR99]. Само понятие нечувствительных к кэшированию алгоритмов было впервые предложено в этом докладе. Библиотека FFTW основана на этом алгоритме. Подробности устройства библиотеки FFTW см. в [FJ05].

Интересный алгоритм типа "разделяй и властвуй" для умножения многочленов (см. [KO63]) с временем исполнения $O(n^{1,59})$ рассматривается в книгах [AHU74] и [Man89]. Алгоритм на основе быстрого преобразования Фурье, умножающий два числа длиной в n бит за время $O(n\lg n\lg\lg n)$, был разработан Шонхаге (Schonhage) и Штрассеном (Strassen). Он представлен в [SS71] и [AHU74].

Вопрос о том, действительно ли комплексные переменные являются необходимыми в быстрых алгоритмах для выполнения свертки, является открытым. К счастью, в большинстве приложений быструю свертку можно использовать в виде черного ящика. Быстрая свертка лежит в основе многих разновидностей алгоритмов для сравнения строк (см. [Ind98]).

В последнее время было предложено вместо преобразования Фурье использовать в фильтрации специальные функции, называемые *вейвлетами*. Введение в эту тему можно найти в [Wal99].

Родственные задачи. Сжатие данных (см. *раздел 18.5*), арифметические операции с высокой точностью (см. *раздел 13.9*).

ГЛАВА 14

Комбинаторные задачи

В этой главе мы рассмотрим несколько алгоритмических задач чисто комбинаторного характера. В число этих задач входят задача сортировки и задача генерирования перестановок, которые были первыми нечисловыми задачами, решенными с помощью электронных вычислительных машин. Сортировку можно рассматривать, как полное упорядочивание ключей, а поиск и выбор — как идентификацию ключей по их положению в упорядоченной последовательности.

Здесь также рассматриваются другие комбинаторные объекты, такие как перестановки, разбиения, подмножества, календари и расписания. Особый интерес представляют алгоритмы, которые ранжируют комбинаторные объекты, т. е. устанавливают соответствие между объектом и уникальным целым числом, или выполняют обратную операцию, возвращая объект по числу. Операции ранжирования упрощают многие другие задачи, например генерирование случайных объектов (выбирается произвольное число и выполняется операция, обратная ранжированию) или перечисления всех объектов по порядку (в цикле генерируются числа от 1 до n, для каждого из которых выполняется операция, обратная ранжированию).

В конце главы обсуждается задача генерирования графов. Алгоритмы на графах представлены более подробно в последующих разделах каталога задач.

Среди книг по общим комбинаторным алгоритмам я порекомендую следующие:

- [NW78] — эта книга посвящена алгоритмам создания элементарных комбинаторных объектов, таких как перестановки, подмножества и разбиения. Такие алгоритмы, как правило, трудно найти, и они нередко имеют много тонкостей. Для всех алгоритмов предоставляются программы на языке FORTRAN, а также обсуждаются их теоретические основы. Подробности см. в *разделе 19.1.10*;
- [KS99] — книга по комбинаторным алгоритмам; кроме этого, особое внимание уделяется алгебраическим задачам, таким как изоморфизм и симметрия;
- [Knu97a] и [Knu98] — общепризнанные справочники по сортировке и поиску; содержат обширный материал по комбинаторным объектам, таким как перестановки. Отдельными изданиями выпущен материал (предположительно составляющий содержимое мифического Тома 4) по генерированию перестановок (см. [Knu05a]), подмножеств и разбиений (см. [Knu05b]), а также деревьев (см. [Knu06]);
- [SW86a] — учебник по комбинаторике для студентов вузов, содержащий алгоритмы генерирования перестановок, подмножеств и разбиений множества. Содержит соответствующие программы на языке Pascal;
- [PS03] — описание библиотеки Combinatorica, содержащей свыше 400 функций системы Mathematica для генерирования комбинаторных объектов и объектов тео-

рии графов (см. *раздел 19.1.9*). Авторы книги придерживаются особого взгляда на взаимодействие разных алгоритмов.

14.1. Сортировка

Вход. Множество из n элементов.

Задача. Расположить элементы в возрастающем (или убывающем) порядке (рис. 14.1).

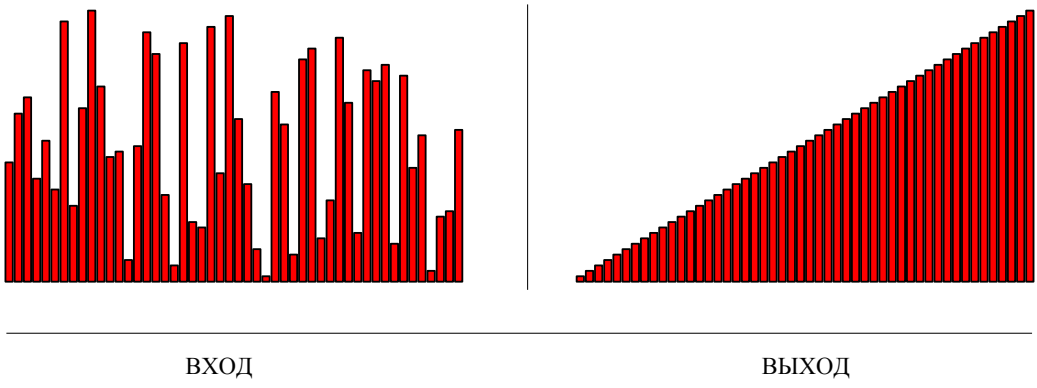


Рис. 14.1. Сортировка

Обсуждение. Сортировка является фундаментальной алгоритмической задачей теории вычислительных систем. Для программиста изучение разных алгоритмов сортировки подобно изучению гамм для музыканта. Как показано в *разделе 4.2*, сортировка является первым шагом в решении множества других алгоритмических задач, а правило "если не знаешь, как поступить, выполняй сортировку" является одним из главных при разработке алгоритмов.

Сортировка также иллюстрирует все стандартные парадигмы разработки алгоритмов. Большинству программистов известно так много разных алгоритмов сортировки, что зачастую им трудно принять решение, какой из них выбрать в конкретном случае. Следующие вопросы помогут вам в выборе алгоритма.

- *Сколько элементов нужно отсортировать?* Для небольших объемов данных ($n \leq 100$) не играет особой роли, какой из алгоритмов с квадратичным временем исполнения использовать. Сортировка вставками быстрее, проще и в меньшей степени чревата ошибками реализации, чем пузырьковая сортировка. Сортировка методом Шелла очень похожа на сортировку вставками, только намного быстрее. Однако для ее реализации необходимо знание правильной последовательности вставок, изложенной в книге [Knu98].

  Если количество сортируемых элементов больше 100, то важно использовать алгоритм с временем исполнения $O(n\lg n)$, такой как пирамидальная сортировка, быстрая сортировка или сортировка слиянием. Разные программисты выбирают разные алгоритмы, руководствуясь личными предпочтениями. Поскольку трудно сказать, какой алгоритм быстрее, выбор того или иного алгоритма не имеет особого значения.

А когда количество элементов для сортировки превышает, скажем, 5 000 000, то пора начинать думать об использовании алгоритма сортировки данных, содержащихся на внешних запоминающих устройствах, который минимизирует обращения к этим устройствам. Алгоритмы обоих типов рассматриваются далее в этом разделе.

- *Содержатся ли в данных дубликаты?* Порядок сортировки полностью определен, если все элементы имеют разные ключи. Но когда два элемента имеют одинаковый ключ, то для решения, какой элемент должен стоять первым, нужно применить дополнительный критерий. Для многих приложений это не играет роли, и подходит любой алгоритм сортировки. А в случаях, когда это важно, очередность элементов с одинаковыми ключами определяется по вторичному ключу, например, по имени для одинаковых фамилий.

  А иногда очередность в отсортированном списке определяется по положению в исходном списке. Допустим, что 5-й и 27-й элементы исходного набора данных имеют одинаковый ключ. В таком случае в отсортированном списке 5-й элемент должен находиться перед 27-м. В случае совпадения ключей *стабильный* алгоритм сортировки сохраняет исходный порядок элементов в отсортированном списке. Большинство алгоритмов сортировки с квадратичным временем исполнения являются стойкими, в то время как многие из алгоритмов с временем исполнения $O(n\lg n)$ — нет. Если важна стабильность сортировки, то в функции сравнения, скорее всего, будет лучше использовать исходное положение элемента в качестве вторичного ключа, чем полагаться на реализацию алгоритма для обеспечения стабильности.

- *Что известно о данных?* Для ускорения сортировки данных часто можно воспользоваться их особенностями. Конечно, сортировка, имеющая время исполнения $O(n\lg n)$, является быстрой операцией, поэтому, если узким местом вашего приложения является время, затрачиваемое на сортировку, можете считать, что вам повезло.

  - *Данные уже частично отсортированы?* В таком случае определенные алгоритмы, например алгоритм сортировки вставками, работают быстрее, чем на полностью неотсортированных данных.

  - *Что известно о распределении ключей?* Если ключи распределены произвольно или равномерно, то имеет смысл использовать *корзинную сортировку* или *сортировку распределением*. Просто помещаем ключи в корзины по их первой букве и выполняем рекурсию до тех пор, пока каждая корзина не достигнет размера, достаточно малого для сортировки полным перебором. Это очень эффективный подход, когда все ключи распределены по корзинам равномерно. Но в случае сортировки списка группы однофамильцев производительность корзинной сортировки будет очень плохой.

  - *Очень длинные или трудные для сравнения ключи?* Если ключи представлены в виде длинных текстовых строк, то, возможно, имеет смысл выполнить предварительную сортировку по сравнительно короткому префиксу (скажем, длиной в 10 символов), после чего выполнить сортировку по полному ключу. Этот метод особенно подходит для сортировки на внешних носителях (см. далее), при которой нежелательно расходовать быструю память на хранение несущественных деталей.

Другим подходом может быть использование поразрядной сортировки, имеющей время исполнения линейное относительно количества символов в файле, что гораздо лучше времени $O(n\lg n)$, которое еще должно быть помножено на стоимость сравнения двух ключей.

- *Невелик диапазон возможных ключей?* Если требуется отсортировать подмножество из, скажем, $n/2$ разных целых чисел, значение каждого из которых лежит в диапазоне от 1 до n, то самым быстрым алгоритмом будет создать n-элементный битовый вектор, установить соответствующие ключам биты, а потом отсканировать вектор слева направо и доложить о позициях с установленными битами.

♦ *Необходимо ли обращаться к диску?* При сортировке больших объемов информации сортируемые данные могут не помещаться в память. В таких случаях мы имеем дело с задачей *внешней сортировки* (external sorting), для которой требуется применение внешнего устройства хранения данных. Первоначально в качестве внешних устройств хранения данных использовались накопители на магнитной ленте, и в книге [Knu98] описываются разнообразные сложные алгоритмы для эффективного слияния данных с разных накопителей. В настоящее время используется виртуальная память и обмен данными с диском. Любой алгоритм сортировки будет работать с виртуальной памятью, но у многих алгоритмов все время будет уходить на перемещение данных между памятью и диском.

Самым простым подходом к внешней сортировке будет загрузка данных в B-дерево (см. *раздел 12.1*) с последующим симметричным обходом этого дерева, позволяющим читать ключи в отсортированном порядке. Однако, действительно высокопроизводительные алгоритмы основаны на сортировке многоканальным слиянием. При этом подходе данные разбиваются на несколько файлов, каждый из которых сортируется с применением быстрой внутренней сортировки, после чего отсортированные файлы сливаются поэтапно с использованием двух- или k-канального слияния. Производительность можно оптимизировать, используя сложные схемы слияния и управления буфером, учитывающие свойства внешнего устройства хранения.

♦ *Сколько времени отводится на разработку и отладку процедуры сортировки?* Если бы мне нужно было реализовать рабочую процедуру в течение часа, то я, скорее всего, предпочел бы простую сортировку методом выбора. Но если бы в моем распоряжении был целый день, то я, скорее всего, использовал бы пирамидальную сортировку, т. к. этот алгоритм обладает надежной производительностью и не требует дополнительной настройки.

Самым лучшим алгоритмом внутренней сортировки является быстрая сортировка (см. *раздел 4.2*), хотя для получения максимальной производительности придется повозиться с настройкой. В действительности, будет намного лучше использовать для этого алгоритма библиотечную функцию вашего компилятора, чем пытаться разработать свою реализацию. Неудачная реализация быстрой сортировки будет работать медленнее, чем плохая реализация пирамидальной сортировки.

Если же вы настроены на создание своей реализации алгоритма быстрой сортировки, то используйте следующие эвристические методы, которые на практике оказывают большое влияние на производительность.

- *Используйте рандомизацию.* Выполнив произвольную перестановку ключей перед сортировкой (см. *раздел 14.4*), вы, возможно, добьетесь того, что время сортировки почти отсортированных данных не будет квадратичным.
- *Выбирайте средний элемент из трех.* Для выбора опорного элемента используйте средний по величине элемент из первого, последнего и центрального элемента массива, чтобы повысить вероятность разбиения массива на приблизительно одинаковые части. По результатам некоторых экспериментов для больших подмассивов следует составлять выборку из большего количества элементов, а для небольших — из меньшего.
- *Для сортировки вставками выбирайте подмассивы небольшого размера.* Переключение с рекурсивной быстрой сортировки на сортировку вставками имеет смысл только при небольшом размере подмассивов, например, не превышающем 20 элементов. Оптимальный размер подмассивов для данной реализации определяется экспериментальным путем.
- *Обрабатывайте наименьшую* порцию данных *в первую очередь.* Исходя из предположения, что ваш компилятор способен оптимизировать хвостовую рекурсию, вы можете минимизировать объем требуемой памяти, обрабатывая меньшие порции данных раньше. Так как каждый следующий рекурсивный вызов занимает в стеке память, превышающую память, отведенную под предыдущий вызов, максимум в полтора раза, то потребуется только $O(\lg n)$ стековой памяти.

Прежде чем приступать к реализации алгоритма быстрой сортировки, прочитайте статью [Ben92b].

Реализации. Наилучшей программой сортировки с открытым кодом предположительно является GNU sort, которая является частью библиотеки основных утилит GNU. Дополнительную информацию см. по адресу **http://www.gnu.org/software/coreutils/**. Имейте в виду, что существуют также коммерческие поставщики высокопроизводительных программ внешней сортировки, такие как Cosort (**www.cosort.com**), Syncsort (**www.syncsort.com**) и Ordinal Technology (**www.ordinal.com**).

Современные языки программирования содержат библиотеки эффективных реализаций сортировки, поэтому вам никогда не придется разрабатывать собственную процедуру. Стандартная библиотека C содержит функцию `qsort`, являющуюся, предположительно, обобщенной реализацией быстрой сортировки. Библиотека STL языка C++ предоставляет методы `sort` и `stable_sort`. Библиотеку STL можно загрузить с веб-сайта **http://www.sgi.com/tech/stl/**. Более подробное руководство по использованию библиотеки STL и стандартной библиотеки C++ можно найти в книгах [Jos99], [Mey01] и [MDS01].

Стандартный пакет утилит Java java.util (**http://java.sun.com/javase/**) содержит небольшую библиотеку структур данных Java Collections (JC), предоставляющую, в частности, классы `SortedMap` и `SortedSet`.

Реализацию нечувствительного к кэшированию алгоритма funnelsort на языке C++ см. на веб-сайте **http://kristoffer.vinther.name/projects/funnelsort/**. Результаты тестов свидетельствуют о его высокой производительности.

Существует множество веб-сайтов, содержащих анимационные апплеты для всех фундаментальных алгоритмов сортировки, включая пузырьковую сортировку, пирамидальную сортировку, быструю сортировку, поразрядную сортировку и сортировку методом Шелла. Безусловно, сортировка является классической задачей для анимации алгоритмов. Типичные примеры таких сайтов: **http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html** и **http://www.cs.bell-labs.com/cm/cs/pearls/sortanim.html**.

Примечания

Самой лучшей книгой по сортировке из числа написанных в прошлом и, пожалуй, в будущем является книга [Knu98]. Ей больше тридцати лет, но читать ее по-прежнему интересно. Одной из областей сортировки, которые были разработаны после первого издания этой книги, является сортировка предварительно отсортированных данных, исследуемая в работе [ECW92]. Заслуживающим внимания справочником по сортировке является книга [GBY91], которая содержит ссылки на алгоритмы для частично отсортированных данных, а также реализации на языках C и Pascal всех фундаментальных алгоритмов.

Описания основных алгоритмов внутренней сортировки содержатся в каждом учебнике по алгоритмам. Алгоритм пирамидальной сортировки был разработан в 1964 г. Дж. Вильямсом (J. Williams), см. [Wil64]. Алгоритм быстрой сортировки был разработан в 1960 г. Чарльзом Хоаром (Charles Hoare), см. [Hoa62]; а всесторонний анализ и реализация были осуществлены Р. Седжвиком (R. Sedgewick), см. [Sed78]. Первая реализация алгоритма сортировки слиянием (на компьютере EDVAC в 1945 г.) была выполнена фон Нейманом. Полное изложение истории сортировки, берущей начало во времена электронно-вычислительных машин, работавших на перфокартах, см. в книге [Knu98].

Главным чемпионатом по высокопроизводительной сортировке являются ежегодные соревнования, начало которым положил Джим Грей (Jim Gray). На веб-сайте **http://sortbenchmark.org/** опубликована информация о текущих и предыдущих результатах, которые могут вдохновлять или расстраивать, в зависимости от вашей точки зрения. Достигнутый прогресс вдохновляет (тестовые экземпляры размером в миллион записей, которые использовались в первых соревнованиях, теперь кажутся незначительными), но лично меня расстраивает тот факт, что вопросы управления системой и памятью оказываются гораздо важнее комбинаторно-алгоритмических аспектов сортировки.

В настоящее время создание высокопроизводительных программ сортировки включает в себя работу над алгоритмами как требовательными к кэшированию (см. [LL99]), так и нечувствительными к нему (см. [BFV07]).

Для сортировки существует хорошо известная нижняя граница $\Omega(n\lg n)$ на модели алгебраического дерева решений (см. [BO83]). Задача определения точного количества сравнений, необходимого для сортировки n элементов при небольших значениях n, вызвала значительный интерес у исследователей. Описание задачи можно найти в [Aig88] и [Raw92], а последние результаты — в [Pec04] и [Pec07].

Но эта нижняя граница не соблюдается при других моделях вычислений. В работе [FW93] представляется алгоритм сортировки со сложностью $O(n\sqrt{\lg n})$ по модели вычислений, позволяющей выполнять арифметические операции над ключами. Обзор алгоритмов быстрой сортировки по таким нестандартным моделям вычислений приведен в [And05].

Родственные задачи. Словари (см. *раздел 12.1*), поиск (см. *раздел 14.2*), топологическая сортировка (см. *раздел 15.2*).

14.2. Поиск

Вход. Набор S из n ключей и ключ запроса q.

Задача. Определить местоположение ключа q в наборе ключей S (рис. 14.2).

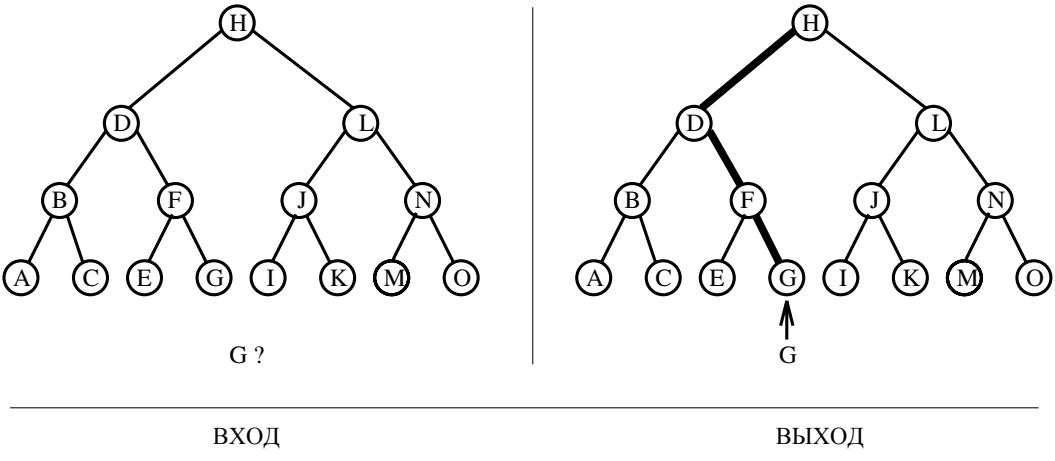


Рис. 14.2. Поиск в наборе ключей

Обсуждение. Для разных людей слово "поиск" означает разные понятия. Поиск глобального максимума или минимума функции является задачей *безусловной оптимизации*, которая рассматривается в *разделе 13.5*. Программы для игры в шахматы выбирают свой лучший ход, выполняя исчерпывающий перебор/поиск всех возможных ходов, используя вариант поиска с возвратом (см. *раздел 7.1*).

Мы же ставим перед собой задачу найти ключ в списке, массиве или дереве. Словарные структуры данных предоставляют эффективный доступ к наборам ключей для вставки и удаления (см. *раздел 12.1*). Типичными представителями словарных структур данных являются двоичные деревья и хэш-таблицы.

Мы рассматриваем задачу поиска без привязки к словарям, поскольку в нашем распоряжении имеются более простые и эффективные решения, когда мы занимаемся статическим поиском без вставок и удалений. При правильном использовании во внутреннем цикле такие простые структуры данных могут обеспечить заметное повышение производительности. Кроме того, идеи двоичного поиска и самоорганизации применимы при решении других задач, что оправдывает наше внимание к ним.

Существуют два основных метода поиска: последовательный поиск и двоичный поиск. Оба достаточно просты, но у них имеются интересные варианты. При *последовательном поиске* мы движемся по списку ключей с самого начала и сравниваем каждый последующий элемент с ключом поиска, пока не найдем совпадающий или не достигнем конца списка. При *двоичном поиске* мы работаем с отсортированным списком или массивом ключей. Чтобы найти ключ q, мы сравниваем значение q со значением среднего ключа массива $S_{n/2}$. Если значение ключа q меньше, чем значение ключа $S_{n/2}$, то данный ключ должен находиться в верхней половине массива S; в противном случае он должен находиться в его нижней половине. Повторяя этот процесс на той части массива, кото-

рая содержит элемент q, мы находим его за $[\lg n]$ сравнений, что намного лучше, чем ожидаемые $n/2$ сравнений при последовательном поиске. Дополнительную информацию по двоичному поиску см. в *разделе 4.9*.

Последовательный поиск является простейшим алгоритмом и, скорее всего, будет самым быстрым, когда количество элементов не превышает 20. При ста и более элементах двоичный поиск будет эффективнее, чем последовательный, и оправдает затраты на сортировку при большом количестве запросов. Но для принятия решения о выборе варианта алгоритма следует учитывать и другие факторы.

- *Сколько времени отводится на программирование?* Известно, что алгоритм двоичного поиска сложен для программирования. Между изобретением алгоритма и опубликованием первой *правильной* реализации прошло целых семнадцать лет! Поэтому я советую вам начать с одной из реализаций, описанных далее. Полностью протестируйте ее, создав программу, которая выполняет поиск каждого ключа в наборе S, а также для промежуточных значений ключей.
- *Какова частота обращений к разным ключам?* Определенные английские слова (например, "the") встречаются намного чаще, чем другие (например, "defenestrate"). Количество сравнений в последовательном поиске можно уменьшить, поместив часто употребляемые слова вверху списка, а более редкие — внизу. Неравномерность запросов является скорее правилом, чем исключением. Слова во многих языках распределяются согласно *степенным законам*. Классическим примеров является распределение слов в английском языке, которое довольно точно моделируется *законом Ципфа*. Согласно этому закону, i-й по частоте обращения ключ выбирается с вероятностью, которая в $(i-1)/i$ раз превышает вероятность выбора ключа, $(i-1)$-го по популярности, для всех $1 \le i \le n$.

  Знание частоты обращений легко использовать в последовательном поиске. Но в двоичном дереве дело обстоит сложнее. Мы хотим, чтобы часто употребляемые ключи располагались возле корня (таким образом мы быстрее найдем их), но не за счет разбалансирования дерева и превращения двоичного поиска в последовательный. Решением этой проблемы будет применение алгоритма динамического программирования для построения оптимального дерева двоичного поиска. Принципиально важным здесь является то обстоятельство, что каждый возможный корневой узел i делит пространство ключей на две части (слева и справа от i), каждую из которых можно представить оптимальным деревом двоичного поиска для меньшего поддиапазона ключей. Корень оптимального дерева выбирается таким образом, чтобы минимизировать ожидаемую стоимость поиска в получившемся разбиении.
- *Может ли частота обращений измениться со временем?* Для предварительной сортировки списка или дерева с целью воспользоваться асимметричной закономерностью доступа требуется знать эту закономерность доступа наперед. Для многих приложений получение такой информации может быть трудной задачей. Лучшим подходом будет использовать *самоорганизующиеся списки*, в которых порядок ключей изменяется в ответ на запросы. Самой лучшей схемой самоорганизации является перемещение ключа, к которому выполнялось самое последнее обращение, в начало списка. Таким образом ключи с наибольшим числом обращений продвигаются в начало списка, а с наименьшим — к концу. Отслеживать частоту обращений нет

необходимости, мы просто перемещаем ключ при обращении к нему. В самоорганизующихся списках также используется принцип временной локальности, т. к. существует вероятность повторных обращений к данному ключу. Таким образом, пользующийся спросом ключ будет удерживаться вверху списка на протяжении последовательности обращений, даже если в прошлом другие ключи пользовались бóльшим спросом.

Принцип самоорганизации может расширить размер полезного диапазона последовательного поиска, но когда количество ключей превышает 100, то следует переключаться на двоичный поиск. Но также можно рассмотреть возможность использования косых деревьев, являющихся деревьями двоичного поиска, в которых элемент, к которому было выполнено обращение, перемещается в корневой узел. Эти деревья дают гарантию отличной амортизированной производительности.

- *Известно ли примерное местоположение ключа?* Допустим, мы знаем, что требуемый ключ находится справа от позиции p, причем на небольшом расстоянии. Если мы правы в нашем предположении, то последовательный поиск быстро возвратит нужный элемент, но за ошибку нам придется дорого заплатить. Гораздо лучше проверять ключи через возрастающие интервалы справа от p ($p + 1$, $p + 2$, $p + 4$, $p + 8$, $p + 16$, ...), пока мы не дойдем до ключа, расположенного справа от целевого. Тогда у нас появится окно, содержащее целевой элемент, и мы сможем найти его, применив двоичный поиск.

  Такой односторонний двоичный поиск возвращает целевой ключ в позиции $p + l$ за, самое большее, $2\lceil \lg l \rceil$ сравнений, т. е. он работает быстрее двоичного поиска, если $l \ll n$, и такая производительность никогда не станет намного хуже. Односторонний двоичный поиск особенно полезен при неограниченном поиске, например, при извлечении корня.

- *Находится ли структура данных на внешнем устройстве?* Когда количество ключей слишком велико, двоичный поиск теряет свой статус лучшего метода поиска. Мы будем метаться по пространству ключей в поисках средней точки для сравнения с целевым ключом и для каждого из этих сравнений потребуется загрузить новую страницу из внешнего устройства. Намного лучше будет использовать такие структуры данных, как B-деревья (см. *раздел 12.1*) или деревья Емде Боаса (см. примечания далее), которые собирают ключи в страницы, чтобы свести к минимуму количество обращений к диску в каждом поиске.

- *Можно ли угадать местоположение ключа?* В *интерполяционном поиске* мы используем свое знание распределения ключей, чтобы угадать следующую позицию для сравнения. Кстати, при поиске в телефонной книге более уместно было бы говорить об интерполяционном, а не о двоичном поиске. Допустим, что мы ищем номер телефона человека по фамилии Washington. Мы можем уверенно делать первое сравнение где-то в третьей четверти книги, фактически выполняя два сравнения по цене одного.

  Хотя идея интерполяционного поиска кажется привлекательной, мы рекомендуем не использовать его по трем причинам. Во-первых, нужно будет выполнить большую работу, чтобы оптимизировать алгоритм поиска, прежде чем можно будет на-

деяться получить лучшую производительность, чем при двоичном поиске. Во-вторых, даже если вы и повысите производительность по сравнению с двоичным поиском, то вряд ли настолько, чтобы оправдать вложенные усилия. И наконец, в-третьих, ваша программа будет намного менее надежна и эффективна при смене локали, например, при поиске французских слов вместо английских.

Реализации. Элементарные алгоритмы последовательного и двоичного поиска достаточно просты, что можно рассматривать возможность их самостоятельной реализации. Тем не менее, стандартная библиотека C содержит процедуру `bsearch`, являющуюся, предположительно, обобщенной реализацией двоичного поиска. Библиотека STL языка C++ предоставляет итераторы `find` (последовательный поиск) и `binary_search` (двоичный поиск). Стандартный пакет утилит Java java.util (**http://java.sun.com/javase**) предоставляет процедуру `binarySearch`.

Многочисленные реализации приводятся во многих учебниках по структурам данных. Реализации косых деревьев и других поисковых структур на языке C++ и Java даются в книгах [Sed98] (**http://www.cs.princeton.edu/~rs**) и [Wei06] (**http://www.cs.fiu.edu/~weiss**).

Примечания

В книге ([MS05]) приводится обзор современного состояния дел в области словарных структур данных. Другие обзоры представлены в книгах [MT90b] и [GBY91]. В книге [Knu97a] представлен детальный анализ и описание основных алгоритмов поиска и словарных структур данных, однако в ней отсутствуют такие современные структуры данных, как красно-черные и косые деревья.

Очередная позиция сравнения при линейном интерполяционном поиске в массиве отсортированных чисел определяется по следующей формуле:

$$next = (low - 1) + \lceil \frac{q - S[low - 1]}{S[high + 1] - S[low - 1]} \times (high - low + 1) \rceil$$

где q — числовой ключ запроса, а S — отсортированный массив числовых значений. Если ключи распределены равномерно и выбираются независимым образом, то ожидаемое время поиска равно $O(\lg\lg n)$ (см. [DJP04] и [PIA78]).

Неравномерность распределения обращений можно использовать в деревьях двоичного поиска, организовав их таким образом, чтобы часто запрашиваемые ключи находились возле корня, что позволит минимизировать время поиска. Такие оптимальные деревья поиска можно создавать за время $O(n\lg n)$ посредством динамического программирования (см. [Knu98]). В книге [SW86b] описан нетривиальный алгоритм для эффективного преобразования двоичного дерева в дерево минимальной высоты (оптимально сбалансированное) посредством операций ротации.

Метод Емде Боаса для создания двоичного дерева (или отсортированного массива) обеспечивает более высокую производительность при работе с внешними устройствами хранения данных, чем двоичный поиск, но за счет более сложной реализации. Информацию об этой и других нечувствительных к кэшированию структурах см. в [ABF05].

Родственные задачи. Словари (см. *раздел 12.1*), сортировка (см. *раздел 14.1*).

14.3. Поиск медианы и выбор элементов

Вход. Набор из n чисел (или ключей), целое число k.

Задача. Найти ключ такой, что ровно k ключей больше него (рис. 14.3).

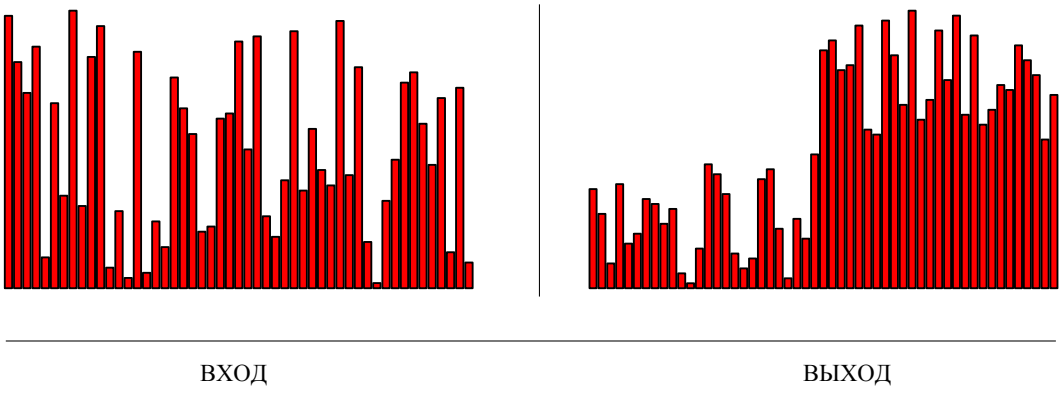


Рис. 14.3. Сортировка по среднему значению

Обсуждение. Поиск медианы (элемента, среднего по местоположению) является важной задачей в статистике, поскольку он возвращает более устойчивое понятие, чем *среднее значение*. На среднее значение доходов всех людей, опубликовавших научные работы по сортировке, оказывает сильное влияние факт присутствия среди них Билла Гейтса (см. [GP79]), но его влияние на *медиану* доходов сводится к нейтрализации одного малообеспеченного аспиранта.

Задача поиска медианы является частным случаем общей *задачи о выборе*, в которой требуется выбрать i-й элемент в отсортированном списке. Задача выбора возникает во многих приложениях, примеры которых приводятся в следующем списке:

- *Фильтрация элементов с резко отклоняющимися значениями.* При работе с зашумленными данными будет полезно отбросить, скажем, 10% самых больших и самых меньших значений. Для этого выбираются элементы, соответствующие 10-му и 90-му процентилям, после чего значения, не попадающие в диапазон, ограниченный этими точками, отбрасываются.
- *Идентификация наиболее перспективных кандидатов.* В компьютерной программе для игры в шахматы мы можем быстро оценить все возможные следующие ходы и затем рассмотреть 25% самых лучших ходов более пристально. В результате фильтрации мы получим оптимальный вариант.
- *Применение децилей и аналогичные способы деления диапазона.* Удобным способом представления распределения доходов населения является отображение зарплат людей на графике по равномерным интервалам, границы которых устанавливаются, например, по 10-му перцентилю, по 20-му перцентилю и т. д. Вычисление этих значений просто сводится к выбору элементов из соответствующего интервала.

- *Статистика порядков.* Особенно интересные частные случаи задачи выбора — это поиск наименьшего элемента ($k = 1$), наибольшего элемента ($k = n$) и медианного элемента ($k = n/2$).

Среднее значение n чисел можно вычислить за линейное время, сложив все элементы и разделив получившуюся сумму на n. Но поиск медианы представляет более трудную задачу. Алгоритмы вычисления медианы можно легко обобщить к произвольной выборке. Перечислю вопросы, на которые следует ответить при поиске медианы и выборе элементов.

- *Какой должна быть скорость работы?* Самый простой алгоритм вычисления медианы сортирует элементы за время $O(n\lg n)$, после чего возвращает элемент, занимающий $(n/2)$-ю позицию. Бесспорное достоинство этого алгоритма состоит в том, что он выдает намного больше информации, чем одна лишь медиана, позволяя выбрать k-й элемент (для любого $1 \le k \le n$) за постоянное время после сортировки. Но если вам требуется только медиана, то для ее поиска существуют более быстрые алгоритмы.

  В частности, одним из таких алгоритмов является алгоритм на основе быстрой сортировки с *ожидаемым временем* исполнения $O(n)$. Этот алгоритм работает следующим образом. Выбираем в наборе данных произвольный элемент и используем его для разбиения данных на два набора, один из которых содержит элементы меньшие, чем элемент-разделитель, а другой — большие. Зная размер этих наборов, мы знаем позицию элемента-разделителя в исходном наборе, а следовательно, и расположение медианы — слева или справа от разделителя. Рекурсивно выполняем эту процедуру на соответствующем наборе данных, пока не найдем медиану. Нам понадобится, в среднем, $O(\lg n)$ итераций, а затраты на выполнение каждой последующей итерации примерно равны половине стоимости предыдущей. Времена выполнения каждой итерации образуют геометрическую прогрессию, которая сходится к линейному времени исполнения. Впрочем, если нам очень не повезет, то время исполнения может оказаться таким же, как и для быстрой сортировки, т. е. $O(n^2)$.

  Более сложные алгоритмы находят медиану за линейное время в наихудшем случае. Однако для практического применения лучше всего подойдет алгоритм с ожидаемым линейным временем исполнения. Только обязательно выбирайте случайные элементы-разделители, чтобы избежать наихудшего случая.

- *Как поступить, если мы видим каждый элемент только один раз?* На больших наборах данных операции выбора элементов и вычисления медианы становятся слишком дорогими, т. к. для них обычно требуется выполнить несколько проходов по данным, хранящимся на внешних устройствах. А у приложений, работающих с потоковыми данными, объем данных слишком велик для того, чтобы их сохранять. В результате их повторное рассмотрение (и, следовательно, вычисление точного значения медианы) невозможно. В таких случаях лучше построить менее объемную модель данных для дальнейшего анализа, например, на основе децилей или моментов распределения (где k-й момент потока x определяется как $F_k = \sum_i x_i^k$).

  Одним из решений такой задачи будет произвольная выборка. Решение, сохранять ли значение, принимается на основе броска монеты, у которой вероятность выпаде-

ния стороны, соответствующей сохранению, достаточно низка, чтобы не переполнить буфер. Скорее всего, медиана сохраненных выборок будет близкой к медиане исходного набора данных. В качестве альтернативного варианта можно выделить определенную область памяти для хранения, например, значений децилей больших блоков, после чего объединить децили, чтобы получить более точные границы.

- *Насколько быстро можно найти моду?* Кроме среднего и медианы существует еще одно понятие средней величины — *мода*. Мода определяется как наиболее часто встречающийся элемент набора данных. Наилучший алгоритм для вычисления моды сортирует набор данных за время $O(n\lg n)$, в результате чего одинаковые элементы оказываются рядом. Перебирая отсортированные данные слева направо, мы можем найти длину самой протяженной последовательности одинаковых элементов и вычислить моду за общее время $O(n\lg n)$.

  Необходимо заметить, что более быстрого алгоритма для вычисления моды не существует, т. к. можно доказать, что задача проверки наличия двух идентичных элементов в наборе имеет нижнюю временную границу $\Omega(n\lg n)$. Задача поиска одинаковых элементов эквивалентна задаче поиска нескольких мод. Существует возможность, по крайней мере, теоретическая, улучшить время исполнения для больших значений моды за счет использования быстрых вычислений медианы.

Реализации. Библиотека STL на языке C++ содержит универсальный метод выбора элемента (`nth_element`), реализованный на основе алгоритма с ожидаемым линейным временем исполнения. Библиотеку STL можно загрузить вместе с документацией с вебсайта **http://www.sgi.com/tech/stl/**. Библиотека STL и стандартная библиотека C++ подробно описаны в книгах [Jos99], [Mey01] и [MDS01].

Примечания

Алгоритм с ожидаемым линейным временем исполнения для вычисления медианы и выбора элемента был разработан Хоаром (Hoare); см. [Hoa61]. В работе [FR75] представлен алгоритм, выполняющий в среднем меньшее число сравнений. Хорошие описания алгоритма выбора с линейным временем исполнения приведены в книгах [AHU74], [BvG99], [CLRS01] и [Raw92], среди которых [Raw92] является наиболее информативной.

Потоковые алгоритмы широко применяются для работы с большими наборами данных; подробный обзор этих алгоритмов можно найти в книге [Mut05].

Значительный теоретический интерес представляет задача определения *точного* количества сравнений, достаточных для вычисления медианы n элементов. Алгоритм с линейным временем исполнения, описываемый в работе [BFP+72], доказывает, что достаточно cn сравнений, но мы хотим знать, чему равно c. В работе [DZ99] доказано, что для вычисления медианы достаточно выполнить $2{,}95n$ сравнений. Эти алгоритмы стремятся минимизировать количество сравнений элементов, но не общее количество операций, и поэтому на практике не оказываются намного быстрее существующих. Кроме того, они оставляют на месте наилучшую на данный момент нижнюю границу времени вычисления медианы, равную $(2 + \varepsilon)$ сравнениям.

Исследование пределов комбинаторных алгоритмов для решения задач выбора представлено в книге [Aig88]. Оптимальный алгоритм для вычисления моды приведен в работе [DM80].

Родственные задачи. Очереди с приоритетами (см. *раздел 12.2*), сортировка (см. *раздел 14.1*).

14.4. Генерирование перестановок

Вход. Целое число n.

Задача. Создать: все возможные перестановки, или случайную перестановку, или очередную перестановку размером n (рис. 14.4).

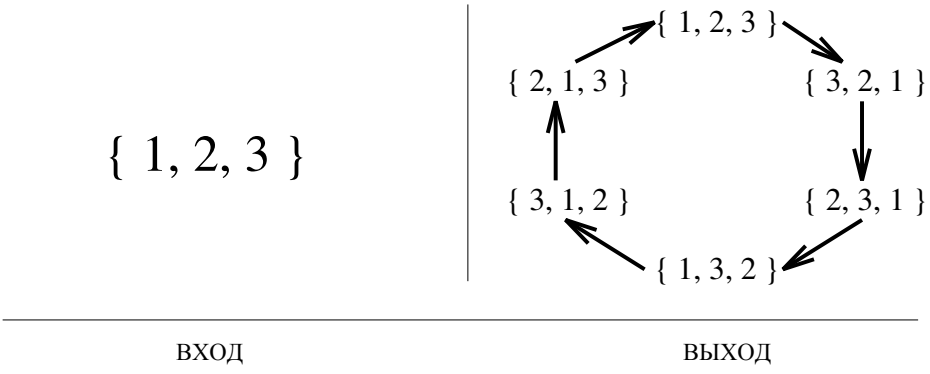


Рис. 14.4. Перестановки

Обсуждение. Перестановкой называется упорядоченный набор элементов. Во многих алгоритмических задачах из этого каталога требуется найти наилучший способ упорядочения набора объектов. В качестве примеров можно назвать задачу *коммивояжера* (определение порядка посещения n городов, имеющего наименьшую стоимость), задачу *уменьшения ширины ленты* (упорядочивание вершин графа в линию таким образом, чтобы минимизировать длину самого длинного ребра) и задачу *изоморфизма графа* (упорядочивание вершины графа таким образом, чтобы он был идентичным другому графу). Любой алгоритм для предоставления точного решения таких задач должен создавать в процессе решения последовательность перестановок.

Из n элементов можно создать $n!$ перестановок. С увеличением n количество перестановок возрастает так быстро, что не стоит надеяться сгенерировать все перестановки для $n \geq 12$, т. к. 12! = 479 001 600. Подобные числа должны охладить пыл любого человека, пытающегося решить задачу методом исчерпывающего перебора, и помочь объяснить важность генерирования случайных перестановок.

При обсуждении генерирования перестановок основным является понятие порядка, т. е. последовательности, в которой они создаются. Наиболее естественным порядком генерирования перестановок является *лексикографический*, т. е. такой, в котором они находились бы после сортировки. Например, лексикографический порядок перестановок для $n = 3$ будет {1, 2, 3}, {1, 3, 2}, {2, 1, 3}, {2, 3, 1}, {3, 1, 2}, {3, 2, 1}. Хотя лексикографический порядок эстетически более привлекателен, часто он не дает никаких определенных преимуществ. Например, при поиске в коллекции файлов порядок, в котором рассматриваются имена файлов, не имеет значения, при условии, что в конечном итоге вы просмотрите их все. Более того, если лексикографический порядок не требуется, то нередко получаются более быстрые и простые алгоритмы.

Для создания перестановок существуют две парадигмы: метод ранжирования и метод инкрементального изменения. Второй метод более эффективен, но первый применим

для решения более широкого класса задач. Ключевым моментом метода ранжирования является определение функций *Rank* и *Unrank* на всех перестановках p и целых числах n, m, где $|p| = n$ и $0 \le m \le n!$.

- *Функция Rank(p)* определяет позицию перестановки p в данном порядке генерирования. Типичная функция ранжирования является рекурсивной. Например, при базовом выражении $Rank(\{1\}) = 0$ общая формула выглядит так: $Rank(p) = (p_1 - 1)\cdot(|p| - 1)! + Rank(p_2, ..., p_{|p|})$.

  Чтобы функция была правильной, необходимо присвоить новые метки элементам меньшей перестановки, чтобы отображать удаленный первый элемент. Таким образом: $Rank(\{2, 1, 3\}) = 1\cdot2! + Rank(\{1, 2\}) = 2 + 0\cdot1! + Rank(\{1\}) = 2$.

- *Функция Unrank(m, n)* возвращает перестановку в позиции m из $n!$ перестановок из n элементов. Типичная функция рекурсивно определяет, сколько раз $(n - 1)!$ встречается в m. Выражение *Unrank(2,3)* говорит нам, что первый элемент перестановки должен быть "2", т. к. $(2 - 1)\cdot(3 - 1)! \le 2$, но $(3 - 1)\cdot(3 - 1)! > 2$. Удаление $(2 - 1)\cdot(3 - 1)!$ из m оставляет меньшую задачу *Unrank*(0,2). Ранг 0 соответствует полному упорядочению. Полным упорядочением на двух оставшихся элементах (т. к. 2 уже было использовано) является {1, 3}, поэтому *Unrank*(2, 3) = {2, 1, 3}.

Что собой представляют фактически функции *Rank* и *Unrank*, не имеет такого большого значения, как то обстоятельство, что они должны быть обратными друг другу. Иными словами, $p = Unrank(Rank(p), n)$ для всех перестановок p. Определив функции ранжирования для перестановок, мы можем решать многие родственные задачи. В частности:

- *генерирование следующей перестановки.* Чтобы определить следующую по порядку после p перестановку, мы можем выполнить *Rank*(p), добавить к результату 1, после чего выполнить *Unrank*(p). Аналогичным образом вычисляется и перестановка перед p: $Unrank(Rank(p) - 1, |p|)$. Выполнение функции *Unrank* над последовательностью целых чисел от 0 до $n! - 1$ будет равнозначно генерированию всех перестановок;

- *генерирование случайных перестановок.* Выбрав случайное целое число от 0 до $n! - 1$, а потом вызвав для него функцию *Unrank*, мы получим действительно случайную перестановку;

- *отслеживание набора перестановок.* Допустим, мы хотим генерировать случайные перестановки и предпринимать некоторое действие только в том случае, когда получаем новую перестановку. Для отслеживания уже созданных перестановок можно создать битовый вектор (см. *раздел 12.5*) из $n!$ битов и устанавливать в нем бит i при генерировании перестановки *Unrank*(i, n). Подобный метод использовался с подмножествами из k элементов в приложении для игры в лото, описанного в *разделе 1.6*.

Этот метод ранжирования лучше всего подходит для небольших значений n, т. к. вычисление $n!$ быстро приведет к переполнению, если только не используются арифметические операции с произвольной точностью (см. *раздел 13.9*). Итеративные методы включают в себя определение операции *next* и *previous* для преобразования одной перестановки в другую, обычно путем обмена местами двух элементов. Здесь сложной

частью является планирование таких обменов местами, чтобы перестановки не повторялись, пока не будут сгенерированы все перестановки. На рис. 14.4 показано упорядочение шести перестановок множества {1, 2, 3} с использованием одного обмена местами между соседними перестановками.

Алгоритмы, созданные на основе итеративных методов для генерирования последовательности перестановок, довольно сложны, но настолько лаконичны, что их можно реализовать программой, состоящей из десятка строк. Ссылки на существующий код см. в *подразделе "Реализации"*. Так как при инкрементальном изменении выполняется только один обмен, то эти алгоритмы могут быть чрезвычайно быстрыми — в среднем, с постоянным временем исполнения — что не зависит от размера перестановки! Секрет заключается в использовании *n*-элементного массива для представления перестановки, чтобы облегчить обмен местами. В некоторых приложениях важно только различие между перестановками. Например, в программе поиска оптимального маршрута коммивояжера методом исчерпывающего перебора стоимостью маршрута, связанного с новой перестановкой, будет стоимость предшествующей перестановки с добавлением или удалением четырех ребер.

До сих пор мы предполагали, что все элементы перестановок отличаются друг от друга. Однако если их множество содержит дубликаты (иными словами, является мультимножеством), то можно сэкономить значительное время и усилия, не рассматривая одинаковые перестановки. Например, для множества {1, 1, 2, 2, 2} имеется не 120 разных перестановок, а только десять. Чтобы избежать появления дубликатов, перестановки нужно генерировать в лексикографическом порядке методом поиска с возвратом.

Генерирование случайных перестановок представляет собой несложную, но важную задачу, с которой разработчикам приходится часто сталкиваться и с которой они не всегда справляются. Для правильного решения этой задачи нужно использовать следующий алгоритм, состоящий из двух строчек кода, с линейным временем исполнения. Предполагается, что функция *Random*[*i*, *n*] генерирует случайное целое число в диапазоне между *i* и *n* включительно.

for $i = 1$ to n do $a[i] = i$;

for $i = 1$ to $(n - 1)$ do *swap*[$a[i]$, $a[Random[i, n]]$];

Далеко не очевидно, что этот алгоритм генерирует все перестановки случайным образом с равномерным распределением. Если вы так не думаете, предоставьте убедительное объяснение, почему следующий алгоритм не генерирует равномерно распределенные перестановки:

for $i = 1$ to n do $a[i] = i$;

for $i = 1$ to $(n - 1)$ do *swap*[$a[i]$, $a[Random[1, n]]$];

Такие тонкости демонстрируют, почему нужно быть очень осторожным с алгоритмами генерации случайных чисел. Более того, мы рекомендуем подвергнуть достаточно длительному испытанию любой генератор случайных чисел, прежде чем по-настоящему доверять выдаваемым им результатам. Например, сгенерируйте четырехэлементные случайные перестановки 10 000 раз и убедитесь, что количество появлений всех

24 разных перестановок примерно одинаковое. Если же вы знаете, как измерять статистическую значимость, вы сможете оценить надежность вашего алгоритма еще точнее.

Реализации. Библиотека STL на языке C++ содержит две функции для генерирования перестановок в лексикографическом порядке — `next_permutation` и `prev_permutation`. В книге [KS99] описывается реализация алгоритмов для генерирования перестановок итеративным методом и в лексикографическом порядке. Коды на языке C можно загрузить с веб-сайта **http://www.math.mtu.edu/~kreher/cages/Src.html**.

Профессор Фрэнк Раски (Frank Ruskey) из университета г. Виктория в Канаде разработал сервер комбинаторных объектов (Combinatorial Object Server) для генерирования перестановок, подмножеств, разбиений, графов и других комбинаторных объектов. Сервер доступен по адресу **http://theory.cs.uvic.ca/**. Интерактивный интерфейс сервера позволяет указывать объекты, которые вы хотите получить. Для некоторых типов объектов имеются реализации на языках C, Pascal и Java.

Веб-сайт **http://www.jjj.de/fxt/** содержит процедуры на языке C++ для генерирования удивительно широкого диапазона комбинаторных объектов, включая перестановки и циклические перестановки.

Книга [NW78] уже многие годы является замечательным источником информации по генерированию комбинаторных объектов. Она содержит эффективные реализации алгоритмов (на языке FORTRAN) для генерирования случайных перестановок и перестановок методом минимального изменения порядка. Кроме того, в ней приводятся процедуры для выявления циклов в перестановке. Подробности см. в *разделе 19.1.10*.

Библиотека Combinatorica (см. [PS03]) предоставляет реализации алгоритмов (на языке программирования пакета Mathematica) для генерирования случайных перестановок и последовательностей перестановок с минимальными различиями и в лексикографическом порядке. Эта библиотека также содержит процедуру поиска с возвратом для создания всех несовпадающих перестановок мультимножества и поддерживает разнообразные групповые операции над перестановками. Подробности см. в *разделе 19.1.9*.

Примечания

Самым лучшим на сегодня справочным материалом по генерированию перестановок является одна из последних работ Дональда Кнута [Knu05a]. Обзор, представленный в [Sed77], написан раньше, но прогресс в этой области весьма незначителен. Хорошие описания можно найти в [KS99], [NW78] и [Rus03].

Методы быстрого генерирования перестановок выполняют только один обмен элементов местами для получения очередной перестановки. Алгоритм Джонсона-Троттера (см. [Joh63] и [Tro62]) удовлетворяет еще более строгому условию, а именно, что местами всегда обмениваются только смежные элементы. Простые функции с линейным временем исполнения для ранжирования перестановок описаны в работе [MR01].

В прежние времена, когда компьютеры не были так распространены, многие пользовались таблицами случайных перестановок, печатаемых в специальных книгах, например, [MO63]. Рассмотренный ранее алгоритм для генерирования случайных перестановок методом обмена местами двух элементов перестановки был впервые описан в этой же книге.

Родственные задачи. Генерирование случайных чисел (см. *раздел 13.7*), генерирование подмножеств (см. *раздел 14.5*), генерирование разбиений (см. *раздел 14.6*).

14.5. Генерирование подмножеств

Вход. Целое число n.

Задача. Сгенерировать: все возможные подмножества, или случайное подмножество, или очередное подмножество целых чисел $\{1, ..., n\}$ (рис. 14.5).

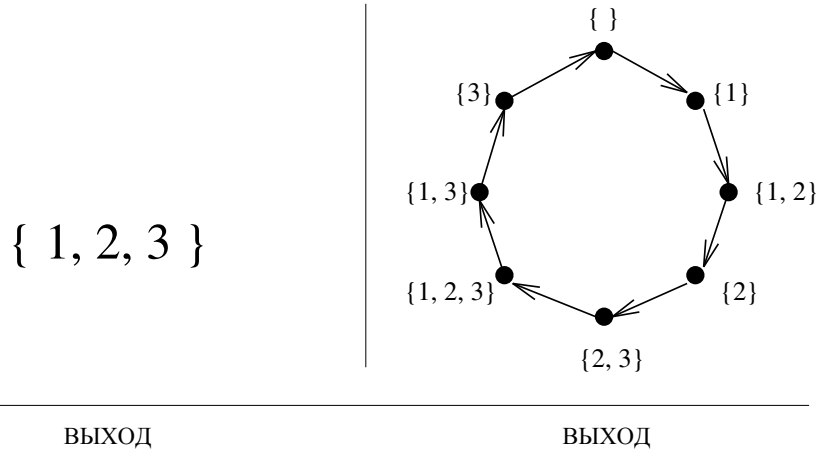


Рис. 14.5. Подмножества

Обсуждение. Подмножеством называется выборка объектов, порядок расположения которых не имеет значения. Целью многих алгоритмических задач является найти наилучшее подмножество набора объектов. Например, в *задаче о вершинном покрытии* требуется найти наименьшее подмножество вершин, покрывающее все ребра графа; в *задаче о рюкзаке* требуется найти наиболее дорогостоящее подмножество объектов в пределах заданного общего веса; а в *задаче упаковки множества* требуется найти наименьшее подмножество подмножеств, которые в совокупности содержат ровно по одному вхождению каждого элемента.

Множество из n элементов имеет 2^n разных подмножеств, включая пустое множество и само исходное множество. Количество подмножеств возрастает экспоненциально по отношению к значению n, но значительно медленнее, чем $n!$ перестановок из n элементов. В самом деле, т. к. $2^{20} = 1\,048\,576$, то задача просмотра всех подмножеств множества из 20 элементов легко выполнима. Так как $2^{30} = 1\,073\,741\,824$, то предел, несомненно, будет при немного больших значениях n.

По определению подмножества, относительный порядок его членов не играет роли при выяснении различий между подмножествами. Таким образом, подмножества {1, 2, 5} и {2, 1, 5} являются одинаковыми. Но всегда полезно поддерживать отсортированный или *канонический* порядок элементов подмножества, чтобы ускорить выполнение таких операций, как проверка двух подмножеств на идентичность.

Так же, как и в случае с перестановками (см. *раздел 14.1*), принципиальным моментом в задаче генерирования подмножеств является установление числовой последовательности среди всех 2^n подмножеств. Для этого существуют три основных подхода:

- *лексикографический порядок.* Лексикографический порядок означает, что элементы отсортированы, и часто является наиболее естественным способом генерирования

комбинаторных объектов. Восемь подмножеств множества {1, 2, 3} располагаются в лексикографическом порядке таким образом: {}, {1}, {1, 2}, {1, 2, 3}, {1, 3}, {2}, {2, 3} и {3}. Но генерирование подмножеств в лексикографическом порядке оказывается на удивление трудной задачей. Поэтому не стоит использовать этот подход, если только у вас нет какой-либо важной причины для этого;

- *код Грея.* Особенно интересной и полезной последовательностью подмножеств является порядок с минимальными различиями, в котором соседние подмножества отличаются присутствием (или отсутствием) ровно одного элемента. Такое упорядочение подмножеств, называющееся кодом Грея (Gray code), показано на рис. 14.5.

  Генерирование подмножеств в порядке кода Грея выполняется очень быстро, т. к. для этого существует удачная рекурсивная процедура. Создаем код Грея из $n-1$ элементов G_{n-1}. Создаем второй код Грея, являющийся обращенной копией первого кода, и добавляем n к каждому подмножеству этой копии. Потом выполним конкатенацию всех подмножеств, чтобы получить G_n. Чтобы лучше понять этот процесс, изучите пример, представленный на рис. 14.5.

  Кроме этого, так как разница между подмножествами состоит только в одном элементе, то алгоритмы поиска методов исчерпывающего перебора на основе кодов Грея могут быть довольно эффективными. Например, при решении задачи вершинного покрытия таким методом нужно будет обновлять изменение в покрытии, удалив или добавив только одно подмножество;

- *двоичный счет.* Самый простой подход к генерированию подмножеств основан на том обстоятельстве, что любое подмножество S' определяется элементами множества S, входящими в S'. Мы можем представить подмножество S' битовым вектором из n элементов, в котором бит i установлен в том случае, если i-й элемент множества S входит в подмножество S'. Таким образом определяется взаимно однозначное соответствие между 2^n двоичными последовательностями длиной n и 2^n подмножествами n элементов. Для $n=3$ метод двоичного счета генерирует подмножества в следующем порядке: {}, {3}, {2}, {2, 3}, {1}, {1, 3}, {1, 2}, {1, 2, 3}.

  Это двоичное представление является ключом к решению всех задач генерирования подмножеств. Чтобы генерировать все подмножества, мы просто ведем счет от 0 до 2^n-1. Для каждого целого числа последовательно маскируем биты и составляем подмножество из элементов, соответствующих установленным битам. Чтобы создать следующее или предшествующее подмножество, просто увеличиваем или уменьшаем на единицу число текущего подмножества. Процедура маскирования как раз и является функцией, обратной ранжированию подмножества, в то время как функция ранжирования создает двоичное число, в котором установленные биты соответствуют элементам S, после чего преобразует это двоичное число в целое число.

  Чтобы создать случайное подмножество, можно сгенерировать случайное целое число в диапазоне от 0 до 2^n-1 и выполнить по нему операцию, обратную ранжированию. Но в зависимости от способа округления, применяемого в генераторе случайных чисел, некоторые подмножества могут никогда не появиться. Будет намного лучше бросить монету n раз и по результату i-го броска решить, включать или нет

элемент i в подмножество. Бросок монеты можно надежно смоделировать, генерируя случайное действительное или очень большое целое число и проверяя, превышает ли оно центральное значение своего диапазона. Таким образом, для представления подмножеств можно использовать массив булевых значений из n элементов, соответствующий целому числу с заранее наложенной маской.

На практике необходимость генерирования подмножеств часто возникает в работе с двумя тесно связанными друг с другом структурами данных — k-подмножествами и строками.

- Вместо создания всех подмножеств нам могут требоваться только подмножества из k элементов. Число таких подмножеств равно $\binom{n}{k}$, что значительно меньше, чем 2^n, особенно для небольших значений k.

  Самым лучшим способом конструирования всех k-подмножеств будет генерирование их в лексикографическом порядке. Функция ранжирования основана на том обстоятельстве, что существует $\binom{n-f}{k-1}$ k-подмножеств, у которых наименьший элемент равен f. Отсюда мы можем определить наименьший элемент в m-м k-подмножестве n элементов. Далее рекурсивно делаем то же самое с последующими элементами подмножества. Подробности см. в *подразделе "Реализации"*.

- Генерирование всех подмножеств равнозначно генерированию всех 2^n строк из значений ИСТИНА и ЛОЖЬ. Те же самые основные методы применимы к задаче генерирования всех случайных строк на алфавитах размером α, за исключением, что общее количество строк будет равным α^n.

Реализации. Генераторы как подмножеств, так и k-подмножеств в лексикографическом порядке и в порядке кода Грея рассматриваются в книге [KS99]. Реализации этих генераторов на языке C можно загрузить с веб-сайта **http://www.math.mtu.edu/~kreher/cages/Src.html**.

Профессор Фрэнк Раски (Frank Ruskey) из университета г. Виктория в Канаде разработал сервер комбинаторных объектов для генерирования перестановок, подмножеств, разбиений, графов и других комбинаторных объектов. Сервер доступен по адресу **http://theory.cs.uvic.ca**. Интерактивный интерфейс сервера позволяет указывать объекты, которые вы хотите получить. Для некоторых типов объектов имеются реализации на языках C, Pascal и Java.

Веб-сайт **http://www.jjj.de/fxt/** содержит процедуры на языке C++ для генерирования удивительно широкого диапазона комбинаторных объектов, включая подмножества и k-подмножества.

Книга [NW78] уже многие годы является замечательным источником информации по генерированию комбинаторных объектов. Она содержит эффективные реализации алгоритмов (на языке FORTRAN) для создания случайных подмножеств и для генерирования подмножеств в порядке кода Грея и в лексикографическом порядке. Кроме того, в ней приводятся процедуры для создания случайных k-подмножеств и для генерирования подмножеств в лексикографическом порядке. Алгоритм 515 (см. [BL77]) из коллекции ACM реализован на языке FORTRAN и предназначен для генерирования k-подмножеств в лексикографическом порядке. Подробности см. в *разделе 19.1.3*.

Библиотека Combinatorica (см. [PS03]) предоставляет реализации алгоритмов (на языке программирования пакета Mathematica) для создания случайных подмножеств и для генерирования последовательностей подмножеств с помощью Грея, методом двоичного счета и в лексикографическом порядке. Также предоставляются процедуры для создания случайных k-подмножеств и для генерирования последовательностей подмножеств в лексикографическом порядке. Дополнительную информацию по Combinatorica см. в *разделе 19.1.9*.

Примечания

Самым лучшим на сегодня справочным материалом по генерированию подмножеств является одна из последних работ Дональда Кнута [Knu05b]. Хорошие описания можно найти в [KS99], [NW78] и [Rus03]. В книге [Wil89] приводится более свежая информация, чем в [NW78], в том числе содержится подробное обсуждение современных проблем генерирования кода Грея.

Первоначально коды Грея были разработаны для обеспечения надежности передачи цифровых сигналов по аналоговому каналу (см. [Gra53]). При установке кодовых слов в порядке кода Грея i-е слово лишь слегка отличается от $(i + 1)$-го слова, так что незначительные колебания уровня аналогового сигнала искажают лишь небольшое количество бит. Коды Грея довольно точно соответствуют гамильтоновым циклам на гиперкубе. В работе [Sav97] представлен отличный обзор кодов Грея (упорядочения по принципу минимального изменения) для большого класса комбинаторных объектов, включая подмножества.

Популярная головоломка Spinout, выпускаемая компанией ThinkFun (ранее носившей название Binary Arts Corporation), решается на основе идей кодов Грея.

Родственные задачи. Генерирование перестановок (см. *раздел 14.4*), генерирование разбиений (см. *раздел 14.6*).

14.6. Генерирование разбиений

Вход. Целое число n.

Задача. Создать: все возможные разбиения целого числа или множества размером n, или случайное разбиение, или очередное разбиение (рис. 14.6).

Обсуждение. Существует два типа комбинаторных объектов, обозначаемых словом "разбиение", а именно разбиения целых чисел и разбиения множеств. Вам необходимо понимать разницу между ними.

- *Разбиения целого числа n* представляют собой мультимножества ненулевых целых чисел, сумма которых равна n. Например, число 5 имеет семь разных разбиений: {5}, {4, 1}, {3, 2}, {3, 1, 1}, {2, 2, 1}, {2, 1, 1, 1} и {1, 1, 1, 1, 1}. Одним из интересных приложений, требующим генерирования разбиений целых чисел, с которым мне пришлось сталкиваться, было моделирование расщепления ядра. При расщеплении атома его ядро разбивается на множество кластеров меньшего размера. Общее количество частиц в этом множестве кластеров должно быть равным первоначальному количеству частиц ядра. Таким образом, целочисленные разбиения этого первоначального количества частиц представляют все возможные способы расщепления атома.

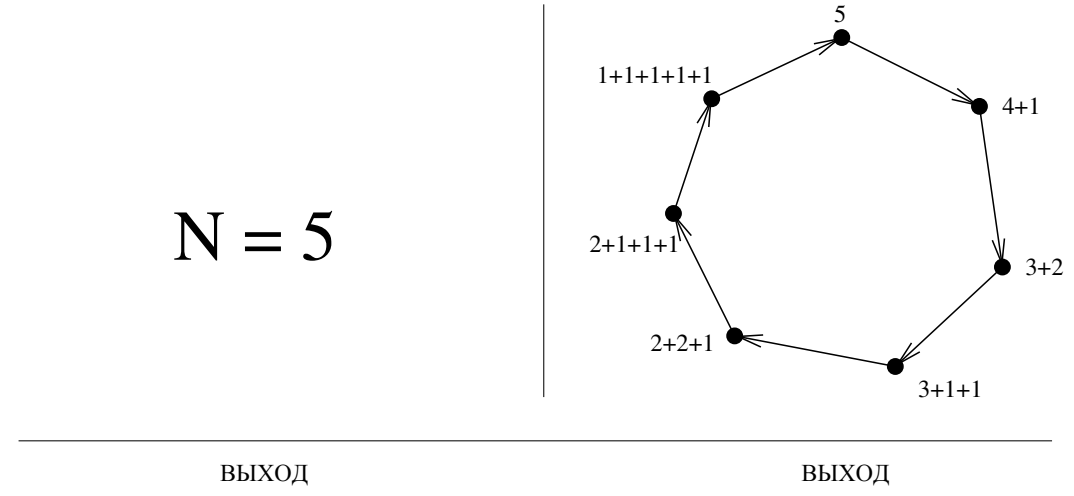


Рис. 14.6. Разбиения

- *Разбиения множества* разделяют множество {1, ..., *n*} на непустые подмножества. Например, для множества *n* = 4 существует 15 разных разбиений: {1234}, {123,4}, {124,3}, {12,34}, {12,3,4}, {134,2}, {13,24}, {13,2,4}, {14,23}, {1,234}, {1,23,4}, {14,2,3}, {1,24,3}, {1,2,34} и {1,2,3,4}. Некоторые алгоритмические задачи, включая задачу раскраски вершин/ребер и задачу поиска компонент связности, имеют в качестве результатов разбиения множества.

Для краткости мы будем называть разбиения целых чисел просто разбиениями, а для другого вида употреблять полное название — разбиение множеств. А полное название разбиения целого числа будем употреблять в тех случаях, где нужно избежать недоразумения, какое именно из разбиений имеется в виду.

Количество разбиений возрастает экспоненциально по отношению к *n*. Так, для целого числа *n* = 20 существует всего лишь 627 разбиений. Более того, возможно перечислить даже все разбиения числа *n* = 100, т. к. их количество равно 190 569 292.

Самым простым способом создания разбиений будет генерирование их в порядке, обратном лексикографическому. Первым разбиением будет само число {*n*}. Общее правило состоит в вычитании 1 из наименьшего элемента разбиения, большего единицы, с последующим объединением всех единиц с целью соответствия новому наименьшему элементу, большему единицы. Например, разбиением, следующим после {4, 3, 3,3, 1, 1, 1, 1}, будет {4, 3, 3, 2, 2, 2, 1}, т. к. пять единиц, получаемые после операции вычитания 3 – 1 = 2, лучше всего объединяются в виде элемента {2, 2, 1}. Когда мы получаем разбиение, все члены которого являются единицами, мы завершаем первый проход по всем разбиениям.

Этот алгоритм достаточно сложен для программирования, поэтому следует рассмотреть использование одной из готовых реализаций, упоминаемых далее. В любом случае, проверьте, что используемая реализация выдает ровно 627 разных разбиений для *n* = 20.

Задача генерирования случайных разбиений с равномерным распределением более сложна, чем аналогичная задача для перестановок или подмножеств. Это потому, что

выбор первого (т. е. самого большого) элемента разбиения сильно влияет на количество разбиений, которые можно сгенерировать. Обратите внимание, что независимо от размера n, существует только одно разбиение n, наибольший элемент которого равен 1. Количество разбиений числа n с наибольшим элементом, не привышающим k, задается следующим рекуррентным соотношением:

$$P_{n,k} = P_{n-k,k} + P_{n,k-1}$$

с двумя граничными условиями $P_{x-y,x} = P_{x-y,x-y}$ и $P_{n,1} = 1$. Посредством этой функции можно выбрать наибольший элемент случайного разбиения с требуемой вероятностью, после чего рекурсивно создать случайное разбиение целиком.

При генерировании случайных разбиений наблюдается тенденция появления огромного количества сравнительно небольших элементов, как можно видеть в диаграмме Феррерса (Ferrers diagram) на рис. 14.7.

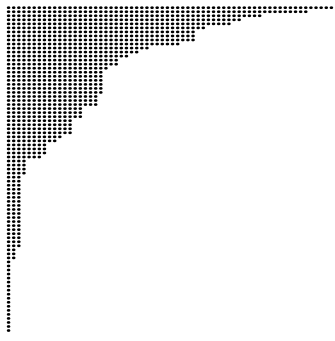


Рис. 14.7. Диаграмма Феррерса для случайного разбиения числа $n = 1000$

Каждая строка диаграммы соответствует одному элементу разбиения, размер которого представлен количеством точек в данной строке.

Разбиения множества можно генерировать, используя методы, подобные методам генерирования разбиений целых чисел. Каждое разбиение множества представляется в виде функции ограниченного роста $a_1, ..., a_n$, где $a_1 = 0$ и $a_i \leq 1 + \max(a_1, ..., a_i)$ для $i = 2, ..., n$. Каждая цифра определяет подмножество (или блок) разбиения, в то время как условие роста обеспечивает сортировку блоков в каноническом порядке по наименьшему элементу в каждом блоке. Например, функция ограниченного роста 0, 1, 1, 2, 0, 3, 1 определяет такое разбиение множества: {{1, 5},{2, 3, 7},{4},{6}}.

Так как между разбиениями множеств и функциями ограниченного роста существует взаимно однозначное соответствие, мы можем воспользоваться лексикографическим порядком функций ограниченного роста для упорядочивания разбиений множеств. В самом деле, 15 разбиений множества {1, 2, 3, 4}, перечисленных в определении разбиения множества в начале рассмотрения, расположены в соответствии с лексикографическим порядком их функций ограниченного роста (вы можете в этом убедиться).

Для генерирования случайных разбиений множества можно использовать метод двоичного счета, аналогичный используемому для генерирования разбиений целых чисел.

Числами Стирлинга второго рода $\binom{n}{k}$ называется количество разбиений n-элементного

множества $\{1, ..., n\}$ на k непустых подмножеств. Для их вычисления используется следующее рекуррентное соотношение:

$$\left\{ {n \atop k} \right\} = \left\{ {n-1 \atop k-1} \right\} + k \left\{ {n-1 \atop k} \right\}$$

с краевыми условиями $\left\{ {n \atop n} \right\} = \left\{ {n \atop 1} \right\} = 1$. Подробности см. в *подразделе "Реализации"*.

Реализации. Реализации, описываемые в книге [KS99], генерируют разбиения как целых чисел, так и множеств в лексикографическом порядке, включая функции ранжирования. Реализации этих генераторов на языке C можно загрузить с веб-сайта **http://www.math.mtu.edu/~kreher/cages/Src.html**.

Профессор Фрэнк Раски (Frank Ruskey) из университета г. Виктория в Канаде разработал сервер комбинаторных объектов (Combinatorial Object Server) для генерирования перестановок, подмножеств, разбиений, графов и других комбинаторных объектов. Сервер доступен по адресу **http://theory.cs.uvic.ca/**. Интерактивный интерфейс сервера позволяет указывать объекты, которые вы хотите получить. Для некоторых типов объектов имеются реализации на языках C, Pascal и Java.

Веб-сайт **http://www.jjj.de/fxt/** содержит процедуры на языке C++ для генерирования удивительно широкого диапазона комбинаторных объектов, включая разбиения целых чисел.

Книга [NW78] уже многие годы является замечательным источником информации по генерированию комбинаторных объектов. Она содержит эффективные реализации (на языке FORTRAN) алгоритмов для генерирования случайных и последовательных разбиений целых чисел и множеств и таблиц Юнга. Подробности см. в *разделе 19.1.10*.

Библиотека Combinatorica (см. [PS03] предоставляет реализации алгоритмов (на языке программирования пакета Mathematica) для генерирования случайных и последовательных разбиений целых чисел, строк и таблиц Юнга, а также средства для подсчета и манипулирования этими объектами. Подробности см. в *разделе 19.1.9*.

Примечания

Самым лучшим на сегодня справочным материалом по алгоритмам для генерирования разбиений как целых чисел, так и множеств является одна из последних работ Дональда Кнута [Knu05b]. Хорошие описания можно найти, например, в [KS99], [NW78], [Rus03] и [PS03]. Основным справочником по разбиениям целых чисел и родственным темам является книга [And98], а книга [AE04] представляет собой доступное введение в данную область.

Разбиения целых чисел и множеств являются частными случаями *разбиений* мультимножеств или разбиений множеств с не обязательно разными элементами. В частности, разные разбиения мультимножества $\{1, 1, 1, ..., 1\}$ в точности соответствуют разбиениям целых чисел. Разбиения мультимножеств обсуждаются в работе Дональда Кнута [Knu05b].

Длинная история генерирования комбинаторных объектов подробно излагается в работе Дональда Кнута [Knu06]. Особенно интересна связь между разбиениями множеств и японским ритуалом сжигания благовоний, а также связь между всеми 52 разбиениями множества для $n = 5$ и разными главами самого старого известного романа "Повесть о Гэндзи".

Родственными комбинаторными объектами являются таблицы Юнга и композиции целых чисел, хотя их возникновение в реальных приложениях маловероятно. Алгоритмы генерирования объектов обоих типов представлены в [NW78], [Rus03] и [PS03].

Таблицы Юнга представляют собой двумерные формирования целых чисел в диапазоне {1, ..., *n*}, где количество элементов в каждой строке определяется одним из разбиений целого числа *n*. Элементы каждой строки и каждого столбца отсортированы в возрастающем порядке, а строки выровнены по левому краю. Это понятие охватывает широкий набор разнообразных структур в виде частных случаев. Они обладают многими интересными свойствами, включая взаимно однозначное соответствие между парами таблиц и перестановок.

Композиции целых чисел представляют всевозможные способы распределения множества из *n* одинаковых шаров по *k* различным урнам. Например, три шара можно разместить по двум урнам таким образом: {3, 0}, {2, 1}, {1, 2} или {0, 3}. Композиции легче всего генерировать последовательно в лексикографическом порядке. Для генерирования случайных композиций выбираем случайное $(k-1)$-подмножество из $n+k-1$ элементов с помощью алгоритма, описанного в *разделе 14.5*, после чего подсчитываем количество оставшихся элементов между выбранными. Например, для $k=5$ и $n=10$, $(5-1)$-подмножество {1, 3, 7, 14} множества {1, ..., (n + *k*-1) = 14} определяет композицию {0, 1, 3, 6, 0}, т. к. нет элементов ни слева от элемента 1, ни справа от элемента 14.

Родственные задачи. Генерирование перестановок (см. *раздел 14.4*), генерирование подмножеств (см. *раздел 14.5*).

14.7. Генерирование графов

Вход. Параметры, описывающие граф, включая количество вершин *n* и количество ребер *m* или вероятность наличия ребра *p*.

Задача. Сгенерировать: все графы, удовлетворяющие заданным параметрам, или случайный граф, или очередной граф (рис. 14.8).

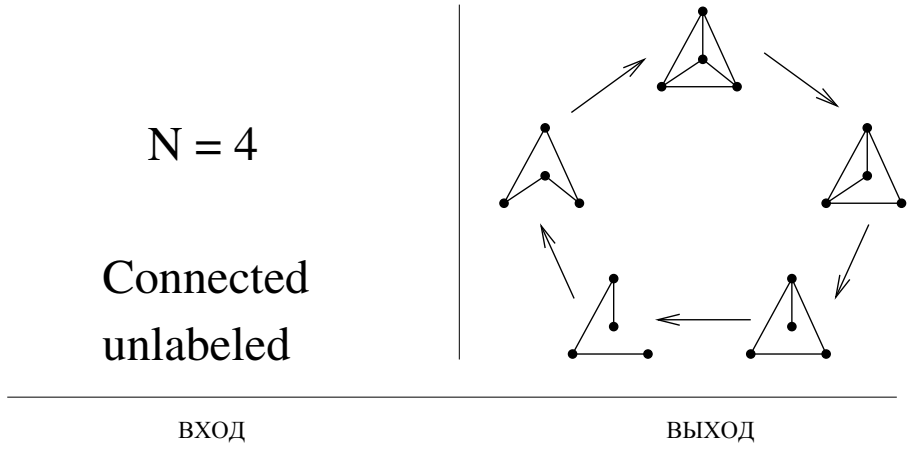


Рис. 14.8. Генерирование графов

Обсуждение. Задача генерирования графов обычно возникает при создании тестовых данных для программ. Возможно, у вас имеются две разные программы для решения

одной и той же задачи, и вы хотите узнать, какая из них работает быстрее, или удостовериться, что обе дают одинаковый результат. Другим приложением задачи является проверка, обладают ли данные графы определенным свойством, или насколько оно распространено среди них. В справедливость теоремы четырех цветов легче поверить после демонстрации четырехцветной раскраски всех планарных графов на 15 вершинах.

Еще одно приложение задачи генерирования графов возникает при проектировании сетей. Допустим, что вы должны объединить в сеть десять компьютеров, использовав кабель как можно меньшей длины. Сеть при этом должна оставаться работоспособной после одновременного выхода из строя до двух узлов. Одним из подходов в данном случае будет проверка всех возможных конфигураций сети с данным числом соединений/ребер, пока не будет найдена конфигурация, отвечающая требованиям. Но для сетей с большим количеством узлов такой подход неприемлем и, скорее всего, потребуются эвристические методы, наподобие метода имитации отжига.

Задача генерирования графов усложняется многими факторами. Прежде всего, нужно точно знать, граф какого типа требуется создать. Некоторые важные свойства графов изображены на рис. 5.2. При генерировании графов необходимо ответить на следующие вопросы.

- *Требуются помеченные или непомеченные графы?* Иными словами, имеют ли значение при сравнении графов названия вершин. При генерировании помеченных графов мы создаем все возможные маркировки для всех возможных топологий графов. А при генерировании непомеченных графов нам достаточно создать только по одному представителю каждой топологии и можно игнорировать метки. Например, из трех вершин можно создать только два связных непомеченных графа — треугольник и простой путь. Но из тех же трех вершин можно создать четыре связных помеченных графа — один треугольник и три трехвершинных пути, отличающиеся друг от друга названием центральной вершины. В общем, намного легче создавать помеченные графы. Но их так много, что очень быстро мы будем завалены изоморфными копиями одних и тех же графов.
- *Требуются ориентированные или неориентированные графы?* Большинство естественных алгоритмов генерируют неориентированные графы. Эти графы можно преобразовать в ориентированные, выполнив ориентацию ребер по броску монеты. Из любого графа можно сделать бесконтурный орграф, упорядочив случайным образом вершины в линию и направив каждое ребро слева направо. При таком разнообразии вариантов вы должны все обдумать и решить, генерируются ли все графы единообразно и случайным образом, а также насколько важен для вас способ генерирования.

Кроме прочего, нужно определиться с тем, что вы считаете случайным графом. Существуют три основные модели случайных графов, и все они генерируют графы согласно разным вероятностным распределениям:

- *генерирование случайных ребер.* Параметры этой модели задаются вероятностью наличия ребра p. Обычно значение p устанавливается равным 1/2, хотя для генерирования случайных разреженных графов можно использовать меньшие значения. Для каждой пары вершин x и y решение о добавлении ребра (x, y) принимается по

результату броска монеты. При $p = 1/2$ будут сгенерированы с одинаковой вероятностью все помеченные графы;

- *выбор случайных ребер.* Параметры этой модели задаются требуемым количеством ребер m. Модель генерирует граф, выбирая m разных ребер случайным образом с равномерным распределением. Одним из способов сделать это будет создавать случайные пары вершин (x, y) и соединять их ребрами, если данная вершина еще не в графе. Альтернативным подходом будет создать набор из $\binom{n}{2}$ возможных ребер и случайным образом выбрать из них m-подмножество, как рассматривается в *разделе 14.5*;
- *избирательное присоединение* (preferential attachment). Согласно модели "богатые становятся богаче", более вероятно, что новые ребра будут направлены к вершинам высокого уровня, а не низкого. Посмотрите, как добавляются новые ссылки (ребра) к графу веб-страниц. По любой реалистичной сетевой модели генерирования более вероятно, что очередная ссылка будет на Google, чем на **http://www.cs.sunysb.edu/~algorith**[1]. Выбор следующей соседней вершины с вероятностью, пропорциональной ее степени, порождает графы со свойствами степенной зависимости, которыми обладают многие реальные сети.

Какая из этих моделей подходит лучше всего для вашего приложения? Скорее всего, никакая. Случайные графы плохо структурированы по определению. Но графы часто используются для моделирования высокоструктурированных взаимоотношений. Интересные и легко выполнимые эксперименты на случайных графах, как правило, не отражают реальное положение вещей.

Альтернативой случайным графам являются "органические" графы, которые отображают взаимоотношения между реальными объектами. Превосходным источником "органических" графов является рассматриваемая далее база графов Stanford GraphBase. В Интернете имеется много источников взаимоотношений, которые можно превратить в интересные "органические графы", приложив немного усилий по программированию. Возьмем, например, граф, определяемый набором веб-страниц, в котором ребро определяется гиперссылкой с одной страницы на другую. Другой пример — граф, определяемый железными дорогами или авиарейсами, где вершины представляют станции или аэропорты, а ребра — прямую связь между ними. Наконец, каждая большая компьютерная программа определяет граф вызовов процедур, в котором вершины представляют процедуры, а ребра — вызовы одних процедур из других.

Два класса графов имеют особенно интересные алгоритмы генерирования:

- *деревья.* Коды Прюфера (Prufer codes) предоставляют простой способ ранжирования *помеченных* деревьев и, таким образом, решения всех стандартных задач генерирования (см. *раздел 14.4*). Для n вершин существует ровно n^{n-2} помеченных деревьев и столько же строк длиной $n - 2$ существует на алфавите $\{1, 2, ..., n\}$.

  Ключом к взаимно однозначному соответствию Прюфера является то обстоятельство, что каждое дерево имеет, по крайней мере, две вершины степени 1. Таким обра-

[1] Пожалуйста, создайте ссылку с вашей домашней страницы на наш веб-сайт, чтобы исправить эту ситуацию.

зом, в любом помеченном дереве вершина v, соединенная с листом с наименьшей меткой, является четко определенной. Пусть S_1, первый символ кода, — это вершина v. Удалим связанный лист и будем повторять эту процедуру до тех пор, пока не останутся только две вершины. Таким образом мы получим уникальный код S для любого помеченного дерева, который можно использовать для ранжирования дерева. Чтобы построить дерево по коду, воспользуемся тем обстоятельством, что степень вершины v дерева на один больше, чем количество вхождений вершины v в код S. Листом с наименьшей меткой будет целое число, отсутствующее в S, которое совместно с S_1 определяет первое ребро дерева. Применяя индукцию, получаем все дерево;

- *графы с фиксированной степенной последовательностью. Степенной последовательностью* (degree sequence) графа G называется разбиение целого числа $p = (p_1, ..., p_n)$, где p_i является степенью i-й вершины с наивысшей степенью графа. Так как каждое ребро учитывается при подсчете степени двух вершин, то p является разбиением целого числа $2m$, где m является количеством ребер в графе.

  Не все разбиения соответствуют степенным последовательностям графов. Но существует рекурсивный метод, который создает граф для данной степенной последовательности, если такой граф вообще существует. Если разбиение выполнимо, то вершину с наивысшей степенью v_1 можно соединить с вершинами со степенями, наивысшими из оставшихся, или вершинами, соответствующими частям $p_2,...,p_{p_1+1}$. Удалив p_i и уменьшив $p_2,...,p_{p_1+1}$ на единицу, мы получаем меньшее разбиение, которое мы обрабатываем рекурсивно. Если мы завершим процедуру, не создав отрицательных чисел, то разбиение является осуществимым. Так как мы всегда соединяем вершину наивысшей степени с другими вершинами высокой степени, то важно после каждой итерации переупорядочить элементы разбиения по размеру.

  Хотя этот метод является детерминистическим, с его помощью из графа G можно создать полуслучайную коллекцию графов, используя операции обмена ребер. Допустим, что ребра (x, y) и (w, z) входят в граф G, а ребра (x, w) и (y, z) — нет. Обменяв эти пары ребер, мы получим другой (не обязательно связный) граф, не меняя степень ни одной из вершин.

Реализации. База графов Stanford GraphBase (см. [Knu94]) является, пожалуй, самым хорошим генератором экземпляров графов для использования их в качестве тестовых данных для других программ. Она содержит графы, полученные путем обработки взаимоотношений персонажей известных романов, статей из словаря синонимов Роджета, визуальных характеристик "Моны Лизы", графов-расширителей, а также модели экономики США. Она также содержит процедуры генерирования двоичных деревьев, произведений графов и других операций на графах основных типов. Наконец, поскольку в ней используются аппаратно-независимые генераторы случайных чисел, созданные с ее помощью случайные графы можно воспроизвести на других машинах, что позволяет использовать их в качестве входных данных при экспериментальном сравнении алгоритмов. Дополнительную информацию см. в *разделе 19.1.8*.

Библиотека Combinatorica (см. [PS03]) содержит генераторы (на языке пакета Mathematica) для таких типов графов, как "звезда" и "колесо", для полного графа, для

случайных графов и деревьев, а также для графов с заданной степенной последовательностью. Кроме этого, библиотека включает в себя операции создания более сложных графов на основе этих, в том числе объединение и произведение графов.

Сервер комбинаторных объектов (**http://theory.cs.uvic.ca**) предоставляет процедуры для генерирования как свободных, так и корневых деревьев.

В работе [VL05] описывается реализация на языке C++ алгоритма генерирования простых связных графов с указанной степенной последовательностью. Код можно загрузить с веб-страницы **http://www.liafa.jussieu.fr/~fabien/generation.**

Программа Nauty для тестирования графов на изоморфизм (см. *раздел 16.9*) содержит набор программ для генерирования неизоморфных графов, а также специальные генераторы для создания двудольных графов, ориентированных графов и мультиграфов. Все эти программы можно загрузить с веб-страницы **http://cs.anu.edu.au/~bdm/nauty.**

Математики Брендан Маккей (Brendan McKay) (**http://cs.anu.edu.au/~bdm/data**) и Гордон Ройли (Gordon Royle) (**http://people.csse.uwa.edu.au/gordon/data.html**) предоставляют на своих веб-страницах исключительно полные каталоги нескольких семейств графов и деревьев, размер которых ограничен лишь разумным количеством вершин.

В книге [NW78] предоставляются эффективные процедуры (на языке FORTRAN) перечисления всех помеченных деревьев с помощью кодов Прюфера и создания случайных непомеченных корневых деревьев. Подробности см. в *разделе 19.1.10.* В книге [KS99] описывается алгоритм на языке C для генерирования помеченных деревьев. Реализации можно загрузить с веб-сайта **http://www.math.mtu.edu/~kreher/cages/Src.html.**

Примечания

На тему генерирования случайных графов с равномерным распределением написано множество книг. Среди прочих обзоров можно назвать [Gol93] и [Tin90]. С задачей генерирования классов графов тесно связана задача их подсчета. Обзор достижений в этой области приведен в книге [HP73].

Самым лучшим на сегодня справочным материалом по генерированию деревьев является одна из последних работ Дональда Кнута [Knu06]. Взаимно однозначное соответствие между $(n-2)$-строками и помеченными деревьями было установлено немецким математиком Хайнцем Прюфером (Heinz Prufer) в 1918 г. (см. [Pru18]).

В теории случайных графов пороговые законы определяют плотность ребер, при которой становится высокой вероятность существования таких свойств, как связность. Изложение теории случайных графов можно найти в книгах [Bol01] и [JLR00].

Модель избирательного присоединения графов возникла сравнительно недавно при изучении сетей. Введение в эту интересную область содержится в книгах [Bar03] и [Wat04].

Разбиение целого числа является *графическим*, если существует простой граф с такой степенной последовательностью. В работе [EG60] приводится доказательство, что степенная последовательность является графической тогда и только тогда, когда для любого целого числа $r < n$ соблюдается следующее условие:

$$\sum_{i=1}^{r} d_i \le r(r-1) + \sum_{i=r+1}^{n} \min(r, d_i)$$

Родственные задачи. Генерирование перестановок (см. *раздел 14.4*), изоморфизм графов (см. *раздел 16.9*).

14.8. Календарные вычисления

Вход. Календарная дата *d*, заданная месяцем, днем и годом.

Задача. На какой день недели выпадает *d* в данной календарной системе (рис. 14.9)?

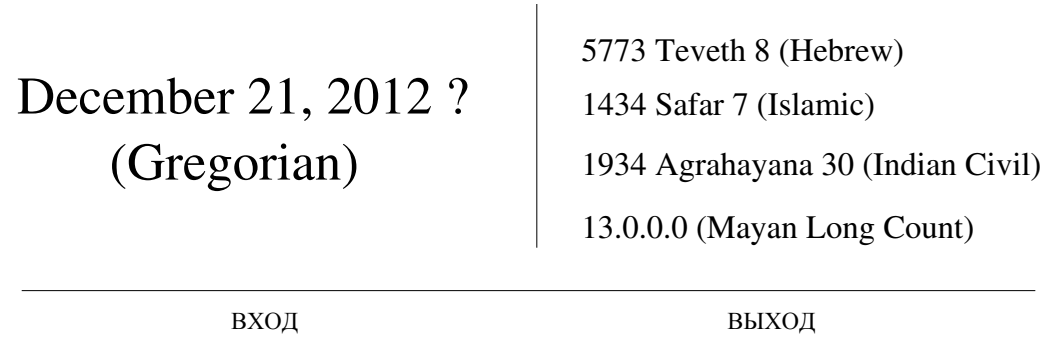


Рис. 14.9. Календарные вычисления

Обсуждение. Во многих бизнес-приложениях требуется выполнять календарные вычисления. Например, возможно нужно вывести календарь для определенного месяца и года или вычислить, в какой день недели или года произойдет какое-то событие. Важность календарных вычислений была ярко продемонстрирована "ошибкой 2000 года", присутствовавшей в старых программах, в которых для обозначения года использовались только две последние цифры.

Более сложные вопросы возникают в международных приложениях, т. к. разные народы и этнические группы используют разные календарные системы. Некоторые из этих календарных систем, такие как используемый в большинстве стран мира григорианский календарь, основаны на солнечном цикле, в то время как другие календари, например иудейский календарь, основаны на лунных циклах. А смогли бы вы назвать сегодняшнюю дату по китайскому календарю?

Календарные вычисления отличаются от других задач в этой книге, потому что календари являются историческими объектами, а не математическими. В области календарных вычислений алгоритмические вопросы касаются установления правил календарной системы и правильной их реализации, а не разработки эффективных методов вычисления.

Подход, лежащий в основе всех календарных систем, заключается в выборе начальной точки и ведении отсчета от этой точки. Разные календарные системы отличаются друг от друга специфическими правилами, определяющими конец одного месяца или года и начало другого. Для реализации календаря требуются две функции, одна из которых по данной дате возвращает количество дней, прошедших от начальной точки, а другая по данному целому числу n возвращает дату календаря, отстоящей ровно на n дней от точки начала отсчета. Эти функции аналогичны функциям ранжирования для комбинаторных объектов, таких как перестановки (см. *раздел 14.4*).

Главным источником проблем в календарных системах является то обстоятельство, что в солнечном году не целое число дней. Чтобы поддерживать синхронизацию календар-

ного года с реальным, время от времени в него нужно вносить поправки в виде високосных дней. Но так как продолжительность солнечного года составляет 365 дней, 5 часов, 49 минут и 12 секунд, то добавление високосного дня через каждые четыре года делает корректируемый календарный год на 10,8 минут длиннее.

В первоначальном юлианском календаре эти лишние минуты не принимались во внимание, и к 1582 году накопилось отставание в 10 дней. Тогда Папа Григорий XIII ввел в действие исправленный календарь, названный григорианским, который используется по сей день. Исправления состояли в одноразовом переносе даты календаря на 10 дней вперед и нового правила добавления високосных дней в годы столетий — в годы, кратные 400, но не 100. Годы, кратные четырем, по-прежнему оставались високосными. По некоторым сведениям, введение нового календаря сопровождалось бунтами среди населения. Люди считали, что их жизнь укоротилась на десять дней. Вне пределов влияния католической церкви принятие нового календаря проходило медленно. В Англии и Америке григорианский календарь был принят в 1752 г., а в Турции — в 1927 г.

Правила для большинства календарных систем достаточно сложны, вследствие чего будет разумным воспользоваться уже готовым кодом из надежного источника, вместо того, чтобы пытаться писать свою реализацию.

Существует много алгоритмов типа "удиви своих друзей", посредством которых можно в уме высчитать день недели для определенной даты. Но такие алгоритмы часто работают правильно только в определенном столетии, и их не следует реализовывать на компьютере.

Реализации. Библиотеки процедур для работы с календарями широко доступны как на языке C, так и Java. Библиотека Boost предоставляет надежную реализацию григорианского календаря на языке C++. Код можно загрузить с веб-страницы **http://www.boost.org/doc/libs/1_43_0/doc/html/date_time.html**. Стандартный пакет утилит Java java.util содержит класс `Calendar`, который реализует григорианский календарь. Любой из этих реализаций будет, скорее всего, достаточно для большинства приложений.

В книге [RD0l] описаны алгоритмы для разных календарей, включая григорианский, китайский, индийский, мусульманский и иудейский, а также календари, представляющие исторический интерес. Реализация этих календарей на языке Common Lisp, Java и Mathematica называется Calendrical и содержит процедуры преобразования дат между разными календарными системами, вычисления дня недели, а также определения светских и религиозных праздников. Пакет Calendrical является, пожалуй, самым обширным и надежным источником существующих календарных процедур. Подробности см. на веб-сайте **http://calendarists.com**.

Реализации интернациональных календарей, написанные на языках C и Java, доступны на веб-сайте SouceForge (**http://sourceforge.net**), но об их надежности ничего неизвестно. Чтобы найти требуемый календарь, выполните поиск по соответствующему ключу, например "Gregorian calendar" для григорианского календаря.

Примечания

Всесторонний обзор алгоритмов для календарных вычислений представлен в работах [DR90] и [RDC93], на основе которых была написана книга [DR01], содержащая алгорит-

мы, по крайней мере, для 25 национальных и исторических календарей. В книге [DR02] представлены календарные таблицы на 300 лет, с 1900 г. по 2200 г.

Некоторые люди обеспокоены тем, что 21 декабря 2012 г. может настать конец света. Опасение основано на том обстоятельстве, что эта дата представляет дату окончания цикла по календарю майя и переход на новый цикл 13.0.0.0.0. Читателям моей книги не следует переживать по этому поводу, т. к. я не посвятил бы столько усилий ее написанию, если бы конец света был так близок. Авторитетное описание календаря майя дается в книге [RD01].

Родственные задачи. Арифметические операции с произвольной точностью (см. *раздел 13.9*), генерирование перестановок (см. *раздел 14.4*).

14.9. Календарное планирование

Вход. Бесконтурный орграф $G = (V, E)$, в котором вершины представляют задания, а ребро (u, v) отражает тот факт, что задание u должно быть завершено перед заданием v.

Задача. Составить такое расписание выполнения всех заданий, которое минимизирует время выполнения или количество процессоров (рис. 14.10).

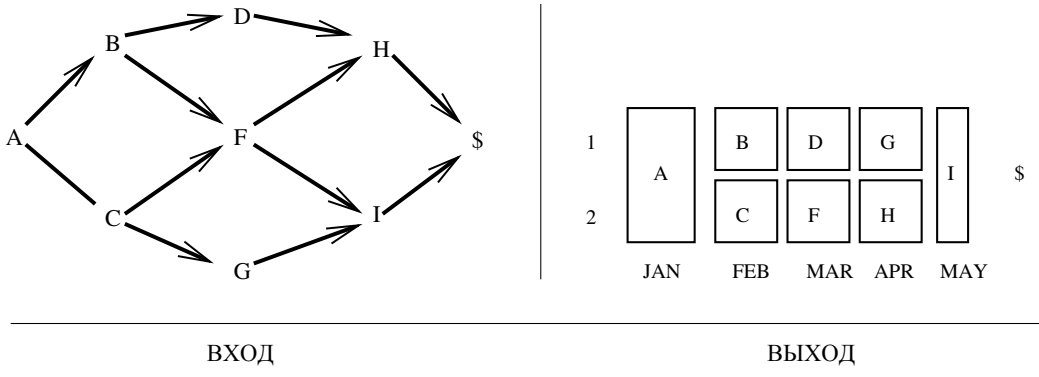


Рис. 14.10. Задача календарного планирования

Обсуждение. Задача разработки расписания, удовлетворяющего набору ограничивающих условий, является фундаментальной для многих приложений. Распределение заданий между процессорами является критическим аспектом любой системы параллельной обработки данных. Плохое планирование может привести к простаиванию всех процессоров, кроме того, который выполняет задание, представляющее собой узкое место. Другими примерами задач календарного планирования являются выдача заданий рабочим, размещение студенческих групп по аудиториям или составление расписания экзаменов.

Задачи календарного планирования различаются по типам ограничивающих условий и видам составляемых расписаний. Некоторые из рассматриваемых в каталоге задач имеют отношение к различным типам календарного планирования. Говоря более конкретно:

- посредством топологической сортировки можно создать расписание, отвечающее ограничивающим условиям очередности. Подробности см. в *разделе 15.2*;

- паросочетание в двудольных графах можно использовать для сопоставления набора заданий с рабочими, имеющими навыки, требуемые для их выполнения. Подробности см. в *разделе 15.6*;
- с помощью раскраски ребер и вершин можно сопоставить набор заданий с временными периодами таким образом, чтобы избежать попадания конфликтных заданий в один и тот же временной интервал. Подробности см. в *разделах 16.7* и *16.8*;
- алгоритм коммивояжера можно использовать, чтобы выбрать оптимальный маршрут для посещения указанных адресов разносчиком пиццы. Подробности см. в *разделе 16.4*;
- алгоритм поиска эйлерова цикла можно использовать для построения наиболее эффективного пути следования снегоуборочной машины, проходящей по всем заданным улицам. Подробности см. в *разделе 15.7*.

Рассмотрим задачу календарного планирования с условиями очередности для бесконтурных орграфов. Допустим, что вы разбили большую работу на несколько меньших заданий. Для каждого задания известно время, требуемое для его выполнения (или, возможно, верхний временной предел). Кроме этого, для каждой пары заданий известно, необходимо ли, чтобы одно из них выполнялось перед другим. Чем меньше ограничивающих условий, тем лучшее расписание можно получить. Эти ограничивающие условия должны определять бесконтурный орграф, т. к. цикл при ограничивающих условиях очередности создает конфликт, который невозможно разрешить.

Нас интересуют следующие задачи:

- *критический путь*. Критический путь — это самый длинный путь от начальной вершины до конечной. Эта информация может быть важной, т. к. единственным способом сократить общее время выполнения проекта будет сокращение времени выполнения одного задания в каждом критическом пути. Задания в критических путях можно определить за время $O(n+m)$, используя методы динамического программирования, рассматриваемые в *разделе 15.4*;
- *минимальное время завершения*. Самое короткое время, за которое можно завершить данный проект, соблюдая при этом условия очередности и предполагая наличие неограниченного количества исполнителей. При отсутствии условий очередности все задания можно было бы выполнять одновременно, а общее время было бы равно времени, требуемого для выполнения самого длительного из них. А если на отдельные задания наложены строгие ограничивающие условия, согласно которым каждое задание должно выполняться только после выполнения его непосредственного предшественника, то минимальное время завершения будет равно сумме времен выполнения всех заданий.

  Для бесконтурного орграфа минимальное время выполнения можно вычислить за время $O(n+m)$. Это время зависит от критического пути. Чтобы составить расписание, перебирайте задания в топологическом порядке. Каждое задание должно запускаться на новом процессоре, как только завершится выполнение непосредственно предшествующего ему задания;
- *компромисс между количеством исполнителей и временем завершения?* Мы заинтересованы в быстрейшем выполнении работы при заданном количестве исполнителей. К сожалению, эта и большинство подобных ей задач являются NP-полными.

Любое реальное приложение календарного планирования будет, скорее всего, иметь комбинацию ограничивающих условий, смоделировать которые посредством описанных методов будет трудно или невозможно. Существуют два приемлемых подхода к решению таких задач. В первом случае мы игнорируем некоторые ограничивающие условия, пока задача не сведется к одному из описанных выше типов, решаем задачу, а потом оцениваем полученное решение с учетом других ограничивающих условий. Возможно, что полученное расписание можно будет легко подправить вручную, чтобы оно удовлетворяло таким негласным ограничениям, как запрещение совместной работы двум работникам, недолюбливающим друг друга. Другий подход — сформулировать задачу календарного планирования во всей ее сложности в терминах линейного целочисленного программирования (см. *раздел 13.6*). Я рекомендую сначала применить самый простой способ и посмотреть, что из этого получится, и только потом переходить к более сложным методам.

Другая фундаментальная задача календарного планирования заключается в распределении набора заданий без ограничивающих условий очередности между идентичными исполнителями таким образом, чтобы минимизировать общее время исполнения. Например, в типографии требуется распределить несколько работ между несколькими копировальными машинами таким образом, чтобы завершить работу к концу рабочего дня. Такую задачу можно смоделировать как задачу разложения по контейнерам (см. *раздел 17.9*), где каждому заданию присваивается числовое значение, равное количеству часов, требуемому для ее завершения, а каждая машина представляется контейнером с емкостью, равной количеству рабочих часов.

В более сложных вариантах задачи календарного планирования для каждого задания указывается время, раньше которого нельзя начинать выполнение, и крайний срок завершения. Для решения задач календарного планирования существуют эффективные эвристические алгоритмы, основанные на сортировке заданий по размеру и требуемому времени завершения. Дополнительную информацию см. в *подразделах "Реализации"* и *"Примечания"*. Обратите внимание на то, что эти задачи календарного планирования становятся сложными только в том случае, если задания нельзя распределить по нескольким машинам или прерывать их исполнение с последующим возобновлением. Если ваше приложение дает вам высокую степень свободы, то ею следует воспользоваться.

Реализации. JOBSHOP представляет собой коллекцию программ на языке C для решения задач календарного планирования на основе вычислительного изучения, представленного в работе [AC91]. Коллекцию можно загрузить с веб-сайта **http://www2.isye.gatech.edu/~wcook/jobshop**.

Программа с открытым кодом Tablix (**http://tablix.org**) предназначена для решения задач создания расписаний в настоящих учебных заведениях. Поддерживает параллельные и кластерные вычисления.

Программа календарного планирования LEKIN (см. [Pin02]) предназначена для образовательный целей. Программа поддерживает планирование как для одной, так и для нескольких машин. Загрузить программу можно с веб-сайта **http://www.stern.nyu.edu/om/software/lekin/**.

Для коммерческих приложений планирования доступна современная программа ILOG CP. Дополнительную информацию можно найти на веб-сайте **http://www.ilog.com/products/cp**.

Алгоритм 520 (см. [WBCS77]) из коллекции алгоритмов ACM представляет собой программу на языке FORTRAN для планирования расписания сети с множественными ресурсами. Алгоритм является частью библиотеки NetLib (см. *раздел 19.1.5*).

Примечания

Существует огромное количество литературы об алгоритмах планирования. Подробный обзор достижений в этой области дается в книгах [Bru07] и [Pin02]. Подборка свежих обзоров по всем аспектам календарного планирования приводится в книге [LA04].

Имеется хорошо определенная классификация нескольких тысяч вариантов расписаний по машинной среде, по особенностям обработки и ограничивающим условиям, а также по параметрам, подлежащим минимизации. Обзоры содержатся в книгах [Bru07], [CPW98], [LLK83] и [Pin02].

Диаграммы Ганта предоставляют визуальные представления решений распределения работ по машинам, где ось абсцисс представляет время, а разные машины отображаются в разных рядах. Пример диаграммы Ганта показан на рис. 14.10. На этом рисунке каждая запланированная работа представлена в виде прямоугольника, определяющего ее исполнителя и время начала и окончания. Методы календарного планирования проектов с ограничивающими условиями очередности часто называют системой PERT/CPM (Program Evaluation and Review Technique/Critical Path Method, система оценки и пересмотра планов/Метод критического пути). Как диаграммы Ганта, так и система PERT/CPM обсуждаются в большинстве учебников по исследованию операций, включая [Pin02].

При рассмотрении задач календарного планирования уроков и других подобных задач часто используется термин *составление расписания* (timetabling). На проводящейся дважды в год конференции PATAT (Practice and Theory of Automated Timetabling, практика и теория автоматического составления расписания) обсуждаются новые результаты в этой области. Протоколы конференции доступны на веб-сайте **http://www.asap.cs.nott.ac.uk/patat/patat-index.shtml**.

Родственные задачи. Топологическая сортировка (см. *раздел 15.2*), паросочетание (см. *раздел 15.6*), вершинная раскраска (см. *раздел 16.7*), реберная раскраска (см. *раздел 16.8*), разложение по контейнерам (см. *раздел 17.9*).

14.10. Выполнимость

Вход. Набор дизъюнкций в конъюнктивной нормальной форме.

Задача. Определить, существует ли такой набор значений истинности булевых переменных, для которого одновременно выполняются все дизъюнкции (рис. 4.11).

Обсуждение. Задача выполнимости возникает в ситуации, когда нужно найти конфигурацию или объект, который должен удовлетворять набору логических ограничений. Задача выполнимости, в основном, применяется для проверки того, что проектируемая аппаратная или программная система работает должным образом для всех входных экземпляров. Допустим, что данная логическая формула $S(\bar{X})$ обозначает указанный результат на входных переменных $\bar{X} = X_1, ..., X_n$, а другая формула $C(\bar{X})$ обозначает

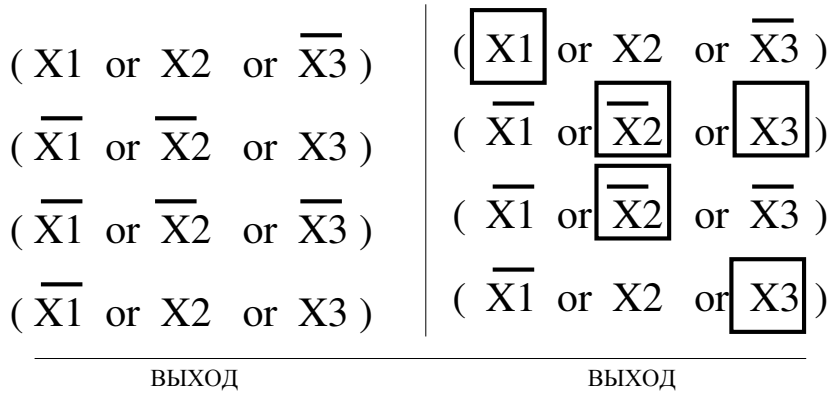


Рис. 4.11. Задача выполнимости

логику предлагаемой схемы вычисления $S(X)$. Схема будет правильной в том случае, если нет такого значения $\overline{X}$, для которого $S(\overline{X}) \neq C(\overline{X})$.

Задача выполнимости (или задача SAT) является *самой первой* NP-полной задачей. Хотя она применяется на практике в таких областях, как выполнимость ограничивающих условий, логика и автоматическое доказательство теорем, она имеет большую теоретическую значимость как корневая задача, на основе которой строятся все другие доказательства NP-полноты. В создание лучших из современных систем решения задач SAT было вложено столько усилий, что их следует использовать, когда вам действительно нужно *точно* решить NP-полную задачу. Однако самым практичным подходом обычно является применение эвристических алгоритмов, которые дают хорошие, хотя и не оптимальные результаты.

При проверке выполнимости необходимо найти ответы на следующие вопросы.

- *В какой форме представлена ваша формула?* В задаче выполнимости ограничивающие условия указываются в виде логической формулы. Существуют два основных способа выражения логических формул — в конъюнктивной нормальной форме (КНФ) и дизъюнктивной нормальной форме (ДНФ). Формулы в КНФ состоят из конъюнкций (операция И) дизъюнкций (операция ИЛИ), как в следующем примере:

  $(v_i$ ИЛИ $\overline{v}_2)$ И $(v_2$ ИЛИ $v_3)$

  Для выполнения формулы в КНФ нужно выполнить все дизъюнкции.

  Формулы в ДНФ состоят из дизъюнкций (операция ИЛИ) конъюнкций (операция И). Предыдущую формулу можно записать в ДНФ таким образом:

  $(\overline{v}_1$ И $\overline{v}_2$ И $v_3)$ ИЛИ $(\overline{v}_1$ И v_2 И $\overline{v}_3)$ ИЛИ $(\overline{v}_1$ И v_2 И $v_3)$ ИЛИ $(v_1$ И $\overline{v}_2$ И $v_2)$

  Для выполнения формулы в ДНФ нужно выполнить одну из конъюнкций.

  Решение задачи разрешимости формулы в ДНФ является тривиальным, т. к. любая формула ДНФ является выполнимой, если только *каждая* конъюнкция не содержит как литерал, так его дополнение (отрицание). Но задача выполнимости формулы в КНФ является NP-полной. Это кажется парадоксальным, т. к. посредством законов де Моргана можно преобразовать формулу из КНФ в ДНФ и наоборот. Причина за-

ключается в том, что в процессе такого преобразования приходится создавать новые члены, количество которых может возрастать экспоненциально, вследствие чего само преобразование не может быть выполнено за полиномиальное время.

- *Каков размер дизъюнкций?* Задача k-SAT является частным случаем задачи выполнимости, в которой каждая дизъюнкция состоит из, самое большее, k переменных. Задача 1-SAT является тривиальной, т. к. для ее решения достаточно, чтобы истинное значение имела любая переменная в любой дизъюнкции. Задача 2-SAT тривиальной не является, но все же ее можно решить за линейное время. Этот факт представляет интерес, т. к. при определенной изобретательности удается моделировать некоторые задачи в виде задачи 2-SAT. Впрочем, задачи с дизъюнкциями, содержащими три переменные (т. е. 3-SAT), являются NP-полными.
- *Достаточно ли выполнить только одну из дизъюнкций?* Если требуется точное решение, то вам в любом случае придется использовать какой-нибудь алгоритм перебора с возвратом, например, метод Дэвиса-Путнама (Davis-Putnam). В наихудшем случае потребуется проверить 2^m наборов значений истинности, но, к счастью, существует много способов сократить пространство поиска. Хотя теоретически задача выполнимости является NP-полной, трудность конкретной задачи зависит от способа создания экземпляров. Естественно определенные "случайные" экземпляры часто решаются с неожиданной легкостью; более того, задача генерирования по-настоящему сложных экземпляров является нетривиальной.

  Тем не менее, иногда полезно сделать задачу менее строгой, поставив в качестве цели выполнимость наибольшего количества дизъюнкций. Качество решений, полученных случайным методом или с помощью эвристического алгоритма, можно улучшить с помощью оптимизационных методов, таких как метод имитации отжига. В самом деле, при присвоении переменным любого случайного набора значений истинности вероятность выполнимости каждой k-SAT дизъюнкции составляет $1-(1/2)^k$, так что, скорее всего, с первой попытки выполнится большинство дизъюнкций. Однако завершить работу будет труднее. Задача поиска набора значений истинности, удовлетворяющего наибольшему количеству дизъюнкций, является NP-полной даже для экземпляров, не имеющих решения.

Когда мы сталкиваемся с задачей неизвестной сложности, доказательство ее NP-полноты может оказаться важным первым шагом в ее решении. Если вы думаете, что ваша задача может быть сложной, посмотрите, нет ли ее в списке сложных задач в книге [GJ79]. Если нет, то я рекомендую попробовать самостоятельно доказать сложность задачи, используя базовые задачи 3-SAT, вершинного покрытия, независимого множества, разбиения целого числа, клики и гамильтонова цикла. Я советую начинать именно с этих задач, а не со сложных задач из книги [GJ79]. Кроме того, это поможет вам лучше разобраться в вопросе, поскольку сложность вашей задачи не будет заслонена доказательством сложности задачи из книги. Стратегии доказательства сложности рассматриваются в *главе 9*.

Реализации. В последние годы наблюдался заметный рост производительности средств решения задач SAT. На ежегодном соревновании по решению задач SAT определяются лучшие средства решения для трех категорий экземпляров задач — промышленного, ручного и случайного.

В соревновании SAT в 2007 г. наилучшими решателями промышленных задач были Rsat (**http://reasoning.cs.ucla.edu/rsat**), PicoSAT (**http://fmv.jku.at/picosat**) и MiniSAT (**http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat**). Исходный код этих трех решателей и другая информация доступны на веб-странице **http://www.satcompetition.org**.

Веб-сайт SAT Live! (**http://www.satlive.org**) — источник самых свежих научных работ, программ и тестовых наборов для задач выполнимости и родственных задач оптимизации логических функций.

Примечания

Наиболее полный обзор тестирования выполнимости представлен в книге [KSBD07]. Алгоритм обхода с возвратом DPLL (Davis-Putnam-Logemann-Loveland) для решения задач выполнимости был представлен в 1962 г. Методы локального поиска работают лучше на определенном классе задач, трудных для решателей DPLL. Особенно популярным является решатель Chaff (см. [MMZ+01]), который можно загрузить с веб-сайта **http://www.princeton.edu/~chaff/**. Обзор последних достижений в области тестирования выполнимости представлен в работе [KS07].

В работе [DGH+02] описывается алгоритм для решения задачи 3-SAT за время $O^*(1{,}4802^n)$ в наихудшем случае. Обзор эффективных (но имеющих не полиномиальное время исполнения) алгоритмов решения NP-полных задач приводится в работе [Woe03].

Основным справочником по NP-полноте является книга [GJ79], содержащая список около 400 NP-полных задач. Уже долгие годы эта книга остается чрезвычайно полезным справочником. Я обращаюсь к ней чаще, чем к какой-либо другой. Обновления к книге предоставляются в колонке Дэвида Джонсона (David Johnson), публикующейся время от времени в журналах "Journal of Algorithms" и (в последнее время) "ACM Transactions on Algorithms".

Обсуждению теоремы Кука (см. [Coo71]), доказывающей сложность задачи выполнимости, посвящены работы [CLRS01], [GJ79] и [KT06]. Важность результатов работы Кука становится ясной из доклада Карпа (Karp) (см. [Kar72]), в котором доказывается сложность свыше 20 разных комбинаторных задач.

В работе [APT79] представлен алгоритм решения задач 2-SAT за линейное время. Интересное применение задачи 2-SAT для нанесения обозначений на карты излагается в работе [WW95]. Самый лучший известный эвристический алгоритм выдает приблизительное решение задачи MAX-2-SAT с точностью до 1,0741 (см. [FG95]).

Родственные задачи. Условная оптимизация (см. *раздел 13.5*), задача коммивояжера (см. *раздел 16.4*).

ГЛАВА 15

Задачи на графах с полиномиальным временем исполнения

Алгоритмические задачи на графах составляют приблизительно треть всех задач в этом каталоге. Многие задачи из других разделов можно было бы сформулировать с тем же успехом в виде графов, например задачу минимизации ширины ленты и задачу оптимизации конечного автомата. Одним из основных навыков хорошего алгориста является способность определить название задачи теории графов. Если вы знаете название своей задачи, вы найдете в каталоге точные инструкции, как приступить к ее решению.

В этой главе рассматриваются только те задачи, для решения которых существуют эффективные алгоритмы. Так как часто существует несколько способов моделирования приложения, то имеет смысл, прежде чем использовать одну из более трудных постановок решаемой задачи, поискать в этой главе способ ее моделирования.

Время исполнения представленных в этой главе алгоритмов возрастает полиномиально с увеличением размера графа. Буквой n обозначается количество вершин графа, а буквой m — количество его ребер.

Графы легче всего воспринимать, когда они представлены в виде рисунков. Поэтому мы также рассмотрим алгоритмы рисования графов, деревьев и планарных графов.

Большинство сложных алгоритмов на графах трудно программировать. Но существуют уже готовые реализации, нужно только знать, где их можно найти. Лучшие источники реализаций алгоритмов на графах — библиотека LEDA (см. [MN99]) и библиотека Boost Graph Library (см. [SLL02]). Но для многих задач существуют специальные программы.

Самые свежие обзоры разнообразных алгоритмов для работы с графами приведены в книге [TNX08]. Другие интересные обзоры можно найти в книге [vL90a] и некоторых главах из книги [Ata98]. Среди специализированных книг по алгоритмам обработки графов можно порекомендовать следующие:

- [Sed98] — том этого учебника, посвященный алгоритмам для работы с графами, содержит всеобъемлющее, но доступное введение в эту область;
- [AMO93] — хотя эта книга посвящена, в основном, потокам в сетях, в ней рассматривается целый набор алгоритмов, причем особое внимание уделяется исследованию операций;
- [Gib85] — хорошая книга по разнообразным алгоритмам для работы с графами, включая проверку планарности, поиск паросочетаний, эйлеровых и гамильтоновых циклов, а также более простые темы;

♦ [Eve79a] — уже не новый, но от этого не менее ценный учебник по алгоритмам для работы с графами, содержащий подробное обсуждение алгоритмов для проверки планарности графов.

15.1. Компоненты связности

Вход. Ориентированный или неориентированный граф G.

Задача. Разбить граф G на части, или компоненты, так, чтобы вершины x и y принадлежали разным компонентам, если из x в y нет пути (рис. 15.1).

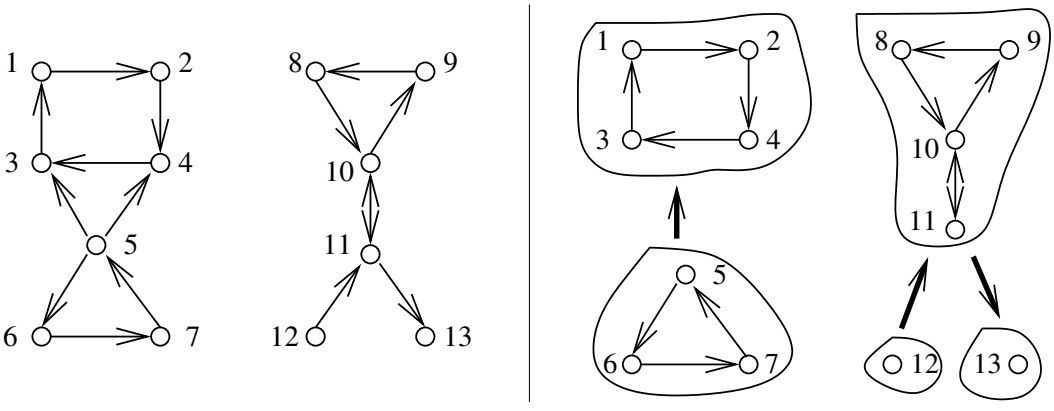


Рис. 15.1. Компоненты связности

Обсуждение. Компоненты связности графа представляют части графа. Две вершины находятся в одной и той же компоненте графа G тогда и только тогда, когда между ними существует какой-либо путь.

Задача поиска компонент связности является центральной во многих приложениях на графах. Рассмотрим, например, задачу определения естественных кластеров в наборе элементов. Для решения задачи мы представляем каждый элемент в виде вершины и соединяем ребром каждую пару элементов, которые считаются "подобными". Компоненты связности этого графа соответствуют разным классам элементов.

Проверка графа на наличие в нем компонент связности является важным шагом предварительной обработки. При исполнении алгоритма только на одной компоненте несвязного графа часто удается выявить скрытые, труднообнаруживаемые ошибки. Проверка на связность выполняется так легко и быстро, что следует всегда проводить ее для графа входа, даже если вы уверены, что он *должен* быть связным.

Связность любого неориентированного графа можно проверить посредством обхода графа в глубину или в ширину (см. *главу 5*). В действительности, не так важно, какой из этих методов обхода вы выберете. При любом обходе поле "номер компоненты" для каждой вершины инициализируется нулевым значением, а потом начинается поиск

первой компоненты из вершины v_1. При открытии каждой вершины этому полю присваивается значение номера текущей компоненты. По завершении первоначального обхода номер компоненты увеличивается на единицу, и поиск выполняется снова, начиная с первой вершины, у которой номер компоненты все еще равен нулю. При правильной реализации с использованием списков смежности (как описано в *разделе 5.7.1*) этот алгоритм имеет время исполнения $O(n + m)$.

На практике также возникают другие вопросы, касающиеся связности.

- *Является ли граф ориентированным?* Для ориентированных графов существует два понятия компонент связности. Ориентированный граф является *слабо связным* (weakly connected), если при игнорировании ориентации ребер он окажется связным. Таким образом, слабо связный граф состоит из одной-единственной компоненты. Ориентированный граф является *сильно связным* (strongly connected), если между любой парой его вершин существует ориентированный путь. Различие между ними легко понять при рассмотрении сети улиц с односторонним и двусторонним движением. Сеть является сильно связной, если возможно проехать между любыми двумя точками, не нарушая правил. Если между любыми двумя точками можно проехать, не нарушая *или* нарушая правила, то сеть является слабо связной. Если же попасть из точки A в точку B не является возможным, то сеть является несвязной.

  Слабо и сильно связные компоненты определяют однозначные разбиения вершин. На рис. 15.1 представлен ориентированный граф, состоящий из двух слабо или пяти сильно связных компонент (также называющихся *блоками* графа G).

  Проверку орграфа на слабую связность можно с легкостью провести за линейное время. Просто сделаем все ребра графа неориентированными и выполним на нем алгоритм поиска компонент связности на основе обхода в глубину. Проверку на сильную связность провести сложнее. Самый простой алгоритм с линейным временем исполнения осуществляет поиск, начиная с какой-либо вершины v, чтобы продемонстрировать достижимость всех вершин графа из этой вершины. Затем мы создаем граф G', поменяв направление всех ребер графа G на обратное. Чтобы узнать, достижимы ли все вершины графа G из вершины v, достаточно выполнить обход графа G' из этой вершины. Граф G является сильно связным тогда и только тогда, когда вершина v достижима из любой вершины в графе G и все вершины в графе G достижимы из вершины v.

  Все сильно связные компоненты графа G можно извлечь за линейное время, используя более сложные алгоритмы типа обхода в глубину. Обобщение только что описанной идеи двойного обхода в глубину с легкостью поддается программированию, но понять, как оно работает, несколько сложнее.

1. Выполняем обход в глубину, начав с произвольной вершины графа G и нумеруя вершины в порядке завершения их обработки (а не открытия).
2. Изменяем направление каждого ребра графа G, создав, таким образом, граф G'.
3. Выполняем обход в глубину графа G', начиная с вершины графа G, имеющей наибольший номер. Если не удается обойти граф G' полностью, продолжаем обход с еще не посещенной вершины с наибольшим номером.

4. Каждое созданное на шаге 3 дерево обхода в глубину является сильно связной компонентой.

   В *разделе 5.10.2* приводится моя реализация однопроходного алгоритма. В любом случае, проще использовать готовую реализацию, чем изучать описание алгоритма в учебнике.

- *Какое самое слабое место в моем графе/сети?* Прочность цепочки измеряется прочностью ее самого слабого звена. Потеря одного или нескольких внутренних звеньев разорвет цепочку на части. *Связность* графа измеряет его "прочность" — количество ребер или вершин, которое нужно удалить, чтобы разбить граф на части. Связность является важным понятием в проектировании сетей и других задачах структурирования.

  Алгоритмические задачи связности графов рассматриваются в *разделе 15.8*. В частности, двусвязные компоненты представляют собой части графа, получаемые в результате разрыва ребер, имеющих общую вершину. Все компоненты двусвязности можно найти за линейное время посредством обхода в глубину. Реализацию этого алгоритма см. в *разделе 5.9.2*. Вершины, удаление которых нарушает связность графа, принадлежат нескольким компонентам двусвязности, ребра которых однозначно распределяются между компонентами.

- *Является ли граф деревом? Как найти цикл, если он существует?* Задача поиска цикла возникает часто, особенно с ориентированными графами. Например, проверка, возможна ли взаимная блокировка последовательности условий, часто сводится к задаче выявления цикла. Если я жду Фреда, а Фред ждет Мэри, а Мэри ждет меня, то налицо цикл и взаимная блокировка.

  Для неориентированных графов аналогичной задачей является идентификация дерева. По определению дерево является неориентированным связным бесконтурным графом. Как описано ранее, обход в глубину можно использовать для проверки графа на связность. Если граф является связным и имеет $n - 1$ ребер для n вершин, то этот граф является деревом.

  Обход в глубину можно использовать для поиска циклов как в ориентированных, так и в неориентированных графах. Всякий раз, когда в процессе обхода в глубину мы обнаруживаем обратное ребро (т. е. ребро к вершине-предшественнику в дереве обхода в глубину), это обратное ребро и дерево совместно определяют ориентированный цикл. Существования другого такого цикла в ориентированном графе невозможно. Ориентированные графы, не содержащие контуров (циклов), называются бесконтурными. Фундаментальной операцией на бесконтурных орграфах является топологическая сортировка.

Реализации. Все реализации структур данных для работы с графами, описанные в *разделе 12.4*, включают реализацию обхода в ширину и обхода в глубину и, соответственно, определенную степень проверки на связность. Библиотека Boost Graph Library для C++ (см. [SLL02]) предоставляет реализации компонент связности. Библиотеку можно загрузить с веб-сайта **http://www.boost.org/libs/graph/doc**. Библиотека LEDA (см. *раздел 19.1.1*) содержит реализации на языке C++ этих компонент, а также двусвязных и трехсвязных компонент, и обходов в глубину и ширину.

Для программистов, пишущих на языке Java, библиотека JUNG (**http://jung.sourceforge.net/**) содержит реализации алгоритмов компонент двусвязности, а библиотека JGraphT (**http://jgrapht.sourceforge.net/**) — компоненты сильной двусвязности.

По моему предвзятому мнению, лучшей реализацией всех основных алгоритмов проверки на связность (включая компоненты сильной связности и компоненты двусвязности) на языке C является библиотека, разработанная для этой книги. Подробности см. в *разделе 19.1.10*.

Примечания

Обход в глубину был впервые использован в девятнадцатом столетии для поиска выхода из лабиринтов (см. [Luc91] и [Tar95]). А обход в ширину был впервые использован в 1957 г. для поиска кратчайшего пути (см. [Moo59]).

В работах [HT73b] и [Tar72] было установлено, что обход в глубину является фундаментальным методом построения эффективных алгоритмов на графах. Алгоритмы обхода в глубину и обхода в ширину приводятся в каждой книге, посвященной алгоритмам, причем книга [CLRS01] содержит, пожалуй, наиболее полное описание этих алгоритмов.

Первый алгоритм для поиска компонент сильной связности был предложен в работе [Tar72], а его описание приведено в книгах [BvG99], [Eve79a] и [Man89]. Еще один алгоритм для поиска компонент сильной связности, более легкий для программирования и более изящный, был разработан Шариром (Sharir) и Косараю (Kosaraju). Хорошие описания этого алгоритма приведены в книгах [AHU83] и [CLRS01]. В своей работе [CM96] Чериян (Cheriyan) и Мэлхорн (Mehlhorn) представляют улучшенные алгоритмы для некоторых задач на плотных графах, включая поиск компонент сильной связности.

Родственные задачи. Связность ребер и вершин (см. *раздел 15.8*), поиск кратчайшего пути (см. *раздел 15.4*).

15.2. Топологическая сортировка

Вход. Бесконтурный (ациклический) орграф $G = (V, E)$, также называющийся *частично упорядоченным множеством* (partially ordered set).

Задача. Найти линейное упорядочение вершин V, такое, что для каждого ребра (i, j) вершина i находится слева от вершины j (рис. 15.2).

Обсуждение. Подзадача топологической сортировки возникает в большинстве алгоритмов на ориентированных ациклических графах. Топологическая сортировка упорядочивает вершины и ребра бесконтурного орграфа простым и непротиворечивым образом, и поэтому играет ту же самую роль для бесконтурных орграфов, что и обход в глубину для общих графов.

Топологическую сортировку можно использовать для создания расписания работ с ограничивающими условиями очередности. Если у нас имеется набор задач, которые должны выполняться в определенной последовательности, то эти ограничивающие условия очередности формируют бесконтурный орграф, и любая топологическая сортировка на этом графе, иногда называемая *линейным расширением* (linear extension), определяет такой порядок выполнения этих работ, при котором каждая из них выполняется только после удовлетворения ее ограничивающих условий.

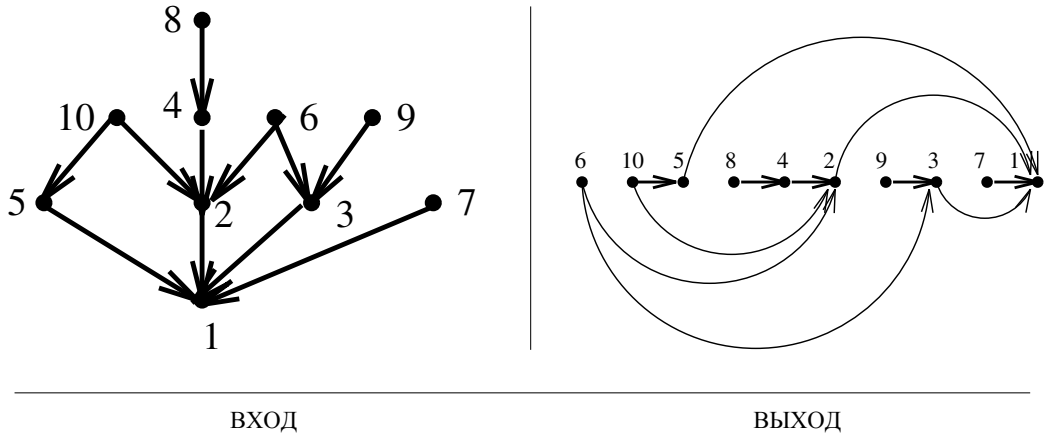


Рис. 15.2. Топологическая сортировка

В отношении топологической сортировки справедливы следующие три важных факта.

- Топологическую сортировку можно выполнять только на *бесконтурных* орграфах, т. к. любой ориентированный цикл принципиально противоречит линейному порядку выполнения работ.
- Топологическую сортировку можно выполнить на *любом* бесконтурном орграфе, поэтому для любого разумного набора ограничивающих условий очередности работ должно существовать, по крайней мере, одно расписание их выполнения.
- Для бесконтурного орграфа можно выполнить несколько разных топологических упорядочений, особенно при небольшом количестве ограничивающих условий. Так, для n работ, не имеющих никаких ограничивающих условий, любая из $n!$ перестановок работ является допустимым топологическим упорядочением.

Самый концептуально простой алгоритм топологической сортировки выполняет обход в глубину бесконтурного орграфа, чтобы найти полный набор вершин-истоков, т. е. вершин, у которых количество входящих дуг равно нулю. Любой бесконтурный орграф должен иметь, по крайней мере, одну такую вершину-исток. Вершины-истоки могут находиться в начале любого расписания, не нарушая никаких ограничивающих условий. Удаление всех исходящих ребер этих вершин-истоков создаст новые вершины-истоки, которые будут располагаться справа от первого набора. Эта процедура повторяется до тех пора, пока не будут учтены все вершины. При разумном выборе структур данных (списков смежности и очередей) этот алгоритм будет иметь время исполнения $O(n + m)$.

В альтернативном алгоритме используется то обстоятельство, что упорядочение всех вершин по времени завершения обхода в глубину в убывающем порядке дает линейное расширение. Реализация этого алгоритма с доказательством его правильности приводится в *разделе 5.10.1*.

В топологической сортировке следует принимать во внимание следующие два аспекта:

- *Как получить все линейные расширения?* В некоторых приложениях важно создать *все* линейные расширения бесконтурного орграфа. Вы должны отдавать себе отчет

в том, что количество линейных расширений может возрастать экспоненциально по отношению к размеру графа. Даже сама задача подсчета количества всех линейных расширений является NP-полной.

Алгоритмы для перечисления всех линейных расширений бесконтурного орграфа основаны на обходе с возвратом. В них создаются все возможные слева направо упорядочения, где кандидатами для следующей вершины являются все вершины с нулевой степенью захода. Прежде чем продолжать обход, из выбранной вершины удаляются все исходящие ребра. Оптимальный алгоритм для перечисления линейных расширений рассматривается в *подразделе "Примечания"*.

Алгоритмы для создания случайных линейных расширений начинают с произвольного линейного расширения. В этом линейном расширении последовательно выбираются пары вершин, которые меняются местами, если получающаяся в результате такого обмена перестановка остается топологической сортировкой. При выборе достаточного количества случайных пар вершин этот процесс дает случайное линейное расширение. Подробности см. в *подразделе "Примечания"*.

- *Как поступать, когда граф не является бесконтурным?* Когда набор ограничивающих условий содержит внутренние противоречия, естественно возникает задача поиска наименьшего набора условий, удаление которого устраняет все конфликты. Наборы непрестижных работ (вершин) или ограничений (ребер), после удаления которых орграф становится бесконтурным, называются *разрывающими множествами вершин* (feedback vertex set) и *разрывающими множествами дуг* (feedback arc set) соответственно. Эти структуры рассматриваются в *разделе 16.11*. К сожалению, задачи поиска обоих наборов являются NP-полными.

  Так как алгоритм топологической сортировки зацикливается, как только он находит вершину в ориентированном контуре, мы можем удалить проблемное ребро или вершину и продолжать работу. В конечном итоге этот простой и эффективный эвристический подход делает орграф бесконтурным, но может удалить больше элементов, чем необходимо. Лучший подход к решению этой задачи описывается в *разделе 9.10.3*.

Реализации. По сути, все упоминаемые в *разделе 12.4* реализации структур данных для представления графов, включая библиотеку Boost Graph Library (**http://www.boost.org/libs/graph/doc**) и библиотеку LEDA (см. *раздел 19.1.1*), содержат реализации топологической сортировки. Для языка Java библиотека JDSL (**http://www.jdsl.org/**) содержит реализацию специальной процедуры для выполнения взвешенной топологической нумерации. Советую также обратить внимание на библиотеку JGraphT (**http://jgrapht.sourceforge.net**).

Сервер комбинаторных объектов на веб-сайте **http://theory.cs.uvic.ca** предоставляет программы на языке C для генерирования линейных расширений как в лексикографическом порядке, так и в порядке кода Грея, а также средства для их перечисления. Сервер оснащен интерактивным интерфейсом.

По моему предвзятому мнению, лучшей реализацией всех основных алгоритмов решения задач на графах (включая топологическую сортировку), на языке C, является библиотека, разработанная для этой книги. Подробности см. в *разделе 19.1.10*.

Примечания

Хорошие описания топологической сортировки включают книги [CLRS01] и [Man89]. В своей работе [BW91] Брайтвэл (Brightwell) и Уинклер (Winkler) доказали, что задача подсчета количества линейных расширений частичного порядка является #P-полной. Класс сложности #P включает класс сложности NP, следовательно, любая #P-полная задача является, по крайней мере, NP-полной.

Прусс (Pruesse) и Раски (Ruskey) в своей работе ([PR86]) представляют алгоритм для генерирования линейных расширений бесконтурного орграфа за постоянное амортизированное время. Кроме этого, каждое расширение отличается от своего предшественника перестановкой только одного или двух смежных элементов. Этот алгоритм можно использовать для подсчета линейных расширений $e(G)$ n-вершинного графа G за время $O(n^2 + e(G))$. Альтернативным методом перечисления линейных расширений является метод поиска в обратном направлении Ависа (Avis) и Фукуды (Fukuda), представленный в работе [AF96]. Программа обхода с возвратом для генерирования всех линейных перечислений описана в работе [KS74] Дональда Кнута.

В работе [Hub06] Губера (Huber) представлен алгоритм случайной выборки с равномерным распределением линейных расширений из произвольного частичного порядка за время $O(n^3 \lg n)$, что улучшает результаты работы алгоритма, представленного в работе [BD99].

Родственные задачи. Сортировка (см. *раздел 14.1*), разрывающее множество вершин и ребер (см. *раздел 16.11*).

15.3. Минимальные остовные деревья

Вход. Граф $G = (V, E)$ со взвешенными ребрами.

Задача. Найти подмножество ребер $E' \in E$, которое формирует дерево на вершинах V и имеет минимальный вес (рис. 15.3).

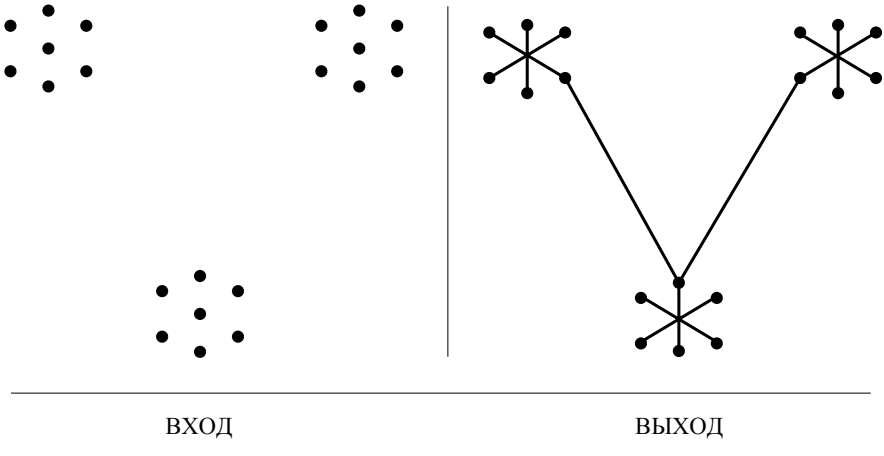


Рис. 15.3. Поиск минимального остовного дерева

Обсуждение. Минимальное остовное дерево графа определяет подмножество ребер с минимальным весом, которое удерживает граф в одной компоненте связности. В решении задачи построения минимального остовного дерева могут быть заинтересованы,

например, телефонные компании, поскольку минимальное остовное дерево набора объектов определяет схему для соединения этих объектов в сеть с использованием кабеля наименьшей длины. Задача построения минимального остовного дерева является основной задачей проектирования сетей.

Важность минимальных остовных деревьев объясняется несколькими причинами:

- они легко и быстро реализуются на компьютере и образуют разреженные подграфы, которые содержат много информации о первоначальных графах;
- они предоставляют способ определить кластеры в наборах точек. Удаление длинных ребер из минимального остовного дерева оставляет компоненты связности, которые определяют естественные кластеры в наборе данных, как показано на рис. 15.3;
- с их помощью можно получить приблизительные решения сложных задач, таких как задача построения дерева Штейнера и задача коммивояжера;
- они могут быть использованы в учебных целях, поскольку алгоритмы для построения минимального остовного дерева дают графическую иллюстрацию того, как "жадные" алгоритмы выдают доказуемо оптимальные решения.

Известны три классических алгоритма для эффективного создания минимальных остовных деревьев. Подробная реализация двух из них (алгоритма Прима и алгоритма Крускала) совместно с доказательством их правильности дается в *разделе 6.1*. Третий алгоритм менее известен, хотя он был изобретен первым, прост в реализации и более эффективен.

Далее даются краткие описания и псевдокод этих алгоритмов.

- *Алгоритм Крускала.* В начале каждая вершина представляет отдельное дерево, и эти деревья сливаются в одно путем последовательного добавления ребер с наименьшим весом, которые соединяют по два поддерева (т. е. не создают цикл). Псевдокод этого алгоритма представлен в листинге 15.1.

Листинг 15.1. Алгоритм Крускала

```
Kruskal(G)
   Сортируем ребра по весу в возрастающем порядке
   count = 0
   while (count < n − 1) do
      get next edge (v, w)
      if (component (v) ≠ component(w))
         add to T
         component(v) = component(w)
```

Проверку, какой компонент выбирать, можно эффективно реализовать с помощью структуры данных "объединение-поиск" (см. *раздел 12.5*), что даст нам алгоритм с временем исполнения $O(m\lg m)$.

- *Алгоритм Прима.* Начинает работу с произвольной вершины v и "выращивает" дерево из этой вершины, в цикле выбирая ребро с наименьшим весом, которое присоединяет к дереву какую-либо новую вершину. В процессе исполнения каждая вершина помечается или как входящая в дерево, или как открытая, но еще не до-

бавленная в дерево вершина, или как не увиденная, (т. е., расположенная на расстоянии более одного ребра от дерева). Псевдокод этого алгоритма представлен в листинге 15.2.

Листинг 15.2. Алгоритм Прима

```
Prim(G)
   Выбираем произвольную вершину, с которой надо начинать построение дерева
   while (имеются вершины, не включенные в дерево)
      выбираем ребро минимального веса между деревом и
                                        вершиной вне дерева
      добавляем выбранное ребро и вершину в дерево
      обновляем стоимость для всех затрагиваемых ребер вне дерева
```

Этот алгоритм создает остовное дерево для любого связного графа, т. к. циклы не могут быть внесены через ребра между вершинами в дереве и вне его. То, что это дерево действительно имеет минимальный вес, можно доказать методом "от противного". Для простых структур данных можно реализовать алгоритм Прима с временем исполнения $O(n^2)$.

- *Алгоритм Борувки.* Основан на том обстоятельстве, что инцидентные каждой вершине ребра наименьшего веса должны быть в минимальном остовном дереве. Результатом объединения этих ребер будет остовной лес, содержащий, как минимум, $n/2$ деревьев. Теперь для каждого из этих деревьев T выбираем такое ребро (x, y) наименьшего веса, для которого $x \in T$ и $y \notin T$. Каждое из этих ребер опять должно быть в минимальном остовном дереве, а результатом их объединения опять будет остовной лес, содержащий, самое большее, половину предыдущего количества деревьев. Псевдокод этого алгоритма представлен в листинге 15.3.

Листинг 15.3. Алгоритм Борувки

```
Boruvka(G)
   Инициализируем остовной лес F, состоящий из n одновершинных деревьев
      while (лес F содержит больше, чем одно дерево)
          for each (дерева T в лесу F) находим ребро с наименьшим весом
                                                  от T к G – T
          добавляем все выбранные ребра в лес F,
          сливая вместе пары деревьев
```

В каждой итерации количество деревьев уменьшается, по крайней мере, вдвое, что дает нам минимальное остовное дерево после, самое большее, $\log n$ итераций, каждая из которых выполняется за линейное время. В целом это дает нам алгоритм с временем исполнения $O(m\log n)$, причем без использования каких-либо хитроумных структур данных.

Построение минимального остовного дерева является только одной из нескольких задач об остовном дереве, возникающих на практике. Чтобы разобраться в них, попытайтесь найти ответы на следующие вопросы.

- *Одинаковый ли вес у всех ребер графа?* Любое остовное дерево из n вершин имеет ровно $n - 1$ ребер. Таким образом, если граф невзвешенный, то *любое* остовное де-

рево будет минимальным остовным деревом. Корневое остовное дерево можно найти за линейное время посредством обхода в глубину или в ширину. Обход в глубину обычно выдает высокие и тонкие деревья, а обход в ширину отражает структуру расстояний графа (см. *главу 5*).

- *Какой алгоритм использовать, Прима или Крускала?* В том виде, в каком они реализованы в *разделе 6.1*, алгоритм Прима имеет время исполнения $O(n^2)$, а алгоритм Крускала — $O(m\log m)$. Таким образом, алгоритм Прима эффективен на плотных графах, а алгоритм Крускала — на разреженных.

  Вместе с тем, используя более сложные структуры данных, можно создать реализации алгоритма Прима с временем исполнения $O(m + n\lg n)$, а реализация алгоритма Прима с использованием парных пирамид будет самой быстрой на практике как для разреженных, так и для плотных графов.

- *Что предпринять, если входные данные — не граф, а точки на плоскости?* Геометрические экземпляры, состоящие из n точек в d измерениях, можно решать, создав граф полного расстояния за время $O(n^2)$, а затем построив минимальное остовное дерево этого графа. Но для точек на плоскости более эффективный подход — непосредственно решить геометрическую версию задачи. Чтобы построить минимальное остовное дерево для n точек, сначала создадим триангуляцию Делоне (Delaunay triangulation) для этих точек (см. *разделы 17.3* и *17.4*). На плоскости это даст граф с $O(n)$ ребрами, который содержит все ребра минимального остовного дерева нашего множества точек. Алгоритм Крускала находит минимальное остовное дерево в этом разреженном графе за время $O(n\lg n)$.

- *Как найти остовное дерево, не имеющее вершин с высокой степенью?* Другой распространенной целью задач об остовном дереве является минимизация степени вершин, обычно чтобы минимизировать исходящие разветвления в сети. К сожалению, задача построения остовного дерева с максимальной степенью вершин, равной 2, является NP-полной задачей, т. к. она идентична задаче поиска гамильтонова пути. Но существуют эффективные алгоритмы для создания остовных деревьев с максимальной степенью вершин на единицу больше, чем требуемая, что, скорее всего, будет достаточно на практике.

Реализации. Среди реализаций структур данных для представления графов из *раздела 12.4* обязательно должны присутствовать реализации алгоритма Прима и/или алгоритма Крускала. Этому требованию отвечают библиотеки Boost Graph Library (см. [SLL02]) (**http://www.boost.org/libs/graph/doc**) и LEDA (см. *раздел 19.1.1*). По неизвестной причине библиотеки графов Java, ориентированные преимущественно на социальные сети, не содержат реализаций этих алгоритмов, зато в библиотеке JDSL (**http://www.jdsl.org**) есть реализации обоих алгоритмов.

Эксперименты с измерением времени исполнения алгоритмов для построения минимального остовного дерева дают противоречивые результаты, дающие основание полагать, что разница в их производительности слишком незначительная. Реализации алгоритмов Прима, Крускала и Черитона-Тарьяна (Cheriton-Tarjan) на языке Pascal описываются в книге [MS91]. Здесь же приводится подробный анализ экспериментальных данных, доказывающий, что на большинстве графов самым быстрым будет алгоритм Прима, реализованный с правильно построенной очередью с приоритетами. В про-

грамме Standford GraphBase самым быстрым из четырех реализаций разных алгоритмов построения минимального остовного дерева оказался алгоритм Крускала (см. *раздел 19.1.8*).

Библиотека Combinatorica (см. [PS03]) содержит реализацию (на языке пакета Mathematica) алгоритма Крускала для построения минимального остовного дерева и быстрого подсчета количества остовных деревьев в графе. Подробности см. в *разделе 19.1.9*.

По моему предвзятому мнению, лучшей реализацией на языке C всех основных алгоритмов на графах, включая алгоритмы для построения минимальных остовных деревьев, является библиотека, разработанная для этой книги. Подробности см. в *разделе 19.1.10*.

Примечания

Задача построения минимального остовного дерева была впервые решена в 1926 г., когда был разработан алгоритм Борувки. Алгоритмы Прима (см. [Pri57]) и Крускала для решения этой задачи появились только в середине 1950-х годов. Алгоритм Прима был заново открыт Дейкстрой (см. [Dij59]). Дополнительную информацию об истории алгоритмов построения минимального остовного дерева можно найти в [GH85]. Ву (Wu) и Чао (Chao) написали монографию [WC04b] о задаче построения минимального остовного дерева и родственным задачам.

Самые быстрые реализации алгоритмов Прима и Крускала используют пирамиды Фибоначчи (см. [FT87]). Однако позже были предложены парные пирамиды, которые обеспечивают такую же производительность, но при меньших накладных расходах. Эксперименты с парными пирамидами описаны в журнале [SV87].

Алгоритм, получаемый в результате простой комбинации алгоритма Борувки с алгоритмом Прима, имеет время исполнения равное $O(m\lg\lg n)$. Выполните $\lg\lg n$ итераций алгоритма Борувки, чтобы получить лес из, самое большее, $n/\lg n$ деревьев. Теперь создайте граф G', в котором каждое дерево этого леса представлено одной вершиной, а вес ребра между деревьями T_i и T_j установите равным весу самого легкого ребра (x, y), где $x \in T_i$, а $y \in T_j$. Минимальное остовное дерево графа G' совместно с ребрами, выбранными алгоритмом Борувки, образуют минимальное остовное дерево графа G. Время исполнения алгоритм Прима (реализованного с использованием пирамид Фибоначчи) на этом графе из $n/\lg n$ вершин и m ребер составляет $O(n + m)$.

История выяснения оптимального времени построения минимальных остовных деревьев выглядит примерно так. В своей работе [KKT95] Каргер (Karger), Кляйн (Klein) и Тарьян (Tarjan) предложили рандомизированный алгоритм на основе алгоритма Борувки для построения минимального остовного дерева за линейное время. В свою очередь, Шазель (Chazelle) в работе [Cha00] представил алгоритм с временем исполнения $O(m\alpha(m, n))$, где $\alpha(m, n)$ является функцией, обратной функции Аккермана. Наконец, Петти (Pettie) и Рамачандран (Ramachandran) в работе [PR02] описали доказуемо оптимальный алгоритм, точное время исполнения которого неизвестно (как бы странно это ни звучало), но лежит в диапазоне между $\Omega(n + m)$ и $O(m\alpha(m, n))$.

Остовным подграфом $S(G)$ графа G называется подграф, обеспечивающий эффективный компромисс между двумя противоречащими друг другу целями проектирования сетей. Говоря конкретно, общий вес *остовного подграфа* близок к весу минимального остовного дерева полного графа G, и в то же самое время гарантируется, что кратчайший путь между вершинами x и y в графе $S(G)$ близок к кратчайшему пути в полном графе G. Самый свежий исчерпывающий обзор достижений в этой области представлен в монографии [NS07].

Алгоритм для вычисления евклидова минимального остовного дерева был разработан Шамосом (Shamos). Он обсуждается в учебниках по вычислительной геометрии, таких как [dBvKOS00] и [PS85].

В работе [FR94] Фурера (Furer) и Рагхавачари (Raghavachari) представлен алгоритм для создания остовного дерева почти минимальной степени — всего лишь на единицу больше, чем степень остовного дерева наименьшей степени. Данная ситуация аналогична теореме Визинга (Vizing) для раскраски ребер, которая позволяет построить аппроксимирующий алгоритм, выдающий решение с точностью до единицы. В работе [SL07] представлен полиномиальный алгоритм построения остовного дерева с максимальной степенью, равной или меньшей, чем $k + 1$, стоимость которого не больше стоимости оптимального минимального остовного дерева, с максимальной степенью, равной или меньшей, чем k.

Алгоритмы построения минимального остовного дерева можно рассматривать в виде матроидов, которые представляют собой системы подмножеств, замкнутые по включению. Максимальное взвешенное независимое множество матроида можно найти, используя "жадный" алгоритм. Связь между "жадными" алгоритмами и матроидами была установлена Эдмондсом (Edmonds) в его работе [Edm71]. Теория матроидов обсуждается в [Law76] и [PS98].

Динамические алгоритмы на графах стремятся сохранять инвариант (например, минимальное остовное дерево) при операциях вставки или удаления ребер. В работе [HdlT01] представлен эффективный детерминистический алгоритм для поддержания минимальных остовных деревьев (и нескольких других инвариантов) за амортизированное полилогарифмическое время при каждом обновлении.

Алгоритмы генерирования остовных деревьев в порядке возрастания веса обсуждаются в работе [Gab77]. Полный набор остовных деревьев невзвешенного графа можно сгенерировать за постоянное амортизированное время. Обзор алгоритмов генерирования, ранжирования и остовных деревьев представлено в работе [Rus03].

Родственные задачи. Дерево Штейнера (см. *раздел 16.10*), задача коммивояжера (см. *раздел 16.4*).

15.4. Поиск кратчайшего пути

Вход. Граф G со взвешенными ребрами и две вершины, s и t.

Задача. Найти кратчайший путь от вершины s к вершине t (рис. 15.4).

Обсуждение. Задача поиска кратчайшего пути в графе имеет несколько применений, иногда довольно неожиданных:

- *Транспортировка или связь*. Задача состоит в том, чтобы найти наилучший маршрут, например, для отправки грузовика с товаром из Чикаго в Феникс или маршрут для направления сетевых пакетов к месту назначения.
- *Сегментация изображения* — разбиение оцифрованного изображения на области, содержащие отдельные объекты. Задача состоит в том, чтобы провести линии, разграничивающие эти области. При одном из возможных решений такие линии соединяют точки x и y отрезками, не проходящими через пикселы объекта, насколько это возможно. Сетку пикселов можно смоделировать в виде графа, стоимость ребер которого отражает изменение цвета между соседними пикселами. Кратчайший путь

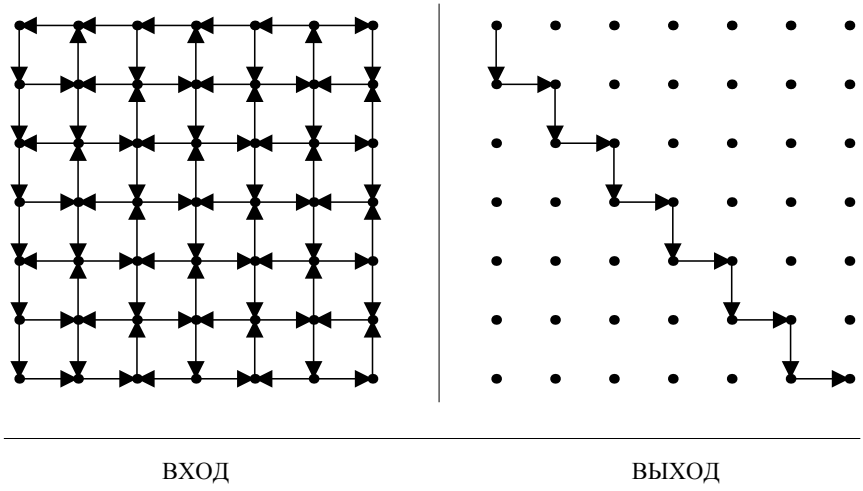


Рис. 15.4. Поиск кратчайшего пути

от точки x к точке y в таком взвешенном графе определяет наилучшую линию раздела.

- *Различение омофонов.* Омофонами называются слова, которые произносятся одинаково, но пишутся по-разному и имеют разные значения. Различение омофонов представляет собой одну из основных задач в области распознавания речи.

  Ключом к решению этой задачи является внедрение грамматических ограничивающих условий в процесс интерпретации слов в предложении. Для этого каждая последовательность фонем (опознанных звуков) сопоставляется со словами, которые, вероятно, им соответствуют. Потом создается граф, в котором вершины представляют эти возможные трактовки слова, а смежные трактовки соединяются ребрами. Если для каждого ребра установить вес, пропорциональный вероятности перехода, то кратчайший путь будет определять наилучшую трактовку предложения. Подробное описание подобного приложения изложено в *разделе 6.4*.

- *Визуальное представление графа.* На рисунке "центр" графа должен находиться в центре страницы. Хорошим определением центра графа будет вершина с минимальным расстоянием до всех других вершин графа. Чтобы найти эту центральную вершину, нужно знать расстояние (т. е. кратчайший путь) между всеми парами вершин.

Основным алгоритмом поиска кратчайшего пути является *алгоритм Дейкстры*, который эффективно вычисляет кратчайший путь от данной начальной вершины x ко всем $n-1$ другим вершинам. При каждой итерации алгоритм находит новую вершину v, для которой известен кратчайший путь из вершины x. Мы сопровождаем набор вершин S, к которым в настоящее время имеется кратчайший путь из вершины v, и с каждой итерацией этот набор увеличивается на одну вершину. При каждой итерации алгоритм находит ребро (u, v), где $u, u' \in S$ и $v, v' \in V - S$, такие, что:

$$dist(x,u) + weight(u,v) = \min_{(u',v') \in E} dist(x,u') + weight(u',v')$$

где *dist* — расстояние, *weight* — вес, min *dist* — минимальное расстояние.

Это ребро (u, v) добавляется к дереву кратчайшего пути с корнем в вершине x, которое описывает все кратчайшие пути из этой вершины.

В *разделе 6.3.1* приводится реализация алгоритма Дейкстры с временем исполнения $O(n^2)$. Как показано ниже, можно получить лучшее время исполнения, используя более сложные структуры данных. Если нам достаточно найти кратчайший путь от x к y, то выполнение алгоритма следует прервать, как только вершина y окажется в множестве S.

Алгоритм Дейкстры применяется для поиска кратчайшего пути из одной начальной точки на графах с положительным весом ребер. Но существуют ситуации, когда нужно воспользоваться другим алгоритмом.

- *Является ли граф взвешенным?* Если граф невзвешенный, то простой обход в ширину, начиная с вершины-источника, предоставит кратчайший путь ко всем другим вершинам за линейное время. Более сложный алгоритм требуется только тогда, когда ребра имеют разный вес. Обход в ширину проще и быстрее, чем алгоритм Дейкстры.
- *Есть ли в графе ребра с отрицательным весом?* Алгоритм Дейкстры предполагает, что все ребра графа имеют положительный вес. Для графов, в которых есть ребра с отрицательным весом, требуется использовать более общий, но менее производительный алгоритм Беллмана-Форда (Bellman-Ford algorithm). Графы, содержащие циклы с отрицательным весом, представляют еще большую проблему. Обратите внимание, что в таком графе кратчайший путь из x в y не определен, т. к. имеется возможность пойти из вершины x по циклу с отрицательным весом и кружить по нему, делая общую стоимость сколь угодно низкой.

  Заметим, что добавление фиксированного значения к весу каждого ребра с целью сделать веса всех ребер положительными не решает проблему. В таком случае алгоритм Дейкстры будет выбирать пути с наименьшим количеством ребер, даже если эти пути не были кратчайшими взвешенными путями в первоначальном графе.
- *Как поступить, если входные данные представлены не графом, а набором геометрических объектов?* Во многих приложениях требуется найти кратчайший путь между двумя точками при наличии геометрических объектов, например, в комнате, обставленной мебелью. Самое простое решение в таком случае заключается в преобразовании входного экземпляра задачи в граф расстояний для последующей его обработки алгоритмом Дейкстры. Вершины будут соответствовать узлам на границах препятствий, а ребра будут определены только между такими парами вершин, что из одной вершины видна другая.

  Существуют и более эффективные геометрические алгоритмы, вычисляющие кратчайший путь непосредственно по расположению препятствий. Информацию о таких геометрических алгоритмах см. в *разделе 17.14* и в *подразделе "Примечания"*.
- *Является ли граф бесконтурным орграфом?* В бесконтурных орграфах кратчайший путь можно найти за линейное время. Сначала выполняем топологическую сортировку, упорядочивая вершины таким образом, чтобы все они расположились слева направо, начиная с исходной вершины s. Очевидно, что расстояние от вершины s до

самой себя, $d(s, s)$, равно 0. Остальные вершины обрабатываются слева направо. Обратите внимание, что

$$d(s,j) = \min_{(x,i)\in E} d(s,i) + w(i,j)$$

т. к. мы уже знаем кратчайший путь $d(s, i)$ для всех вершин слева от вершины i. В самом деле, большинство задач динамического программирования можно сформулировать в виде задачи поиска кратчайшего пути в бесконтурном орграфе. Обратите внимание, что, заменив min на max, мы сможем применить тот же самый алгоритм для поиска самого длинного пути в бесконтурном орграфе. Эту особенность можно использовать во многих приложениях, например, при составлении расписаний (см. *раздел 14.9*).

- *Требуется ли найти кратчайший путь между всеми парами точек?* Если требуется найти кратчайший путь между всеми парами вершин, то одно из решений — выполнить алгоритм Дейкстры n раз, по одному разу для каждой начальной вершины. Также можно использовать алгоритм Флойда-Варшалла, имеющий время исполнения $O(n^3)$. Этот алгоритм работает быстрее, чем алгоритм Дейкстры, и его легче программировать. Он также работает с отрицательным весом ребер, но не с циклами. Описание алгоритма вместе с реализацией представлено в *разделе 6.3.2*, а в листинге 15.4 приводится его псевдокод. Переменная M обозначает матрицу расстояний, в которой $M_{ij} = \infty$, если ребро (i, j) не существует.

Листинг 15.4. Алгоритм Флойда-Варшалла

```
D^0 = M
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            D_ij^k = min(D_ij^(k-1), D_ik^(k-1) + D_kj^(k-1))
Return D^n
```

Ключевым обстоятельством к пониманию алгоритма Флойда-Варшалла является то, что D_{ij}^k обозначает "длину кратчайшего пути от вершины i к вершине j, который проходит через вершины 1, ..., k, как возможные промежуточные вершины." Обратите внимание, что сложность по памяти составляет всего лишь $O(n^2)$, т. к. на этапе k нам требуется знать только расстояния D^k и D^{k-1}.

- *Как найти кратчайший цикл в графе?* Одним из применений алгоритма поиска кратчайшего пути между всеми парами вершин является поиск кратчайшего цикла графа, называющегося обхватом (girth) (рис. 15.5).

  Алгоритм Флойда можно использовать для вычисления расстояния d_{ii}, где $1 \le i \le n$, что является кратчайшим путем из вершины i в вершину i, или, иными словами, самым коротким циклом через вершину i.

  Возможно, это вам и требуется. Кратчайший цикл через вершину x вероятно может проходить от вершины x к вершине y и обратно к вершине x, используя дважды одно и то же ребро. Цикл называется *простым*, если каждое его ребро и вершина по-

сещаются только один раз. Очевидным подходом к поиску кратчайшего простого цикла является вычисление кратчайших путей от вершины i ко всем другим вершинам с последующей проверкой наличия приемлемого ребра от каждой вершины до вершины i.

Задача поиска самого длинного цикла графа включает задачу поиска гамильтонова цикла как частный случай (см. *раздел 16.5*), так что она является NP-полной.

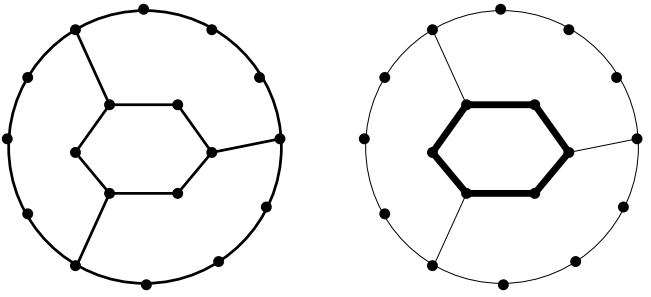


Рис. 15.5. Обхват, или кратчайший цикл, графа

Матрицу кратчайшего пути для всех пар вершин можно использовать для вычисления нескольких полезных инвариантов, связанных с центром графа G. *Эксцентриситетом* вершины v графа называется кратчайший путь к самой дальней вершине от v. На этом понятии основаны некоторые другие инварианты. *Радиусом* графа называется минимальный из эксцентриситетов вершин графа, а вершина, на которой достигается этот минимум, — *центральной вершиной*. *Диаметром* называется наибольшее расстояние между вершинами графа.

Реализации. Самые эффективные программы поиска кратчайшего пути написаны Эндрю Голдбергом (Andrew Goldberg) и его соавторами. Загрузить эти программы можно с веб-сайта **http://www.avglab.com/andrew/soft.html**.

В частности, программа MLB на языке C++ предназначена для поиска кратчайшего пути в графах с ребрами положительного веса, выраженного целыми числами. Подробности об этом алгоритме и его реализации см. в [Gol01]. Время исполнения этого алгоритма обычно только в 4–5 раз больше, чем время обхода в ширину, и он может обрабатывать графы, содержащие миллионы вершин. Также существуют высокопроизводительные реализации на языке C как алгоритма Дейкстры, так и алгоритма Беллмана-Форда (см. [CGR99]).

Все библиотеки графов на C++ и Java, упомянутые в *разделе 12.4*, содержат, как минимум, реализацию алгоритма Дейкстры. Библиотека Boost Graph Library на языке C++ (см. [SLL02]) содержит обширную коллекцию алгоритмов поиска кратчайшего пути, включая алгоритмы Беллмана-Форда и Джонсона. Загрузить библиотеку можно на вебсайте **http://www.boost.org/libs/graph/doc**. Библиотека LEDA (см. *раздел 19.1.1*) содержит хорошие реализации на языке C++ всех рассмотренных здесь алгоритмов поиска кратчайшего пути, включая алгоритмы Дейкстры, Беллмана-Форда и Флойда-Варшалла. Библиотека JGraphT (**http://jgrapht.sourceforge.net**) содержит реализации на языке Java как алгоритма Дейкстры, так и алгоритма Беллмана-Форда. Библиотека

алгоритмов для этой книги содержит реализации на языке C алгоритмов Дейкстры и Флойда-Варшалла. Подробности см. в *разделе 19.1.10*.

Соревнования DIMACS в октябре 2006 г. проводились по алгоритмам поиска кратчайшего пути. Во время соревнований обсуждались реализации эффективных алгоритмов для различных аспектов поиска кратчайшего пути. Соответствующие материалы можно найти на веб-сайте **http://dimacs.rutgers.edu/Challenges/**.

Примечания

Одним из удачных описаний алгоритмов Дейкстры (см. [Dij59]), Беллмана-Форда (см. [Bel58] и [FF62]) и Флойда-Варшалла для поиска кратчайшего пути для всех пар вершин (см. [Flo62]) является книга [CLRS01]. Относительно свежий обзор алгоритмов поиска кратчайшего пути представлен в работе [Zwi01], а обзор геометрических алгоритмов кратчайшего пути — в книге [PN04].

Самый быстрый известный алгоритм — с временем исполнения $O(m + n\log n)$ — для поиска кратчайшего пути из одной вершины в графах с ребрами с положительным весом является алгоритм Дейкстры, использующий пирамиды Фибоначчи. Экспериментальные исследования алгоритмов поиска кратчайшего пути описаны в работах [DF79] и [DGKK79]. Но эти эксперименты проводились до того, как были разработаны пирамиды Фибоначчи. Сравнительно недавнее исследование можно найти в работе [CGR99]. На практике производительность алгоритма Дейкстры можно повысить с помощью эвристических методов. В работе [HSWW05] предоставляется экспериментальное исследование взаимодействия четырех таких эвристических методов.

Приблизительный кратчайший путь между двумя точками разветвленных дорожных сетей нетрудно найти с помощью таких служб, как Mapquest. Решение этой задачи несколько отличается от решения обсуждаемых здесь задач поиска кратчайшего пути. Во-первых, затраты на предварительную обработку можно амортизировать, распределив их среди многих запросов поиска пути от одной точки до другой. Во-вторых, высокоскоростные магистрали большой протяженности могут свести задачу поиска кратчайшего пути к поиску наилучших точек для въезда на магистраль и съезда с нее. И, наконец, для практических целей достаточно получить приблизительные или эвристические результаты.

Алгоритм A* выполняет поиск кратчайшего пути, выбирая первый оптимальный вариант, и сопровождает его анализом нижнего предела, чтобы установить, действительно ли найденный путь является кратчайшим путем графа.

В работах [GKW06] и [GKW07] описана реализация алгоритма A*, способного после двух часов предварительной обработки выполнять за одну миллисекунду запросы о кратчайшем пути между двумя точками дорожной сети национального масштаба.

Во многих приложениях, кроме оптимального пути, требуется найти альтернативные короткие пути. Отсюда возникает задача поиска k кратчайших путей. Существуют варианты этой задачи, в зависимости от того, должен ли требуемый путь быть простым или он может содержать циклы. Приведенная в работе [Epp98] реализация генерирует представление этих путей за время $O(m + n\log n + k)$. Из этого представления отдельные пути можно восстановить за время $O(n)$. Новый алгоритм и экспериментальные результаты см. в работе [HMS03].

Существуют быстрые алгоритмы для вычисления обхвата как общих, так и планарных графов (см. [IR78] и [Dji00] соответственно).

Родственные задачи. Потоки в сетях (см. *раздел 15.9*), планирование перемещений (см. *раздел 17.14*).

15.5. Транзитивное замыкание и транзитивная редукция

Вход. Ориентированный граф $G = (V, E)$.

Задача. Для получения *транзитивного замыкания* создать граф $G' = (V, E')$, у которого $(i, j) \in E'$ тогда и только тогда, когда в графе G существует ориентированный путь от i до j. Для получения *транзитивной редукции* создать небольшой граф $G' = (V, E')$, который содержит ориентированный путь от i до j тогда и только тогда, когда ориентированный путь от i до j существует в графе G (рис. 15.6).

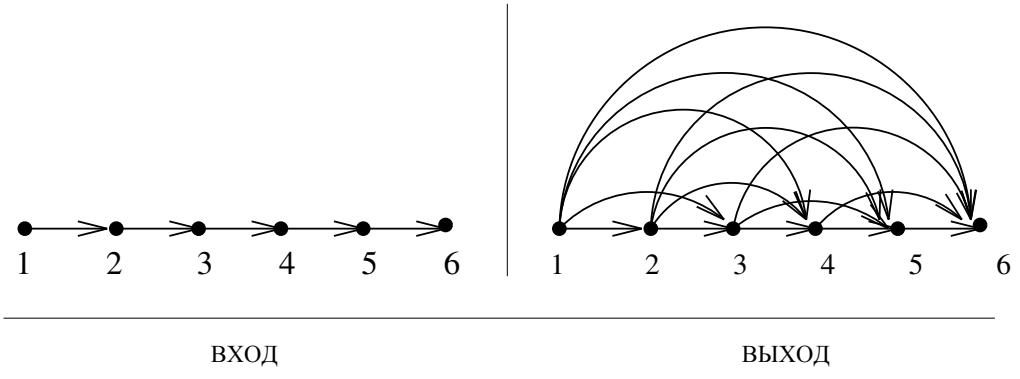


Рис. 15.6. Транзитивное замыкание

Обсуждение. Транзитивное замыкание можно рассматривать как создание структуры данных, позволяющей эффективно решать задачу достижимости (т. е., можно ли попасть в пункт x из пункта y?). После создания транзитивного замыкания на любой вопрос относительно достижимости можно найти ответ за постоянное время, просто возвращая значение соответствующего элемента матрицы.

Транзитивное замыкание лежит в основе распространения изменений атрибутов графа G. Рассмотрим, например, граф, моделирующий электронную таблицу, у которого вершины представляют ячейки таблицы, а ребро (i, j) соединяет соответствующие ячейки, если результат ячейки j зависит от ячейки i. Когда изменяется значение данной ячейки, то также требуется обновить значения всех зависящих от нее (т. е. достижимых из нее) ячеек. Идентификация этих ячеек осуществляется с помощью транзитивного замыкания графа G. Аналогичным образом, многие задачи баз данных сводятся к построению транзитивных замыканий.

Для построения транзитивного замыкания применяются три основных алгоритма:

- самый простой алгоритм выполняет обход графа в ширину или глубину из одной из вершин и запоминает все посещенные вершины. Время работы алгоритма, выполняющего n таких обходов, равно $O(n(n + m))$ и ухудшается до кубического для плотных графов. Этот алгоритм легко поддается реализации, имеет хорошую производительность на разреженных графах и, скорее всего, подходит для вашего приложения;

- алгоритм Варшалла создает транзитивные замыкания за время $O(n^3)$, используя простой, изящный алгоритм, идентичный алгоритму Флойда для поиска кратчайшего пути между всеми парами, рассматриваемый в *разделе 15.4*. Если нас интересует только само транзитивное замыкание, но не длина получившихся путей, то можно уменьшить объем требуемой памяти, используя только один бит для каждого элемента матрицы. Таким образом, $D_{ij}^k = 1$ тогда и только тогда, когда вершина j достижима из вершины i при посещении в качестве промежуточных только вершин 1, ..., k;
- вычислить транзитивное замыкание можно также с помощью перемножения матриц. Пусть M^1 — матрица смежности графа G. Ненулевые значения матрицы $M^2 = M \times M$ идентифицируют все пути длиной 2 графа G. Обратите внимание, что $M^2[i,j] = \sum_x M[i,x] \cdot M[x,j]$, поэтому путь (i,x,j) содержится в $M^2[i,j]$. Таким образом, объединение $\cup_i^n M^i$ выдает транзитивное замыкание T. Кроме этого, это объединение можно вычислить всего лишь за $O(\lg n)$ матричных операций, используя быстрый алгоритм возведения в степень, рассмотренный в *разделе 13.9*.

  Предположительно, для больших значений n можно получить лучшую производительность, используя алгоритм Штрассена для быстрого перемножения матриц, но лично я бы не стал тратить на это свое время. Так как сложность задачи построения транзитивного замыкания такая же, как и у задачи перемножения матриц, надежда на получение значительно более быстрого алгоритма невелика.

Для многих графов время исполнения этих трех процедур можно значительно улучшить. Вспомните, что компонентой сильной связности является набор вершин, в котором любая вершина достижима из любой другой вершины. Например, любой цикл определяет сильно связный подграф. Из всех вершин в любой компоненте сильной связности должно быть достижимо одно и то же подмножество графа G. Таким образом, мы можем выполнить сведение нашей задачи, создав транзитивное замыкание на графе компонент сильной связности, который должен иметь значительно меньшее количество ребер и вершин, чем граф G. Компоненты сильной связности графа G можно найти за линейное время (см. *раздел 15.1*).

Транзитивная редукция (также называемая *минимальным эквивалентным ориентированным графом*) является операцией, обратной операции транзитивного замыкания, а именно уменьшением количества ребер при сохранении первоначальных свойств достижимости. Транзитивное замыкание графа G идентично транзитивному замыканию транзитивной редукции этого графа. Основным применением транзитивной редукции является минимизация пространства, удаляя из графа G повторяющиеся ребра, которые не влияют на достижимость. Необходимость в транзитивной редукции также возникает при рисовании графов, когда требуется убрать как можно больше ненужных ребер, чтобы не загромождать рисунок.

Хотя граф G может иметь только одно транзитивное замыкание, он может иметь несколько разных транзитивных редукций, в число которых входит он сам. Мы ищем наименьшую из этих редукций, но у такой задачи существуют разные формулировки.

- Простой линейный алгоритм для вычисления транзитивной редукции определяет компоненты сильной связности графа G, заменяет каждую из них ориентированным

циклом, а потом добавляет эти ребра к ребрам, связывающим разные компоненты. Хотя эта редукция, вероятно, не является оптимальной, на типичных графах она будет довольно близка к таковой.

Одним из возможных недостатков этого эвристического алгоритма состоит в том, что он может вставить в транзитивную редукцию графа G ребра, которых нет в исходном графе. Впрочем, это может и не представлять проблему в вашем конкретном приложении.

- Если все ребра нашей транзитивной редукции должны существовать в графе G, то найти редукцию минимального размера не удастся. Попробуем разобраться, почему это так. Рассмотрим ориентированный граф, состоящий из одной компоненты сильной связности, чтобы из любой вершины можно было попасть в любую другую вершину. Для такого графа наименьшей возможной транзитивной редукцией будет просто ориентированный цикл из n ребер. Это будет возможным тогда и только тогда, когда граф G является гамильтоновым графом, что доказывает, что задача поиска наименьшего подмножества ребер является NP-полной.

  Эвристическим подходом к поиску такой транзитивной редукции будет последовательный перебор каждого ребра и его удаление, если это удаление не меняет данную транзитивную редукцию. Эффективная реализация этого процесса позволит минимизировать время, затрачиваемое на проверку достижимости. Обратите внимание, что ориентированное ребро (i, j) можно удалить в любом случае, когда от вершины i к вершине j имеется другой путь, минующий данное ребро.

- Редукцию минимального размера, в которой допустимы ребра между произвольными парами вершин, можно найти за время $O(n^3)$. Но для большинства приложений, скорее всего, будет достаточно описанного выше простого, но действенного алгоритма, который, к тому же, легко программировать и который является более эффективным.

Реализации. Реализация алгоритма построения транзитивного замыкания, содержащаяся в библиотеке Boost Graph Library, создана на основе алгоритмов, предложенных в [Nuu95]. Библиотека LEDA (см. *раздел 19.1.1*) содержит реализации на языке C++ алгоритмов для вычисления как транзитивного замыкания, так и транзитивной редукции. Подробности о библиотеке LEDA см. в [MN99].

Создается впечатление, что ни одна из обычных библиотек Java не содержит реализации алгоритмов для поиска транзитивного замыкания или транзитивной редукции. Но пакет Graphlib содержит библиотеку Java Transitivity, которая имеет реализации обоих этих алгоритмов. Подробности можно найти на веб-сайте **http://www-verimag.imag.fr/~cotton/**.

Библиотека Combinatorica (см. [PS03]) содержит реализации (на языке пакета Mathematica) алгоритмов поиска транзитивного замыкания и транзитивной редукции. Подробности см. в *разделе 19.1.9*.

Примечания

Транзитивное замыкание и транзитивная редукция обсуждаются в работе [vL90a]. Эквивалентность перемножения матриц и вычисления транзитивного замыкания была доказана Фишером (Fischer) и Мейером (Meyer) в работе [FM71], а описания алгоритмов можно найти в книге [AHU74].

В последнее время наблюдался повышенный интерес исследователей к транзитивному замыканию, нашедший свое отражение в работе [Nuu95]. Пеннер (Penner) и Прасанна (Prasanna) (см. [PP06]) улучшили производительность алгоритма Варшалла (см. [War62]) приблизительно вдвое посредством реализации, чувствительной к кэшированию.

Доказательство эквивалентности транзитивного замыкания и транзитивной редукции, а также алгоритм вычисления транзитивной редукции за время $O(n^3)$ были представлены в работе [AGU72]. Результаты экспериментальных исследований алгоритмов вычисления транзитивного замыкания можно найти в [Nuu95], [PP06] и [SD75].

Важным аспектом оптимизации запросов к базам данным является оценка размера транзитивного замыкания. Алгоритм решения этой задачи за линейное время представлен в работе [Coh94].

Родственные задачи. Компоненты связности (см. *раздел 15.1*), поиск кратчайшего пути (см. *раздел 15.4*).

15.6. Паросочетание

Вход. Граф (возможно, взвешенный) $G = (V, E)$.

Задача. Найти наибольшее подмножество ребер E' множества E, для которого каждая вершина множества V инцидентна, самое большее, одному ребру из подмножества E' (рис. 15.7).

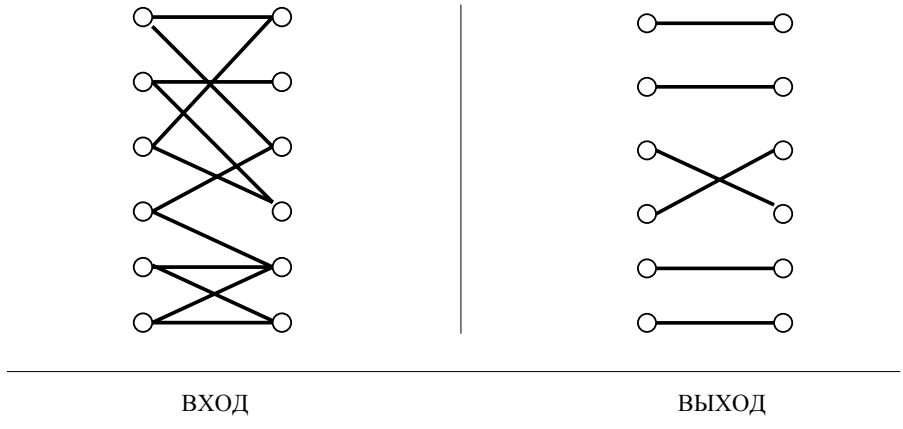


Рис. 15.7. Поиск паросочетаний

Обсуждение. Допустим, что мы руководим группой рабочих, и каждый из них умеет выполнять некоторое подмножество операций, необходимых для выполнения работы. Для решения этой задачи мы создадим граф, в котором вершины представляют как рабочих, так и операции, и соединяем ребрами рабочих с операциями, которые они могут выполнять. Чтобы не перегружать рабочих, мы должны поручить каждому рабочему одну операцию. Нашей целью является наибольший набор ребер, при котором не повторяются ни рабочие, ни операции, т. е. паросочетание.

Паросочетание — это мощный алгоритмический инструмент, и даже удивительно, что оптимальные паросочетания могут быть найдены достаточно эффективным образом. Необходимость в применении паросочетаний на практике возникают часто.

Переженить группу мужчин и женщин таким образом, чтобы каждая пара была счастливой, — еще один пример практического применения задачи паросочетания. В соответствующем графе каждая подходящая пара соединяется ребром. В синтетической биологии (см. [MPC+06]) требуется перемешать символы в строке S таким образом, чтобы максимизировать количество перемещенных символов. Например, символы строки *aaabc* можно перемешать так, что получится строка *bcaaa*, в которой только один символ занимает свою исходную позицию. Это вариант задачи паросочетания, в котором вместо мужчин фигурируют символы строки, а вместо женщин — позиции в строке (от 1 до $|S|$). Ребра соединяют символы с позициями в строке, в которых первоначально находились другие символы.

Эта основная структура паросочетаний может быть улучшена несколькими способами, и при этом останется все той же задачей о назначениях. Вы должны ответить на следующие вопросы.

- *Является ли граф двудольным?* В большинстве задач о назначениях используются двудольные графы, как, например, в классической задаче о назначении заданий рабочим. Это благоприятное обстоятельство, т. к. для поиска паросочетаний в двудольных графах существуют более быстрые и простые алгоритмы.

- *Можно ли некоторым рабочим назначать не одно, а несколько заданий?* Естественные обобщения задачи о назначениях включают поручение некоторым конкретным рабочим нескольких заданий или (что эквивалентно) назначение нескольких рабочих на одно задание. В данном случае целью является не столько паросочетание, сколько постановка "галочек" в сводной таблице работ. Такие требования можно моделировать созданием стольких копий рабочего, сколько заданий мы хотим ему поручить. Как раз этот подход применялся в предыдущем примере с перестановками символов строки.

- *Является ли граф взвешенным?* Многие приложения паросочетаний основаны на невзвешенных графах. Предположим, мы хотим максимизировать общее количество выполняемых работ, равнозначных друг другу. В этом случае мы ищем паросочетание *максимальной мощности*, в идеале *совершенное* паросочетание, в котором каждая вершина паросочетания сопоставляется с другой.

  Но для других приложений нам нужно дополнить каждое ребро весом, например, отражающим способность рабочего к выполнению данного задания. Теперь задача превращается в задачу создания паросочетания с максимальным весом, т. е. набора независимых ребер с максимальным общим весом.

Эффективные алгоритмы для построения паросочетаний работают, создавая в графах увеличивающие пути. Для частичного паросочетания M в графе G увеличивающим путем является путь P из ребер, которые то входят в паросочетание M, то выходят из него. Имея такой увеличивающий путь, мы можем расширить паросочетание на одно ребро, заменив четные ребра пути P из M нечетными ребрами этого пути. Согласно теореме Берга, паросочетание является максимальным тогда и только тогда, когда оно не содержит ни одного увеличивающего пути. Поэтому мы можем создавать паросочетания максимальной мощности, выполняя поиск увеличивающих путей, прекращая поиск, когда больше нет таких путей.

Задача поиска паросочетания в общих графах более сложная, т. к. в них возможны увеличивающие пути, являющиеся циклами нечетной длины (т. е. с одной и той же начальной и конечной вершиной). Такие циклы невозможны в двудольных графах, которые не содержат их по определению.

Стандартные алгоритмы вычисления паросочетания в двудольном графе основаны на потоках в сети. В них применяется простое преобразование двудольного графа в эквивалентный потоковый граф. Реализация такого подхода приведена в *разделе 6.5*.

Однако необходимо иметь в виду, что для решения задач паросочетания во взвешенных графах требуются другие подходы, наиболее известным из которых является венгерский алгоритм.

Реализации. Эндрю Голдберг (Andrew Goldberg) в соавторстве с другими исследователями разработал высокопроизводительные коды для вычисления паросочетаний как во взвешенных, так и в невзвешенных двудольных графах. В частности, Голдберг и Кеннеди написали на языке C программу CSA для вычисления паросочетаний во взвешенных двудольных графах, основанную на потоках в сети, масштабирующих стоимость (см. [GK95]). Более быстрая программа BIM для вычисления паросочетаний в невзвешенных двудольных графах, основанная на методах с использованием увеличивающих путей, представлена в работе [CGM+98]. Обе программы можно загрузить для некоммерческого использования с веб-сайта **http://www.avglab.com/andrew/soft.html**.

Первое соревнование по реализации алгоритмов DIMACS (см. [JM93]) было посвящено, в основном, потокам в сетях и паросочетаниям. На этом соревновании было выявлено несколько удачных генераторов экземпляров задач, а также ряд реализаций алгоритмов для вычисления паросочетаний максимальной мощности и паросочетаний с максимальным весом. Эти программы можно загрузить по адресу **ftp://dimacs.rutgers.edu/pub/netflow/matching/**. В их число входят следующие программы:

- решатель на языке FORTRAN 77 для вычисления паросочетаний максимальной мощности, разработанный Мэттингли (Mattingly) и Ричи (Ritchey);
- решатель на языке C, разработанный Эдвардом Ротбергом (Edward Rothberg), для вычисления паросочетаний максимальной мощности, реализующий алгоритм Габова (Gabow's algorithm) с временем исполнения $O(n^3)$;
- решатель, разработанный Эдвардом Ротбергом, для вычисления паросочетаний с максимальным весом. Это более медленный, но более общий решатель, чем вышеупомянутый решатель этого же автора.

Обширная библиотека классов C++ GOBLIN (**http://www.math.uni-augsburg.de/~fremuth/goblin.html**) предназначена для решения всех стандартных задач оптимизации графов, включая вычисление паросочетаний во взвешенных двудольных графах. Библиотека LEDA (см. *раздел 19.1.1*) содержит эффективные реализации на языке C++ для вычисления паросочетаний как максимальной мощности, так и максимального веса, как на двудольных, так и на общих графах.

Эффективную программу Blossom IV (см. [CR99]) на языке C для вычисления совершенных паросочетаний минимального веса можно загрузить с веб-сайта **http://www2.isye.gatech.edu/~wcook/software.html**. Реализацию для вычисления паросочетаний максимальной мощности в общих графах за время $O(mn\alpha(m, n))$ можно загрузить с веб-сайта **http://www.cs.arizona.edu/~kece/Research/software.html**.

База графов Stanford GraphBase (см. *раздел 19.1.8*) содержит реализацию венгерского алгоритма для вычисления паросочетаний в двудольных графах. В целях визуализации взвешенных двудольных графов Дональд Кнут использует оцифрованную версию картины Мона Лиза и осуществляет в ней поиск пикселов максимальной яркости в отдельных строчках и столбцах. Паросочетание также используется для построения оригинальных портретов в стиле "домино".

Примечания

Книга Ловаша (Lovasz) и Пламмера (Plummer) [LP86] является исключительно полным справочником по теории паросочетаний и алгоритмам их вычисления. Среди обзорных статей по алгоритмам вычисления паросочетаний особого внимания заслуживает работа [Gal86]. Хорошие описания алгоритмов, использующих потоки в сети для вычисления паросочетаний в двудольных графах, содержатся в работах [CLRS01], [Eve79a] и [Man89], а венгерского алгоритма — в работах [Law76] и [PS98]. Самый лучший алгоритм для вычисления максимальных паросочетаний в двудольных графах, разработанный Хопкрофтом (Hopcroft) и Карпом (Karp) (см. [HK73]), последовательно находит кратчайшие увеличивающие пути (вместо использования сетевого потока) и исполняется за время $O(\sqrt{n}m)$. Время исполнения венгерского алгоритма равно $O(n(m + n\log n))$.

Алгоритм Эдмондса (Edmonds) (см. [Edm65]) для вычисления паросочетаний максимальной мощности представляет большой исторический интерес, поскольку он вызывает вопросы о том, какие задачи можно решить за полиномиальное время. Описания алгоритма Эдмондса можно найти в [Law76], [PS98] и [Tar83]. Реализация алгоритма Эдмондса, принадлежащая Габову (см. [Gab76]), выполняется за время $O(n^3)$. Время исполнения самого лучшего известного алгоритма вычисления общих паросочетаний равно $O(\sqrt{n}m)$ (см. [MV80]).

Рассмотрим задачу паросочетания мужчин и женщин. Предположим, входной экземпляр содержит ребра (B_1, G_1) и (B_2, G_2), причем в действительности мужчина B_1 и женщина G_2 предпочитают друг друга своим текущим партнерам. В реальной жизни эти двое, скорее всего, сойдутся, расторгнув свои браки. Паросочетание, не имеющее таких пар, называется *устойчивым* (stable). Всестороннее изучение устойчивых паросочетаний содержится в книге [Gl89]. Интересно, что независимо от многообразия взаимных предпочтений мужчин и женщин, всегда существует хотя бы одно устойчивое паросочетание. Более того, такое паросочетание можно найти за время $O(n^2)$ (см. статью [GS62]).

В двудольных графах размер максимального паросочетания равен размеру минимального вершинного покрытия. Это означает, что на двудольных графах задачу о минимальном вершинном покрытии и задачу о максимальном независимом множестве можно решить за полиномиальное время.

Родственные задачи. Эйлеров цикл (см. *раздел 15.7*), потоки в сети (см. *раздел 15.09*).

15.7. Задача поиска эйлерова цикла и задача китайского почтальона

Вход. Граф $G = (V, E)$.

Задача. Найти самый короткий маршрут, который проходит по каждому ребру графа G, по крайней мере, один раз (рис. 15.8).

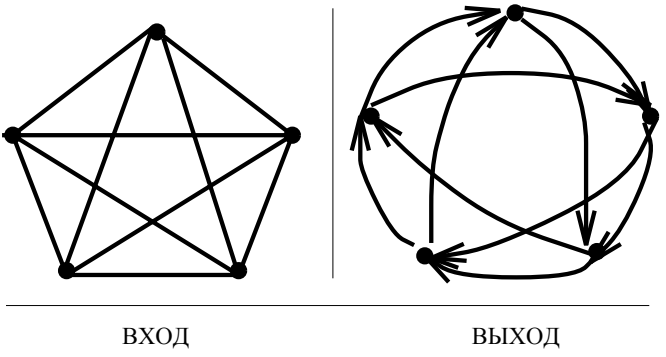


Рис. 15.8. Эйлеров цикл

Обсуждение. Допустим, что вам предоставили карту города и дали задание разработать маршрут для мусоровозов, или для снегоуборочных машин, или почтальонов. В каждом случае необходимо целиком пройти каждую улицу, по крайней мере, один раз. Для эффективного выполнения задания нам нужно минимизировать время прохождения маршрута или (что равносильно) общее расстояние или количество ребер графа, по которым осуществляется обход.

Рассмотрим другой пример: проверка системы меню телефонного автоматизированного справочника. Каждая опция "Нажмите такую-то кнопку для получения такой-то информации" рассматривается как ребро между двумя вершинами графа. Тестировщик меню ищет самый эффективный способ обхода этого графа, стараясь пройти по каждому ребру, по крайней мере, один раз.

Такие задачи являются вариантами задачи поиска эйлерова цикла, которую лучше всего сравнить с головоломкой, где нужно нарисовать геометрическую фигуру, не отрывая карандаш от бумаги и не проводя его по уже нарисованным линиям. В них нужно найти путь или цикл в графе, который проходит по каждому ребру ровно один раз.

Перечислим признаки наличия в графе эйлерова цикла или пути (цепи).

- *Неориентированный* граф содержит эйлеров *цикл* тогда и только тогда, когда граф связный и все его вершины имеют четную степень.
- *Неориентированный* граф содержит эйлеров *путь* тогда и только тогда, когда граф связный и все (кроме двух) вершины имеют четную степень. Эти две вершины будут начальной и конечной точками любого пути.
- *Ориентированный* граф содержит эйлеров *цикл* тогда и только тогда, когда граф сильно связный и все вершины имеют одинаковые степень захода и степень исхода.
- *Ориентированный* граф содержит эйлеров *путь* от вершины x к вершине y тогда и только тогда, когда граф связный и все другие вершины имеют одинаковые степень захода и степень исхода, причем вершина x имеет степень захода на единицу меньше, чем степень исхода, а вершина y имеет степень захода на единицу больше, чем степень исхода.

Эти свойства эйлеровых графов позволяют с легкостью проверить наличие цикла: сначала проверяем граф на связность, выполнив обход в глубину или ширину, а потом подсчитываем количество вершин с нечетной степенью. Явный поиск цикла также за-

нимает линейное время и выполняется следующим способом. Посредством обхода в глубину находим в графе произвольный цикл. Удаляем этот цикл из графа и повторяем процедуру до тех пор, пока все множество ребер не будет разбито на циклы, не имеющие общих ребер. Так как удаление цикла уменьшает степень каждой вершины на четное число, то оставшийся граф продолжает удовлетворять все тем же эйлеровым степенным условиям. Эти циклы будут иметь общие вершины (т. к. граф является связным) и поэтому их можно объединить в виде "восьмерок" по общим вершинам. Объединяя таким образом все извлеченные циклы, мы создаем один цикл, содержащий все ребра.

Эйлеров цикл, если такой существует в графе, решает задачу построения маршрута снегоуборочной машины, т. к. длина любого пути, проходящего по каждому ребру только один раз, должна быть минимальной. Но на практике дорожные сети редко удовлетворяют эйлеровым степенным условиям. Тогда нам приходится решать более общую задачу — задачу китайского почтальона, в которой требуется найти минимальный цикл, проходящий через каждое ребро, по крайней мере, один раз. Этот минимальный цикл никогда не пройдет более двух раз ни по какому ребру, так что удовлетворительный маршрут можно найти для любой дорожной сети.

Оптимальный маршрут китайского почтальона можно создать, добавив соответствующие ребра в граф G, чтобы сделать его эйлеровым. В частности, мы находим в графе G кратчайший путь между каждой парой вершин с нечетной степенью. Добавив путь между двумя вершинами с нечетной степенью, мы превращаем их в вершины с четной степенью, что приближает граф G к тому, чтобы он стал эйлеровым. Задача поиска наилучшего множества кратчайших путей для добавления к графу G сводится к поиску совершенного паросочетания с минимальным весом в специальном графе G'.

Для неориентированных графов вершины графа G' соответствуют вершинам нечетной степени графа G, где вес ребра (i, j) определяется как длина кратчайшего пути от вершины i к вершине j в графе G. Для ориентированных графов вершины графа G' соответствуют вершинам графа G, несбалансированным по степени, при этом все ребра в графе G' направлены из вершин с нехваткой степени исхода в вершины с нехваткой степени захода. Таким образом, для ориентированного графа G будет достаточно использовать алгоритмы для вычисления паросочетания в двудольных графах. Когда граф становится эйлеровым, то искомый цикл можно получить за линейное время, используя только что описанную процедуру.

Реализации. Реализацию алгоритма поиска эйлерова цикла содержат многие библиотеки графов, но реализации решений задачи китайского почтальона встречаются менее часто. Мы рекомендуем реализацию на языке Java для решения задачи китайского почтальона, разработанную Тимблби (Thimbleby); см. [Thi03]. Программу можно загрузить с веб-сайта **http://www.cs.swan.ac.uk/~csharold/cpp/index.html**.

Библиотека GOBLIN содержит обширную коллекцию процедур на языке C++ для всех стандартных задач оптимизации на графах, включая процедуру решения задачи китайского почтальона как на ориентированных, так и на неориентированных графах. Загрузить библиотеку можно с веб-сайта **http://www.math.uni-augsburg.de/~fremuth/goblin.html**. Библиотека LEDA (см. *раздел 19.1.1*) предоставляет все инструменты для эффективной реализации решателей для задач поиска эйлерова цикла, паросочетаний, и кратчайших путей в двудольных и общих графах.

Библиотека Combinatorica (см. [PS03]) содержит реализации (на языке пакета Mathematica) для решения задач поиска эйлеровых циклов и последовательностей де Брейна (de Bruijn sequences). Подробности см. в *разделе 19.1.9*.

Примечания

История теории графов начинается в 1736 г., когда Леонард Эйлер взялся за решение задачи о семи мостах Кенигсберга. Город Кенигсберг (в настоящее время Калининград) расположен на берегах реки. Во времена Эйлера оба берега и два острова соединялись семью мостами. Эту планировку можно смоделировать в виде мультиграфа с четырьмя вершинами и семью ребрами. Эйлер захотел узнать, возможно ли пройти по всем мостам ровно по одному разу и возвратиться в исходную точку (и впоследствии такой маршрут получил название эйлерова цикла). Он доказал, что искомый маршрут невозможен, т. к. все четыре вершины имели нечетную степень. Мосты были разрушены во время Второй мировой войны. История возникновения задачи рассказана в книге [BLW76].

Описания алгоритмов с линейным временем исполнения для создания эйлеровых циклов (см. [Ebe88]) можно найти в работах [Eve79a] и [Man89]. Простым и элегантным методом создания эйлеровых циклов является алгоритм Флери (Fleury); см. [Luc91]. Начинаем обход с любой вершины и удаляем пройденные ребра. Единственным критерием выбора следующего ребра является отказ от прохода по мосту (т. е. ребру, удаление которого разъединит граф) до тех пор, пока не останется других вариантов.

Метод поиска эйлерова пути играет важную роль в параллельных алгоритмах на графах. Многие параллельные алгоритмы начинаются с построения остовного дерева, для которого потом устанавливается корень по методу поиска эйлерова пути. Описания параллельных алгоритмов см. в таких учебниках, как [J92], а последние результаты практического применения — в работе [CB04]. Для подсчета эйлеровых циклов в графах существуют эффективные алгоритмы (см. [HP73]).

Задача поиска кратчайшего маршрута, проходящего через все ребра графа, была впервые представлена китайским ученым Кваном (Kwan) (см. [Kwa62]), что и определило ее название — задача китайского почтальона. Решение задачи китайского почтальона посредством алгоритма поиска паросочетаний в двудольных графах было разработано Эдмондсом (Edmonds) и Джонсоном (Johnson); см. [EJ73]. Этот алгоритм работает как с ориентированными, так и с неориентированными графами, но для смешанных графов задача является NP-полной (см. [Pap76a]). Смешанные графы содержат как ориентированные, так и неориентированные ребра. Описание алгоритма решения задачи китайского почтальона приводится в [Law76].

Последовательность де Брейна (de Bruijn) S протяженностью n на алфавите Σ из α символов представляет собой циклическую строку длиной α^n, содержащую все строки длиной n, как подстроки последовательности S, каждую ровно один раз. Например, для $n = 3$ и $\Sigma = \{0, 1\}$ циклическая строка 00011101 содержит по порядку подстроки 000, 001, 011, 111, 110, 101, 010, 100. Последовательности де Брейна можно рассматривать как "руководство для вскрытия сейфа", которое предоставляет самый короткий набор поворотов ручки кодового замка, причем α дает количество позиций, достаточное для перебора всех комбинаций длины n.

Эти последовательности можно генерировать, создав ориентированный граф, все вершины которого представляют строки α^{n-1} длиной $n - 1$, и в котором ребро (u, v) существует тогда и только тогда, когда $u = s_1 s_2 ... s_{n-1}$ и $v = s_2 ... s_{n-1} s_n$. Любой эйлеров цикл на таком графе описывает последовательность де Брейна. Описания последовательностей де Брейна и способы их построения приведены в [Eve79a] и [PS03].

Родственные задачи. Паросочетание (см. *раздел 15.6*), гамильтонов цикл (см. *раздел 16.5*).

15.8. Реберная и вершинная связность

Вход. Граф G и, возможно, пара вершин s и t.

Задача. Найти наименьшее подмножество вершин (или ребер), удаление которых разъединит граф G или, как вариант, отделит вершину s от вершины t (рис. 15.9).

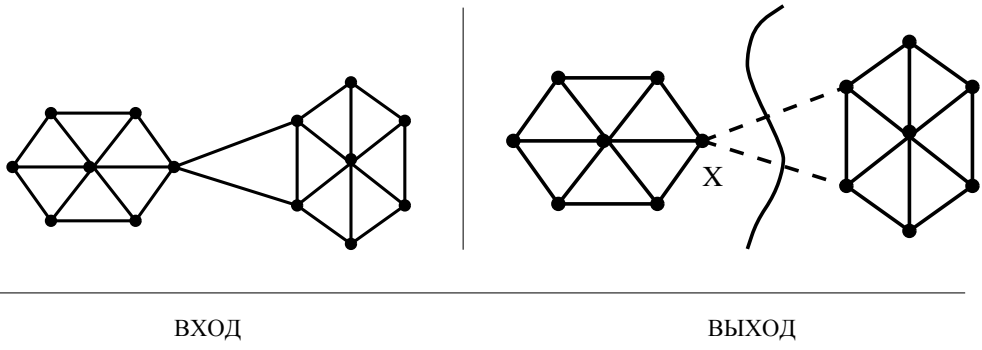


Рис. 15.9. Выяснение числа реберной связности

Обсуждение. Вопрос о связности графов часто возникает в задачах, имеющих отношение к надежности сетей. Например, в контексте телефонных сетей число вершинной связности — это наименьшее количество коммутационных станций, которые нужно вывести из строя, чтобы разорвать сеть, т. е. сделать невозможной связь между любой парой исправных коммутационных станций. А число реберной связности в этом случае — наименьшее количество линий связи, которые нужно разорвать, чтобы достичь этой же цели.

Итак, числом реберной (вершинной) связности графа G называется наименьшее количество ребер (вершин), удаление которых нарушает связность графа. Между этими двумя величинами существует тесная связь. Число вершинной связности всегда меньше или равно числу реберной связности, т. к. удаление одной вершины из каждого ребра разреза нарушает связность графа. Но возможно существование и меньшего подмножества вершин. Верхним пределом, как для реберной, так и для вершинной связности, является минимальная степень вершины, т. к. удаление всех смежных с ней вершин (или ребер, связывающих ее с соседними вершинами) разбивает граф на две части, одна из которых состоит из единственной вершины.

Представляет интерес поиск ответов на следующие вопросы.

- *Является ли граф связным?* Самой простой задачей является проверка связности графа. Все компоненты связности можно найти за линейное время посредством простого обхода в глубину или ширину (см. *раздел 15.1*). Для ориентированных графов следует выяснить, является ли граф *сильно связным*, т. е. в нем существует ориентированный путь между любой парой вершин. В *слабо связном* графе возможно существование путей к узлам, из которых нет возврата.

- *Содержит ли граф "слабое звено"?* Граф G называется двусвязным, если для нарушения его связности требуется удалить две вершины. Вершина, удаление которой приводит к потере связности графа, называется *шарниром*. Мост представляет собой аналогичное понятие для ребер.

  Самый простой алгоритм идентификации шарниров (или мостов) — удалять по одной вершине (или ребру) и после каждого удаления проверять оставшийся граф на связность с помощью обхода в глубину или ширину. Для обеих задач существуют более сложные алгоритмы на основе обхода в глубину. Реализацию см. в *разделе 5.9.2*.

- *Как разделить граф на равные части?* Нередко требуется найти небольшой разрез, который разделяет граф на приблизительно равные части. Например, мы хотим разделить компьютерную программу на две части, которые проще сопровождать. Для этого мы можем создать граф, вершины которого будут соответствовать процедурам программы. Соединим ребрами взаимодействующие вершины (процедуры, одна из которых вызывает другую). Далее нам нужно разделить все процедуры на наборы приблизительно одинакового размера так, чтобы минимизировать количество пар взаимодействующих процедур, расположенных по разные стороны от линии раздела.

  Эта задача называется задачей разбиения графа, которая рассматривается в *разделе 16.6*. Хотя эта задача является NP-полной, для ее решения существуют приемлемые эвристические методы.

- *Можно ли разбивать граф произвольным образом, или требуется разбить конкретную пару вершин?* Существует две разновидности общей задачи связности. В одной требуется найти наименьший разрез для всего графа, а в другой — наименьший разрез, чтобы отделить вершину s от вершины t. Любой алгоритм поиска связности между вершинами s и t можно использовать с любой из $n(n-1)/2$ пар вершин, что даст нам алгоритм для проверки общей связности. Не так очевиден тот факт, что для проверки реберной связности будет достаточно $n-1$ проходов, однако, мы знаем, что после удаления разреза вершина v_1 и хотя бы одна из остальных $n-1$ вершин будут находиться в разных компонентах.

Компоненты связности можно найти с помощью методов, применяемых для решения задач о потоках в сети. В задаче о потоках в сети (см. *раздел 15.9*) взвешенный граф рассматривается как сеть труб, в которой каждое ребро/труба имеет максимальную пропускную способность. Целью задачи является максимизировать поток между двумя данными вершинами графа. Максимальный поток между вершинами v_i и v_j в графе G в точности равен весу наименьшего набора ребер, которые нужно удалить для разъединения этих вершин. Таким образом, число реберной связности можно найти, минимизировав поток между вершиной v_i и каждой вершиной из остальных $n-1$ вершин невзвешенного графа G. Почему? Потому что после удаления минимального реберного разреза вершина v_i будет отделена от какой-либо другой вершины.

Вершинная связность описывается *теоремой Менгера* (Menger's theorem), в которой утверждается, что граф является k-связным тогда и только тогда, когда каждую пару вершин связывают, по меньшей мере, k путей, не имеющих общих вершин. Здесь тоже можно использовать задачу о потоках в сети, т. к. поток объемом k между парой вер-

шин свидетельствует о наличии k путей, не имеющих общих ребер. Чтобы применить теорему Менгера, создаем такой граф G', в котором любой набор путей, не имеющих общих ребер, соответствует путям, не имеющим общих вершин, в графе G. Для этого каждая вершина v_i графа G заменяется такими двумя вершинами $v_{i,1}$ и $v_{i,2}$, что ребро $(v_{i,1}, v_{i,2}) \in G'$ для всех $v_i \in G$, а каждое ребро $(v_i, x) \in G$ заменяется ребрами $(v_{i,j}, x_k)$, где $j \neq k \in \{0, 1\}$ в графе G'. Таким образом, каждому из путей, не имеющих общих вершин, в графе G соответствуют два пути, не имеющих общих ребер, в графе G'.

Реализации. Коллекция MINCUTLIB содержит коды нескольких высокопроизводительных алгоритмов поиска разреза, включая алгоритмы, использующие метод потоков в сети и метод сжатия. Эти реализации были разработаны Чекури (Chekuri) и его коллегами в ходе многочисленных экспериментов (см. [CGK+97]). Для некоммерческого использования программы можно загрузить с веб-сайта **http://www.avglab.com/andrew/soft.html**. Здесь же можно загрузить и полную версию самой работы [CGK+97], описывающей эти алгоритмы и необходимые для их быстрого исполнения эвристические методы.

Большинство библиотек структур данных для представления графов, рассматриваемых в *разделе 15.1*, содержат процедуры проверки на связность и двусвязность. Библиотека процедур на языке C++ Boost Graph Library (см. [SLL02]) содержит процедуры проверки реберной связности. Загрузить библиотеку можно с веб-сайта **http://www.boost.org/libs/graph/doc**.

Библиотека GOBLIN (**http://www.math.uni-augsburg.de/~fremuth/goblin.html**) содержит обширную коллекцию процедур для всех стандартных задач оптимизации на графах, включая процедуры выяснения реберной и вершинной связности.

Библиотека LEDA на языке C++ (см. *раздел 19.1.1*) предоставляет широкую поддержку для выяснения низкоуровневой связности (как двусвязных, так и трехсвязных компонент), реберной связности и минимального разреза.

Библиотека Combinatorica (см. [PS03]) предоставляет реализации (на языке пакета Mathematica) процедур выяснения реберной и вершинной связности, а также компонент связности, двусвязности и сильной связности с мостами и шарнирами. Подробности см. в *разделе 19.1.9*.

Примечания

Хорошие описания использования потоков в сети для выяснения реберной и вершинной связности можно найти в [Eve79a] и [PS03]. Правильность этих алгоритмов основана на теореме Менгера (см. [Men27]), утверждающей, что связность определяется количеством путей, не имеющих общих ребер, и путей, не имеющих общих вершин и связывающих пару вершин. Теорема о минимальном потоке и максимальном разрезе была доказана Фордом (Ford) и Фалкерсоном (Fulkerson); см. [FF62].

Самые быстрые алгоритмы поиска минимального разреза и реберной связности основаны на методе сжатия графа, а не на методе потоков в сети. Сжатие ребра (x, y) в графе G соединяет две инцидентные ему вершины в одну, удаляя петли, но оставляя мультиребра. Любая последовательность таких сжатий может увеличить (но не уменьшить) минимальный разрез в графе G, и не меняет разрез, если не затрагивается ребро разреза. Каргер (Karger) предоставил изящный рандомизированный алгоритм поиска минимального разреза, основанный на наблюдении, что вероятность изменения минимального разреза в те-

чение любой случайной последовательности удалений является ничтожно малой. Самая быстрая версия алгоритма Каргера имеет время исполнения ($m\lg^3 n$) (см. [Kar00]). Отличный обзор рандомизированных алгоритмов, включая алгоритм Каргера, представлен в книге [MR95].

Детерминистический алгоритм, использующий метод сжатия для поиска минимального разреза за время $O(n(m + n\log n))$, можно найти в работе [NI92]. В каждой итерации этот алгоритм находит и сжимает ребро, которое доказуемо не входит в минимальный разрез. Результаты экспериментальных сравнений алгоритмов поиска минимальных разрезов представлены в работах [CGK+97] и [NOI94].

Методы поиска минимального разреза нашли применение в области распознавания образов, в частности, для сегментации изображений. В своей работе [BK04] Бойков и Колмогоров приводят экспериментальную оценку алгоритмов поиска минимального разреза в контексте такого применения.

Матулой (Matula) был разработан алгоритм, не использующий методы потоков в сети, для проверки графа на реберную k-связность за время $O(kn^2)$. Для определенных небольших значений k существуют более быстрые алгоритмы выяснения k-связности. Все трехсвязные компоненты графа можно сгенерировать за линейное время (см. [HT73a]), в то время как 4-связность можно проверить за время $O(n^2)$.

Родственные задачи. Компоненты связности (см. *раздел 15.1*), потоки в сети (см. *раздел 15.9*), разбиение графов (см. *раздел 16.6*).

15.9. Потоки в сети

Вход. Ориентированный граф G, каждое ребро которого $e = (i, j)$ имеет пропускную способность c_e, и две вершины — исток s и сток t.

Задача. Найти максимальный поток, который можно направить из вершины s в вершину t, соблюдая ограничивающие условия пропускной способности каждого ребра (рис. 15.10).

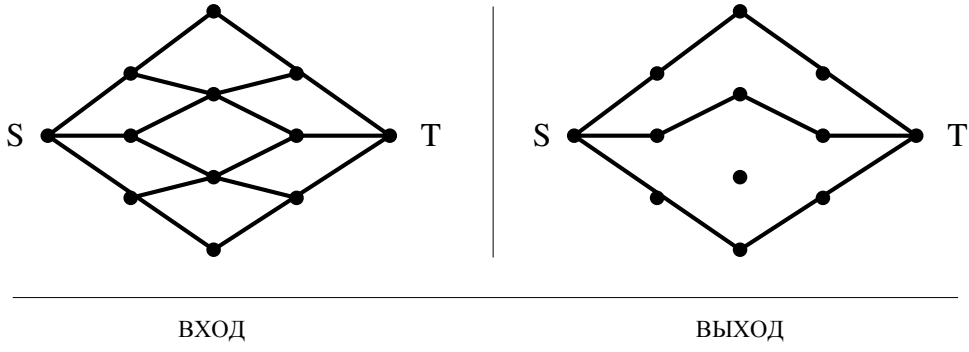


Рис. 15.10. Потоки в сети

Обсуждение. Область применения задачи о потоках в сети гораздо шире, чем прокладка водопровода. Задача о потоках возникает при поиске наиболее экономически эффективного способа транспортировки продукции между множеством предприятий и множеством магазинов или при распределении ресурсов в сетях связи.

Реальная мощь задачи о потоках в сети проявляется в том, что большое количество возникающих на практике задач линейного программирования можно смоделировать в виде задач о потоках в сети и соответствующие алгоритмы позволяют решить эти задачи намного быстрее, чем применение методов линейного программирования общего назначения. Несколько рассмотренных в этой книге задач на графах можно решить, используя методы потоков в сети, включая задачи поиска паросочетаний в двудольных графах, кратчайшего пути и компонент связности.

Вы должны развить в себе умение распознавать возможность моделирования стоящей перед вами задачи в виде задачи о потоках в сети. Такое умение требует практического опыта и теоретических знаний. Я рекомендую сначала создать для вашей задачи модель из области линейного программирования, а потом сравнить ее с моделями для двух основных классов задач о потоках в сети.

- *Максимальный поток.* Здесь мы хотим найти максимальный объем потока из вершины s в вершине t, соблюдая ограничивающие условия пропускной способности ребер графа G. Пусть переменная x_{ij} обозначает объем потока, проходящего из вершины i через ориентированное ребро (i, j). Так как объем потока, проходящего через это ребро, ограничен его пропускной способностью c_{ij}, то $0 \le x_{ij} \le c_{ij}$ для $1 \le i, j \le n$.

  Кроме того, про каждую вершину, не являющуюся ни истоком, ни стоком, можно сказать, что в нее входит поток такого же объема, какой выходит из нее, поэтому

  $$\sum_{j=1}^{n} x_{ij} - \sum_{j=1}^{n} x_{ji} = 0 \text{ for } 1 \le i \le n$$

  Нам нужно найти такой набор значений, который максимизирует поток, идущий в вершину t, а именно $\sum_{i=1}^{n} x_{it}$.

- *Поток минимальной стоимости.* Здесь каждое ребро (i, j) имеет дополнительный параметр, а именно стоимость d_{ij} перемещения единицы потока от вершины i к вершине j. Также установлен целевой объем потока f, который мы хотим направить от вершины s к вершине t с минимальной общей стоимостью. Следовательно, нам требуется найти такой набор значений, который минимизирует следующую формулу:

  $$\sum_{j=1}^{n} d_{ij} \cdot x_{ij}$$

  соблюдая ограничивающие условия реберной и вершинной пропускной способности для максимального потока, а также дополнительное ограничивающее условие, что $\sum_{i=1}^{n} x_{it} = f$.

При этом учитываются следующие особенности:

- *наличие нескольких истоков и/или стоков.* Это обстоятельство не является проблемой, т. к. мы можем модифицировать сеть таким образом, чтобы создать вершину-суперисток, которая питает все истоки, и вершину-суперсток, которая поглощает все стоки;

- *пропускная способность каждого ориентированного ребра равна либо 0, либо 1.* Для таких сетей существуют более быстрые алгоритмы. Подробности см. в *подразделе "Примечания"*;

- *одинаковая стоимость всех ребер.* В таком случае используйте более простые и быстрые алгоритмы для решения задачи о максимальном потоке, а не задачи о потоке минимальной стоимости. Задача о максимальном потоке без стоимости ребер возникает во многих приложениях, включая поиск компонент реберной и вершинной связности и паросочетаний в двудольных графах;

- *перемещение по сети разнотипных продуктов.* В телекоммуникационной сети каждое сообщение снабжается информацией о его источнике и месте назначения. Каждый узел-адресат должен получать только те сообщения, которые предназначены ему, а не общий объем сообщений от всех отправителей. Эту ситуацию можно смоделировать в виде задачи о многопродуктовом потоке, где разные запросы определяют разные продукты, и нам требуется удовлетворить все запросы, не превысив пропускную способность ребра.

  Если допускаются фрагментированные потоки, задачу о многопродуктовом потоке можно будет решить методами линейного программирования. К сожалению, задача о нефрагментированном многопродуктовом потоке является NP-полной даже для двух продуктов.

Алгоритмы решения задач о потоках в сети могут быть сложными и требовать значительных усилий для повышения их производительности. Поэтому мы настоятельно рекомендуем использовать уже имеющиеся реализации алгоритмов, вместо того, чтобы пытаться разработать свою собственную. Есть прекрасные программы, которые описаны в *подразделах "Реализации"* и *"Примечания"*. Существует два основных класса алгоритмов вычисления потоков в сети:

- *методы увеличивающих путей.* Эти алгоритмы последовательно находят путь от истока к стоку с положительной пропускной способностью и добавляют его к общему потоку. Можно доказать, что поток в сети является оптимальным тогда и только тогда, когда в ней нет увеличивающего пути. Так как каждое добавление пути увеличивает поток, то в итоге должен быть найден глобальный максимум. Разница между алгоритмами этого типа заключается в том, как они выбирают увеличивающий путь. Если не проявлять аккуратность, то каждый новый увеличивающий путь будет увеличивать общий поток лишь ненамного, вследствие чего поиск максимального потока может занять много времени;

- *методы проталкивания предпотока.* Эти алгоритмы проталкивают потоки от одной вершины к другой, первоначально игнорируя ограничивающее условие, что для каждой вершины исходящий поток должен быть равным входящему. Алгоритмы проталкивания предпотока работают быстрее, чем методы увеличивающих путей, поскольку возможно одновременное увеличение нескольких путей. Эти алгоритмы очень популярны, и они реализованы в программах, упоминаемых в следующем подразделе.

Реализации. Высокопроизводительные программы для вычисления максимальных потоков и потоков минимальной стоимости были разработаны Эндрю Голдбергом

(Andrew Goldberg) и его коллегами. Для вычисления максимальных потоков существуют программы HIPR и PRF (см. [CG94]), причем в большинстве случаев рекомендуется использовать программу HIPR. Для вычисления потоков минимальной стоимости лучшей является программа CS (см. Gol97]). Все эти программы написаны на языке C и доступны (для некоммерческого использования) на сайте **http://www.avglab.com/andrew/soft.html**.

Библиотека GOBLIN (**http://www.math.uni-augsburg.de/~fremuth/goblin.html**) содержит обширную коллекцию процедур на языке C++ для всех стандартных задач оптимизации на графах, включая несколько процедур поиска максимального потока и потока минимальной стоимости. То же самое относится и к библиотеке LEDA, если требуется решение, имеющее коммерческую ценность. Подробности см. в *разделе 19.1.1*.

На первом соревновании DIMACS по реализациям алгоритмов для задач о потоках в сети и о паросочетаниях (см. [JM93]) было собрано несколько реализаций и генераторов, которые можно загрузить из каталога **ftp://dimacs.rutgers.edu/pub/netflow/maxflow**. В число этих программ входят написанные на языке C реализация решения задачи о потоках в сети методом проталкивания предпотока и реализация одиннадцати вариантов решения задачи о потоках в сети, включая старые алгоритмы Диница (Dinic) и Карзанова (Karzanov).

Примечания

Лучшей книгой, посвященной задачам о потоках в сети и их применению, является [AMO93]. Хорошие описания алгоритмов для вычисления потоков в сети можно найти в [CCPS98], [CLRS01] и [PS98]. Обсуждение сложности задач о многопродуктовом потоке (см. [Ita78]) можно найти в книге [Eve79a].

Максимальный поток и реберная связность графов взаимообусловлены. Фундаментальная теорема о минимальном потоке и максимальном разрезе была доказана Фордом (Ford) и Фалкерсоном (Fulkerson); см. [FF62]. Более простые и эффективные алгоритмы вычисления минимального разреза графов обсуждаются в *разделе 15.8*.

Принято считать, что вычисление потока в сети должно занимать время $O(nm)$; причем наблюдается постоянное снижение временной сложности этого вычисления. Историю алгоритмов для решения данной задачи см. в книге [AMO93]. Время исполнения самого быстрого алгоритма вычисления потока в сети равно $O(nm \lg(n^2/m))$ (см. [GT88]). Экспериментальные исследования алгоритмов вычисления потоков минимальной стоимости можно найти в [GKK74] и [Gol97].

Потоки информации в сети можно моделировать в виде многопродуктовых потоков, принимая во внимание то обстоятельство, что создание дубликатов информации и манипулировании ими может устранить надобность в отдельных путях от истоков к стокам, когда одну и ту же информацию нужно доставить в несколько стоков. Эти идеи используются в области сетевого кодирования (см. [YLCZ05]) для достижения теоретических пределов прохождения информационных потоков, установленных теоремой о минимальном потоке и максимальном разрезе.

Родственные задачи. Линейное программирование (см. *раздел 13.6*), паросочетание (см. *раздел 15.6*), связность (см. *раздел 15.8*).

15.10. Рисование графов

Вход. Граф G.

Задача. Начертить граф G таким образом, чтобы точно отобразить его структуру (рис. 15.11).

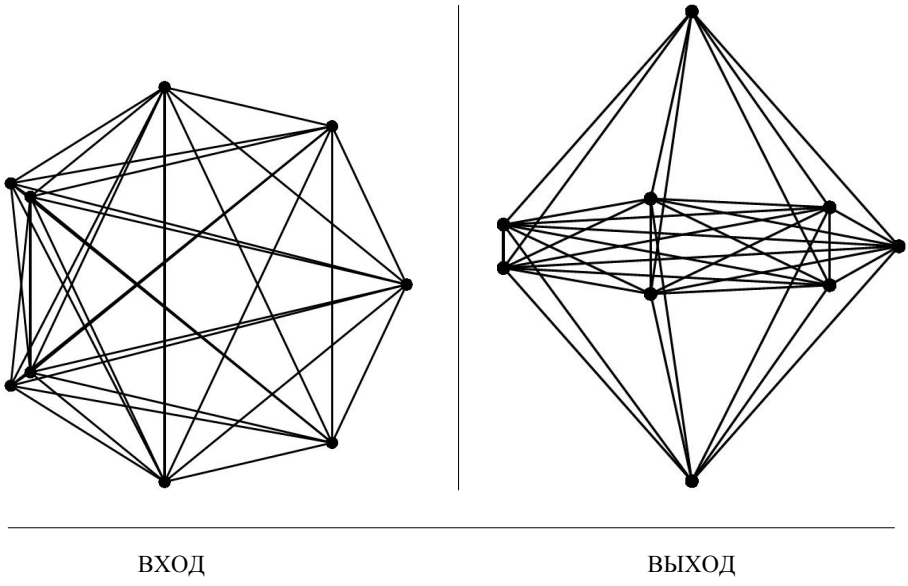


Рис. 15.11. Пример графа

Обсуждение. Задача рисования графов постоянно возникает в приложениях, но по своей природе она не имеет четкого определения. Что такое хорошо нарисованный граф? Нам требуется найти алгоритм, который визуально представляет структуру графа самым понятным для человека образом. В то же самое время мы хотим, чтобы нарисованный граф удовлетворял определенным эстетическим критериям.

К сожалению, это расплывчатые критерии, по которым невозможно разработать оптимизирующий алгоритм. Более того, очень легко создать сильно отличающиеся друг от друга представления одного и того же графа, каждое из которых будет подходящим для определенного контекста. Например, на рис. 16.10 показаны три разных начертания графа Петерсена (Petersen graph). Какое из них является "правильным"?

Однако существуют и более четкие критерии, позволяющие частично оценить качество рисунка графа:

- *пересечения.* Рисунок должен содержать как можно меньше пересечений ребер, чтобы не отвлекать внимание;
- *площадь.* Рисунок должен занимать как можно меньшую площадь, и в то же самое время никакие две вершины не должны быть расположены слишком близко друг к другу;
- *длина ребер.* Рисунок не должен содержать длинные ребра, т. к. они перекрывают другие элементы рисунка;

- *угловое разрешение.* Рисунок не должен содержать очень острые углы между ребрами, инцидентными одной вершине, т. к. это может вызвать частичное или полное наложение линий друг на друга;
- *пропорциональность сторон.* Соотношение высоты к ширине рисунка должно, по возможности, совпадать с соответствующей характеристикой целевого носителя (как правило, это соотношение сторон экрана компьютерного монитора, равное 4/3).

К сожалению, эти критерии противоречат друг другу, и задача выбора наилучшего рисунка графа из непустого подмножества таких рисунков, вероятно, будет NP-полной.

Здесь необходимо сделать два замечания. Для графов, не имеющих естественной симметрии или структуры, существование по-настоящему хорошего рисунка, скорее всего, просто невозможно. Это особенно относится к графам, имеющим более 10–15 вершин. Рисунок большого, плотного графа сможет отобразить не каждый монитор. Полный граф из 100 вершин содержит приблизительно 5 000 ребер. На мониторе с разрешением 1 000×1 000 пикселов получается по 200 пикселов на ребро. Что можно увидеть в таком случае?

Учитывая все сказанное, следует признать, что алгоритмы рисования графов могут быть довольно эффективными. Кроме того, экспериментирование с ними может доставить массу удовольствия. Чтобы выбрать подходящий алгоритм для рисования вашего графа, попытайтесь найти ответ на следующие вопросы.

- *Должны ли все ребра быть прямыми, или допускаются дуги?* Алгоритмы, рисующие прямые линии, сравнительно просты, но имеют свои ограничения. Для визуализации сложных графов, например, электрических схем, лучше всего подходят прямоугольные ломаные (такие, что все отрезки расположены или горизонтально, или вертикально, т. е. наклонные линии не разрешаются). При этом каждое ребро графа представлено последовательностью вертикальных и горизонтальных отрезков, соединенных вершинами или точками изгибов.
- *Возможен ли естественный, специфический для приложения рисунок?* Если ваш граф представляет дорожную сеть, то вы вряд ли найдете лучший способ нарисовать его иначе, чем поместив все вершины в те же позиции, что и города на карте. Этот принцип действителен для многих других приложений.
- *Является ли граф планарным графом или деревом?* В таком случае воспользуйтесь одним из алгоритмов, описанных в *разделах 15.11* и *15.12*, для рисования планарных графов или деревьев.
- *Является ли граф ориентированным?* Ориентация ребра оказывает значительное влияние на характеристики рисунка. При рисовании ориентированных бесконтурных графов важно, чтобы все ребра были направлены в соответствии с некоторым логическим принципом, например, слева направо или сверху вниз.
- *Насколько быстрым должен быть ваш алгоритм?* Если алгоритм предназначается для интерактивного обновления и вывода графов на экран, то он должен работать очень быстро. В таком случае, ваш выбор ограничен инкрементальными алгоритмами, которые изменяют положение вершин только в непосредственной близости от редактируемой вершины. Если же вы рисуете красивое изображение для дли-

тельного изучения, то вы можете потратить дополнительное время на его оптимизацию.

- *Имеет ли граф оси симметрии?* Граф-результат на рис. 15.11 выглядит привлекательно, потому что исходный граф имеет горизонтальную ось симметрии. Оси симметрии в графе можно определить, найдя его *автоморфизмы* (изоморфизмы с самим собой). Все автоморфизмы графа можно без труда найти с помощью программ поиска изоморфизмов (см. *раздел 16.9*).

Я рекомендую сначала создать черновой рисунок графа: расположить вершины по кругу на одинаковом расстоянии друг от друга и соединить их прямыми ребрами. Такие рисунки легко поддаются программированию и быстро выполняются. Их значительное преимущество заключается в том, что никакие два ребра не накладываются друг на друга, т. к. в таких рисунках нет трех вершин, расположенных в одну линию. Этих нежелательных эффектов трудно избежать, когда в рисунке разрешается иметь внутренние вершины. Приятным сюрпризом, связанным с круговыми рисунками графов, может оказаться иногда проявляющаяся симметрия, обусловленная тем, что вершины выводятся в том порядке, в котором они определяются в графе. Рисунок можно значительно улучшить, минимизировав длину ребер или количество их пересечений за счет перестановки вершин методом имитации отжига.

Хороший универсальный эвристический алгоритм моделирует граф в виде системы пружин, а при расстановке вершин применяет принцип минимизации потенциальной энергии. Пусть смежные вершины притягивают друг друга с усилием, пропорциональным логарифму расстояния между ними, в то время как несмежные вершины отталкивают друг друга с усилием, пропорциональным расстоянию между ними. Эти условия заставляют ребра укорачиваться и в то же самое время вынуждают вершины удаляться друг от друга. Поведение такой системы можно аппроксимировать, определив силу, действующую на каждую вершину в определенный момент времени, а потом переместив каждую вершину на небольшое расстояние в соответствующем направлении. После нескольких таких итераций система должна стабилизироваться на приемлемом рисунке графа. Рисунок 15.11 демонстрирует эффективность встраивания пружин в конкретный небольшой граф.

Если вам нужен алгоритм, рисующий граф ломаными линиями, я рекомендую изучить системы, представленные далее, и системы, описанные в книге [JM03], чтобы решить, может ли какой-либо из них подойти для решения вашей задачи. Чтобы разработать свой собственный алгоритм рисования графов, вам придется выполнить значительный объем работы.

Задача рисования графов имеет свои "подводные камни", связанные с расстановкой меток у ребер и вершин. Метки нужно помещать близко к ребрам и вершинам, чтобы однозначно идентифицировать их, но, в то же время, они не должны накладываться ни на элементы графа, ни друг на друга. Можно доказать, что оптимизация размещения меток является NP-полной задачей, но она может быть эффективно решена эвристическими методами, используемыми для решения задачи разложения по контейнерам (см. *раздел 17.9*).

Реализации. Одной из популярных и хорошо поддерживаемых программ для рисования графов является программа GraphViz, разработанная Стивеном Нортом (Stephen

North) из компании Bell Laboratories. Подробности см. на сайте **http://www.graphviz.org**. Программа отображает ребра в виде сплайнов и может создавать рисунки довольно больших и сложных графов. Собственно говоря, возможностей этой программы было достаточно для удовлетворения всех моих профессиональных требований по рисованию графов на протяжении нескольких лет.

Все библиотеки структур данных для представления графов, упомянутые в *разделе 12.4*, содержат средства визуализации графов. Библиотека Boost Graph Library предоставляет интерфейс к программе GraphViz. Особенно подходящими для интерактивных приложений являются библиотеки графов на языке Java, в особенности библиотека JGraphT (**http://jgrapht. sourceforge.net**).

Для задачи рисования графов существуют очень хорошие коммерческие программы, включая программное обеспечение компании Tom Sawyer Software (**www.tomsawyer.com**), семейство продуктов yFiles (**www.yworks.com**) и пакет iLOG JViews (**http://www-01.ibm.com/software/integration/visualization/jviews/enterprise/**). Пакет Pajek (см. [NMB05]) специально предназначен для рисования социальных сетей. Загрузить пакет можно с веб-сайта **http://vlado.fmf.uni-lj.si/pub/networks/pajek**. Все эти пакеты предоставляют бесплатные пробные версии или версии для некоммерческого использования.

Библиотека Combinatorica (см. [PS03]) содержит несколько реализаций (на языке пакета Mathematica) алгоритмов рисования графов, включая круговые, пружинные и упорядоченные укладки. Для дополнительной информации по Combinatorica, см. *раздел 19.1.9*.

Примечания

Существует многочисленное сообщество исследователей в области рисования графов, которое стимулирует проведение ежегодной конференции, посвященной этой теме. Протоколы конференции публикуются издательством Springer-Verlag в серии Lecture Notes in Computer Science. Ознакомившись с этими протоколами, вы получите хорошее представление о последних достижениях в области рисования алгоритмов и о направлениях, в которых ведутся разработки. Книга [Tam08], является, пожалуй, наиболее полным обзором этой области.

Отличными книгами по алгоритмам рисования графов являются [BETT99] и [KW01]. Материал книги [JM03] организован по системам, а не по алгоритмам, однако содержит технические подробности о методах рисования, применяемых в каждой системе. Эвристические алгоритмы для маркировки графов описываются в работах [BDY06] и [WW95].

Задача равномерного размещения n точек вдоль окружности является тривиальной. Но задача размещения точек на поверхности сферы намного труднее. Для облегчения решения этой задачи были разработаны обширные таблицы для $n \leq 130$ (см. [HSS07]).

Родственные задачи. Рисование деревьев (см. *раздел 15.11*), проверка на планарность (см. *раздел 15.12*).

15.11. Рисование деревьев

Вход. Дерево T, являющееся графом, не содержащим циклов.

Задача. Создать рисунок дерева T (рис. 15.12).

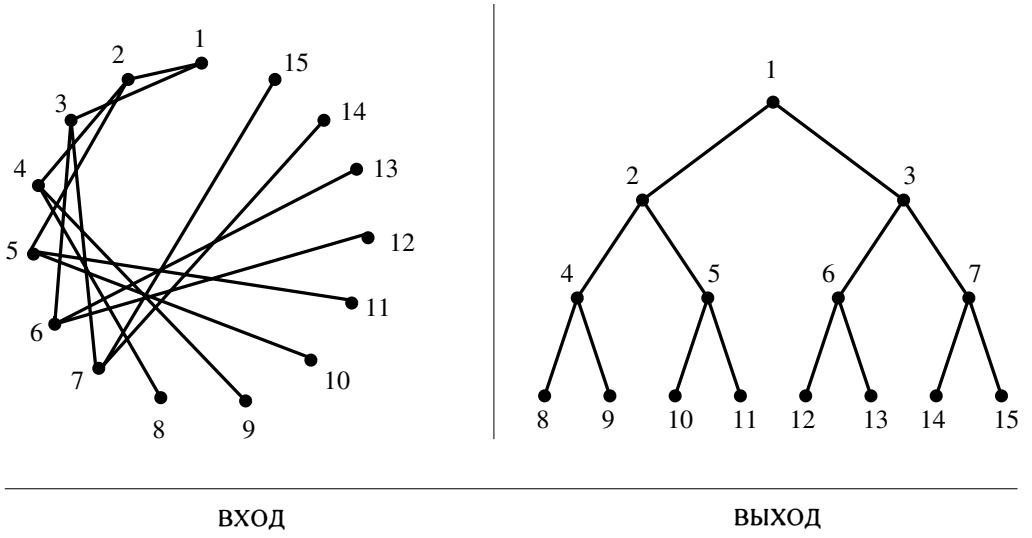


Рис. 15.12. Пример дерева

Обсуждение. Во многих приложениях требуется создавать рисунки деревьев. Диаграммы деревьев часто применяются для отображения иерархической структуры каталогов файловой системы и их обхода. Поиск в Google по фразе "tree drawing software" ("программы рисования деревьев") возвращает около 88 тысяч результатов, включая специализированные приложения для визуализации генеалогических деревьев, синтаксических деревьев (дерево грамматического разбора предложения), а также эволюционные филогенетические деревья.

Каждое приложение предъявляет свои требования к внешнему виду дерева. Однако основным вопросом в области рисования деревьев является выяснение, какое дерево нужно нарисовать — свободное или корневое.

- *Корневые деревья* определяют иерархически упорядоченную структуру, исходящую из одного узла, называемого корнем. Любой рисунок такого дерева должен отражать иерархию его элементов, а также любые специфичные для приложения ограничивающие условия на порядок, в котором должны отображаться дочерние узлы. Например, генеалогические деревья имеют корень, а дочерние узлы в них обычно отображаются слева направо, по очередности рождения.
- *Свободные деревья* не содержат никакой информации о структуре, кроме топологии соединений. Например, минимальное остовное дерево графа не имеет корня, поэтому иерархическое отображение этого дерева не имеет смысла. Рисунки таких деревьев могут обладать свойствами рисунка полного дерева, например, дорожной карты, на которой расстояние между населенными пунктами определяет минимальное остовное дерево.

Деревья являются планарными графами, поэтому их можно и нужно рисовать, без пересечения ребер. Для этой цели годится любой алгоритм рисования на плоскости из *раздела 15.12*, но для создания рисунков деревьев на плоскости можно использовать намного более простые алгоритмы. В частности, эвристические алгоритмы, основан-

ные на "встраивании" пружин (см. *раздел 15.10*), хорошо подходят для рисования свободных деревьев, хотя в некоторых приложениях они могут оказаться слишком медленными.

В большинстве естественных алгоритмов рисования деревьев предполагается, что они работают с корневыми деревьями. Но их можно с таким же успехом применять и для рисования свободных деревьев, выбрав одну из вершин в качестве корневой. Этот псевдокорень можно выбрать произвольно или, что даже лучше, использовать для него *центральную* вершину дерева. Центральная вершина минимизирует максимальное расстояние до других вершин. У деревьев центром всегда является или одна, или две смежных вершины. Этот центр дерева можно найти за линейное время, последовательно удаляя все листья, пока не останется только центральный узел.

При рисовании корневых деревьев вы имеете выбор между *упорядоченной* и *радиальной* укладками.

- *Упорядоченная укладка.* Корень помещается вверху по центру страницы, после чего страница разбивается сверху вниз на полосы, количество которых равно степени корня. Удаление корня создает несколько поддеревьев, количество которых равно степени корня и каждое из которых расположено в своей собственной полосе. Каждое поддерево рисуется рекурсивно, причем его новый корень (вершина, смежная со старым корнем) помещается в центре его полосы на фиксированном расстоянии от верха страницы, а старый корень соединяется линией с новым. На рис. 15.12 изображена упорядоченная укладка сбалансированного двоичного дерева.

  Такие упорядоченные укладки особенно эффективны для корневых деревьев, представляющих какую-либо иерархическую структуру, будь то генеалогическое дерево, структура данных или служебная лестница. Расстояние по вертикали показывает удаленность каждого узла от корня. К сожалению, такое последовательное деление на полосы, в конце концов, приводит к появлению очень узких полосок, вследствие чего большинство вершин оказывается в небольшой части страницы. Старайтесь отрегулировать ширину каждой полосы так, чтобы отразить общее количество узлов, которые она будет содержать. И не бойтесь занять соседнюю область на полосе после завершения построения более коротких поддеревьев.

- *Радиальная укладка.* Свободные деревья лучше рисовать, используя радиальную укладку, в которой корень или центр дерева помещается в центре страницы. Пространство вокруг этой центральной вершины разделяется на секторы для каждого поддерева. Хотя при этом также может возникнуть проблема "скученности", радиальные укладки используют место на странице эффективнее, чем упорядоченные, и являются более естественными для свободных деревьев. Ранг вершин в терминах расстояния от центра иллюстрируется концентрическими кругами вершин.

Реализации. Одной из наиболее популярных программ для рисования графов является программа GraphViz, разработанная и Стивеном Нортом (Stephen North) из компании Bell Laboratories. Подробности см. на сайте **http://www.graphviz.org**. Программа представляет ребра в виде сплайнов и может создавать рисунки довольно больших и сложных графов. В течение многих лет эта программа удовлетворяет все мои профессиональные потребности в рисовании графов.

Для рисования графов и деревьев существуют очень хорошие коммерческие программы, включая программное обеспечение компании Tom Sawyer Software (**www.tomsawyer.com**), семейство продуктов yFiles (**www.yworks.com**) и пакет iLOG JViews (**http://www-01.ibm.com/software/integration/visualization/jviews/enterprise/**). Все эти пакеты предоставляют бесплатные пробные версии или версии для некоммерческого использования.

Библиотека Combinatorica (см. [PS03]) содержит несколько реализаций (на языке пакета Mathematica) алгоритмов рисования деревьев, включая упорядоченные и радиальные укладки. Дополнительную информацию по Combinatorica см. в *разделе 19.1.9*.

Примечания

Все книги и обзоры по рисованию графов содержат описания алгоритмов, специально предназначенных для рисования деревьев. Книга [Tam08] является, пожалуй, наиболее полным обзором положения дел в этой области. Отличными книгами по алгоритмам рисования графов являются [BETT99] и [KW01]. Материал книги [JM03] организован по системам, а не по алгоритмам, однако содержит технические подробности о методах рисования, применяемых в каждой системе.

Исследования эвристических алгоритмов для укладки деревьев предпринимались многими учеными (см. например, работы [RT81, Moe90]), а работа [BJL06] отражает последние достижения в этой области. При некоторых ограничениях, носящих эстетический характер, задача рисования графов является NP-полной (см. [SR83]).

Некоторые алгоритмы укладки деревьев получаются из приложений, не связанных с рисованием графов. Метод Емде Боаса организации двоичного дерева обеспечивает более высокую производительность при работе с внешними устройствами хранения данных, чем обычный двоичный поиск, правда, за счет более сложной реализации. Информацию по этой и другим нечувствительным к кэшированию структурам можно найти в [ABF05].

Родственные задачи. Рисование графов (см. *раздел 15.10*), рисование на плоскости (см. *раздел 15.12*).

15.12. Планарность

Вход. Граф G.

Задача. Выяснить, является ли граф G планарным, т. е. возможно ли нарисовать его на плоскости так, чтобы никакие два ребра не пересекались. Если возможно, создать такой рисунок (рис. 15.13).

Обсуждение. Рисование на плоскости (или укладка) графа дает ясное представление о его структуре, поскольку не содержит ни одного пересечения ребер, которое можно было бы принять за вершину. Графы, моделирующие дорожные сети, компоновку печатных плат и т. п., являются планарными по своей сути, т. к. они полностью определяются плоскостными структурами.

Планарные графы обладают разнообразными свойствами, которые можно использовать, чтобы получить более быстрые алгоритмы для решения многих задач. Самый важный факт, который вы должны знать, заключается в том, что все планарные графы являются *разреженными*. Для любого нетривиального планарного графа $G = (V, E)$ справедлива формула Эйлера $|E| \leq 3|V| - 6$. Это означает, что у каждого планарного

графа количество ребер линейно зависит от количества вершин, а также, что каждый планарный граф имеет вершину со степенью ≤5. Любой подграф планарного графа является планарным, поэтому должна существовать последовательность вершин низкой степени, которые можно удалить из графа G, сводя его к пустому графу.

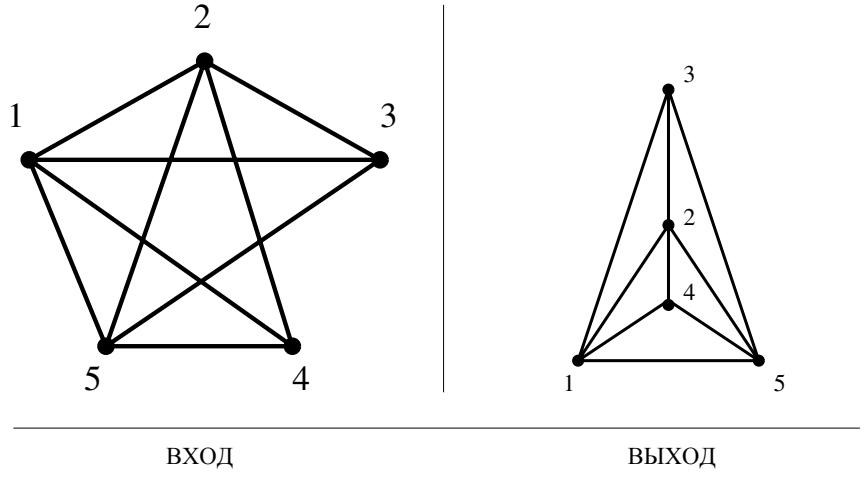


Рис. 15.13. Рисунок графа на плоскости

Чтобы вы получили более полное представление о тонкостях рисования графов на плоскости, я рекомендую вам создать укладку для графа $K_5 - e$, показанного на рис. 15.13, а потом попытаться создать такую укладку, в которой все ребра прямые. Затем добавьте к графу недостающее ребро и попробуйте проделать то же самое для графа K_5.

Исследование планарности значительно способствовало развитию теории графов. Однако следует признать, что на практике необходимость в проверке графов на планарность возникает сравнительно редко. Большинство систем рисования графов не стремится специально создавать планарные укладки. На сайте этой книги, Algorithm Repository (**http://www.cs.sunysb.edu/~algorith**), тема выяснения планарности является самой малопосещаемой (см. [Ski99]). Несмотря на это, вам полезно уметь обращаться с планарными графами.

Необходимо понимать разницу между задачей проверки графа на планарность (т. е. выяснением, возможен ли рисунок графа на плоскости) и задачей создания планарной укладки (т. е. собственно созданием рисунка), хотя и то и другое можно выполнить за линейное время. Многие эффективные алгоритмы для работы с планарными графами не используют рисование, а применяют описанную ранее последовательность удаления вершин с низкими степенями.

Алгоритмы проверки на планарность начинают работу с размещения на плоскости произвольного цикла из графа, после чего изучают дополнительные пути в графе, соединяющие вершины с этим циклом. При пересечении двух таких путей один из них нужно рисовать вне цикла, а другой в цикле. При попарном пересечении трех путей конфликт нельзя разрешить, и, следовательно, граф не может быть планарным. Алгоритмы с линейным временем исполнения, выполняющие проверку на планарность,

основаны на обходе в глубину, но они достаточно сложны, так что вам лучше поискать существующую реализацию, а не пытаться создать собственную.

Алгоритмы выявления пересекающихся путей можно использовать для создания планарной укладки, добавляя пути в рисунок по одному. К сожалению, так как эти алгоритмы работают инкрементально, ничто не может помешать им поместить слишком много вершин и ребер в сравнительно небольшую область рисунка. Размещение чрезмерного количества элементов графа в ограниченной области является большой проблемой, т. к. затрудняет восприятие полученных рисунков графов. Поэтому были разработаны более эффективные алгоритмы, создающие решеточные укладки, в которых вершины расположены в решетке размером $(2n-4)\times(n-2)$. В результате никакие области рисунка не "захламлены", и ни одно из ребер не является слишком длинным. Тем не менее, полученные рисунки обычно выглядят не так естественно, как хотелось бы.

Для непланарных графов часто требуется найти рисунок с минимальным количеством пересечений ребер. К сожалению, задача вычисления количества пересечений ребер графа является NP-полной. Однако существуют эвристические алгоритмы, которые выделяют большой планарный подграф из графа G, выполняют укладку этого подграфа, а потом вставляют по одному остальные ребра таким образом, чтобы минимизировать количество пересечений. Этот способ не очень подходит для плотных графов, в которых невозможно избежать большого количества пересечений, но он хорошо работает с почти планарными графами, такими как многослойные печатные платы, или сети дорог с эстакадами. Большие планарные подграфы могут быть найдены за счет модифицирования алгоритмов проверки на планарность таким образом, чтобы все обнаруженные проблемные ребра удалялись.

Реализации. Библиотека LEDA (см. *раздел 19.1.1*) содержит реализации алгоритмов с линейным временем исполнения как для проверки на планарность, так и для создания решеточных укладок. Программа проверки планарности этих реализаций возвращает препятствующий подграф Куратовского (см. *подраздел "Примечания"*) для любого предположительно непланарного графа, предоставляя убедительное доказательство его непланарности.

Среда JGraphEd (**http://www.jharris.ca/JGraphEd**) для рисования графов, написанная на языке Java, содержит несколько реализаций алгоритмов проверки на планарность и создания планарных укладок, включая как реализацию алгоритма Бута-Люкера (Booth-Lueker) на основе PQ-деревьев, так и реализацию современных алгоритмов создания решеточной укладки.

PIGALE (**http://pigale.sourceforge.net**) — это среда, включающая в себя библиотеку алгоритмов для работы с графами и редактор графов. Она написана на языке C++ и ориентирована, в основном, на планарные графы. Она содержит разнообразные алгоритмы для создания рисунков графов на плоскости, а также эффективные алгоритмы для проверки на планарность и построения препятствующего подграфа ($K_{3,3}$ или K_5), если такой существует.

Алгоритм GRASP (Greedy Randomized Adaptive Search, "жадный" рандомизированный адаптивный поиск) для поиска наибольшего планарного подграфа реализован Рибейро (Ribeiro) и Ресенде (Resende) (см. [RR99]) языке FORTRAN и числится в коллекции

алгоритмов ACM под номером 797 (см. *раздел 19.1.6*). Эту реализацию можно загрузить с веб-сайта **http://www.research.att.com/~mgcr/src**.

Примечания

Куратовский в своей работе [Kur30] впервые охарактеризовал планарные графы, а именно показал, что они не содержат подграфа, гомеоморфного графу $K_{3,3}$ или K_5. Таким образом, если вы все еще пытаетесь выполнить планарную укладку графа K_5, можете прекратить это занятие. Теорема Фари ([F48]) утверждает, что любой планарный граф можно нарисовать так, что все ребра будут прямыми.

Первый алгоритм с линейным временем исполнения для рисования графов был разработан Хопкрофтом (Hopcroft) и Тарьяном (Tarjan); см. [HT74]. Альтернативный алгоритм проверки на планарность на основе PQ-деревьев был разработан Бутом (Booth) и Люкером (Lueker); см. [BL76]. Упрощенные алгоритмы проверки на планарность описываются в работах [BCPB04], [MM96] и [SH99]. Эффективные алгоритмы создания решеточных укладок размером $2n \times n$ были впервые представлены в [dFPP90]. Книга [NR04] содержит хороший обзор алгоритмов рисования планарных графов.

Внешнепланарным называется граф, который можно нарисовать так, чтобы все его вершины принадлежали внешней грани рисунка. Для таких графов характерно отсутствие подграфа, гомеоморфного графу $K_{2,3}$. Распознать их и выполнить их укладку можно за линейное время.

Родственные задачи. Разбиение графов (см. *раздел 16.6*), рисование деревьев (см. *раздел 15.11*).

ГЛАВА 16

Сложные задачи на графах

По поводу алгоритмов для работы с графами существует достаточно циничное мнение, что любая задача, возникающая на практике, слишком сложна. Действительно, ни для одной задачи, представленной в этой главе, не предложено алгоритма с полиномиальным временем исполнения. Все эти задачи являются NP-полными, за исключением задачи об изоморфизме графов, для которой вопрос сложности остается открытым.

Теория NP-полноты гласит, что если для *какой-то одной* NP-полной задачи существует алгоритм с полиномиальным временем исполнения, то такой алгоритм должен существовать для *всех* NP-полных задач. "Перевернув" это утверждение, мы получим, что сведение задачи к заведомо NP-полной является достаточным доказательством отсутствия эффективного алгоритма решения исходной задачи.

Тем не менее, не стоит терять надежду на решение вашей задачи, даже если она рассматривается в этой главе. Для каждой задачи мы предлагаем наиболее эффективный подход к ее решению, будь то комбинаторный поиск, эвристический метод, аппроксимирующий алгоритм или алгоритмы для ограниченных экземпляров. Для сложных задач требуется иная методика, чем для задач, решаемых за полиномиальное время, но при правильном подходе с ними можно успешно справиться.

Следующие книги будут весьма полезны вам при решении NP-полных задач:

- [GJ79] — классический справочник по теории NP-полноты. Он содержит краткий каталог свыше 400 NP-полных задач с соответствующими ссылками и комментариями. Обращайтесь к этому каталогу, как только у вас возникнет сомнение в существовании эффективного алгоритма для решения вашей задачи. К этой книге из моей библиотеки я обращаюсь чаще, чем к остальным;
- [ACG$^+$OS] — эта исключительно полезная книга посвящена аппроксимирующим алгоритмам. Ее справочный раздел находится по адресу **www.nada.kth.se/~viggo/problemlist**. Именно с него вы должны начинать поиски хорошего эвристического алгоритма для решения любой задачи;
- [Vaz04] — подробное изложение теории аппроксимирующих алгоритмов, написанное авторитетным исследователем этой области;
- [Hoc96] — первый обзор аппроксимирующих алгоритмов для решения NP-полных задач. Стремительное развитие данной области привело к тому, что эта хорошая книга немного устарела;
- [Gon07] — этот справочник по аппроксимирующим и метаэвристическим алгоритмам содержит свежие обзоры разных методов решения сложных задач, как прикладных, так и теоретических.

16.1. Задача о клике

Вход. Граф $G = (V, E)$.

Задача. Найти наибольшее подмножество вершин $S \in V$, такое что для всех $(x, y) \in S$ справедливо $(x, y) \in E$ (рис. 16.1).

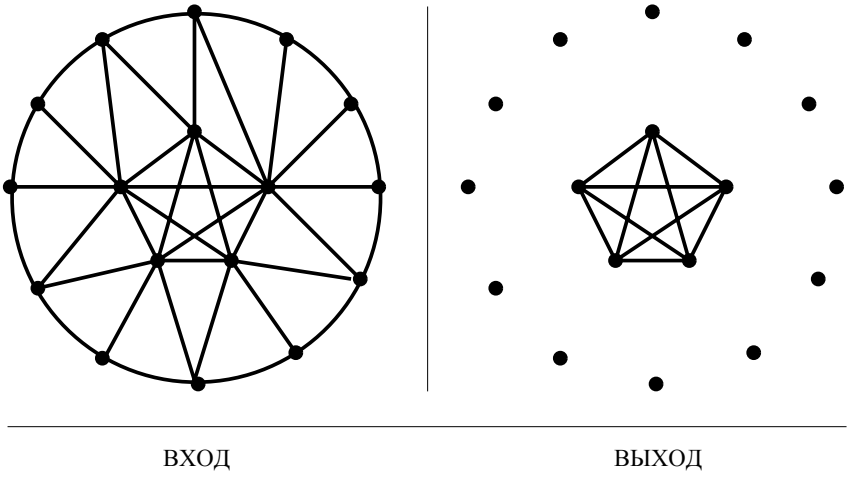


Рис. 16.1. Клика

Обсуждение. Почти каждый из нас может вспомнить, что в его школе существовала "клика" — группа друзей, которые постоянно держались вместе и оказывали влияние на всю социальную жизнь школы. Рассмотрим граф, представляющий школьный социум. Вершины этого графа соответствуют ученикам, а ребра соединяют друзей. Тогда школьная клика будет представлена *кликой* из теории графов, т. е. полным подграфом в графе дружеских отношений.

Задача выявления "кластеров", или родственных объектов, часто сводится к поиску больших клик графа. Интересным примером использования клик является программа, разработанная Налоговой службой США для борьбы с организованным мошенничеством в налоговой сфере. В Налоговую службу поступает большое количество ложных деклараций, составленных в расчете на незаконное получение компенсации. Однако создание большого количества действительно разных деклараций требует больших усилий. Программа Налоговой службы создает графы, в которых вершины представляют поданные налоговые декларации, и соединяет ребрами заявки, которые выглядят подозрительно похожими. Любая большая клика в графе помогает выявить мошенничество.

Так как любые две соединенные ребром вершины представляют клику, трудность заключается не в выявлении хоть какой-нибудь клики, а в поиске большой клики. А это уже сложно, поскольку задача поиска максимальной клики является NP-полной. Ситуацию усугубляет тот факт, что даже аппроксимация клики с точностью до $n^{1/2-\epsilon}$ является сложной задачей. Теоретически, задача поиска клики является самой сложной в этой книге. Итак, на что мы можем надеяться?

Попытайтесь найти ответы на следующие вопросы.

- *Будет ли достаточно максимальной клики?* Максимальной называется клика, которую нельзя увеличить добавлением вершины. Отсюда не следует, что максимальная клика обязательно должна быть сравнима по величине с наибольшей возможной кликой, но это вполне вероятно. Чтобы найти максимальную (и не исключено, что большую) клику, отсортируем вершины по степени в порядке убывания, занесем первую вершину в клику, а потом проверим каждую из оставшихся вершин на смежность со всеми вершинами, добавленными в клику на данном этапе. Если проверяемая вершина удовлетворит этому условию, добавим ее в клику, а если нет, то перейдем к следующей вершине. При использовании битового вектора для пометки вершин, находящихся в клике, этот алгоритм может исполняться за время $O(n + m)$. Альтернативный подход заключается во внесении элемента случайности в упорядочение вершин и принятии самой большой из максимальных клик, найденных после некоторого количества попыток.
- *Не лучше ли искать большой плотный подграф?* Настаивать на выявлении клик при поиске кластеров рискованно, т. к. нехватка одного ребра может исключить вершину из рассмотрения. Поэтому имеет смысл искать большие *плотные* подграфы, т. е. подмножества вершин, соединенных большим количеством ребер. По определению, клики являются наиболее плотными возможными подграфами.

  Наибольшее множество вершин, у которых порожденный подграф имеет минимальную вершинную степень $\geq k$, можно найти с помощью простого алгоритма с линейным временем исполнения. Начнем с удаления всех вершин со степенью меньшей, чем k. Это может понизить степень других вершин до значения, меньшего k, если они были смежными для удаленных вершин с низкой степенью. Повторяя этот процесс до тех пор, пока степень всех оставшихся вершин не будет превышать k, мы получим наибольший подграф с высокой степенью. Можно создать реализацию этого алгоритма со временем исполнения $O(n + m)$, используя списки смежности и очередь с приоритетами, рассматриваемую в *разделе 12.2*. Продолжая удалять вершины с самой низкой степенью, мы, в конечном счете, получим клику или набор клик, но они могут состоять всего лишь из двух вершин.
- *Как поступать с планарным графом?* Планарные графы не могут содержать клики, имеющие больше четырех вершин, в противном случае они не могут быть планарными. Так как каждое ребро определяет клику из двух вершин, то единственными представляющими интерес кликами в планарном графе являются клики из трех и четырех вершин. Эффективные алгоритмы поиска таких небольших клик рассматривают вершины в порядке возрастания их степени. Любой планарный граф должен содержать вершину со степенью, не превышающей 5 (см. *раздел 15.12*), поэтому чтобы найти клику, содержащую эту вершину, нужно выполнить полную проверку лишь области постоянного размера. Эта вершина удаляется, после чего остается меньший планарный граф, содержащий другую вершину низкой степени. Этот процесс проверки и удаления продолжается до тех пор, пока граф не станет пустым.

Если вам действительно нужно найти самую большую клику графа, то единственным реальным решением будет исчерпывающий перебор с возвратом. Поиск выполняется во всех k-подмножествах вершин, и подмножество удаляется, как только в нем обна-

руживается вершина, не смежная со всеми остальными вершинами. Очевидной верхней границей размера максимальной клики графа G является наибольшая степень вершины, увеличенная на 1. Более точную верхнюю границу можно получить, отсортировав вершины в порядке убывания их степени. Пусть j будет наибольшим индексом, таким, что степень j-й вершины равна, по крайней мере, $j - 1$. Самая большая клика графа содержит не больше, чем j вершин, т. к. клика размером j не может содержать вершину со степенью меньшей, чем $(j - 1)$. Чтобы ускорить поиск, из графа G следует удалить все такие вершины.

Эвристические алгоритмы поиска больших клик на основе рандомизированных методов, таких как метод имитации отжига, должны работать достаточно хорошо.

Реализации. Набор процедур Cliquer на языке C, разработанный Патриком Остергардом (Patric Ostergard), предназначен для поиска клик в произвольно взвешенных графах. Процедуры основаны на алгоритме метода ветвей и границ (branch-and-bound algorithm). Загрузить код можно с веб-страницы **http://users.tkk.fi/~pat/cliquer.html**.

Второе соревнование по реализациям DIMACS было посвящено поиску клик и независимых множеств. Наборы данных и код можно загрузить по адресу **ftp://dimacs.rutgers.edu**. Исходные коды находятся в каталоге pub/challenge/graph/solvers, а тестовые данные — в каталоге pub/dsj. Решатель dfmax.c реализует простой алгоритм метода ветвей и границ, похожий на алгоритм, описанный в работе [CP90]. А решатель dmclique.c использует "полужадный" подход для поиска больших независимых множеств, описанный в работе [JAMS91].

В книге [KS99] рассматриваются программы на языке C на основе метода ветвей и границ для поиска максимальной клики, используя разные нижние границы. Загрузить программы можно с веб-страницы **http://www.math.mtu.edu/~kreher/cages/Src.html**.

Библиотека GOBLIN (**http://www.math.uni-augsburg.de/~fremuth/goblin.html**) содержит реализации алгоритмов метода ветвей и границ для поиска больших клик. Утверждается, что эти реализации могут обрабатывать большие графы, содержащие 150–200 вершин.

Примечания

Самый полный обзор задачи поиска максимальных клик приводится в [BBPP99]. Особый интерес представляет работа общества исследования операций по алгоритмам метода ветвей и границ для эффективного поиска клик. Более свежие результаты можно найти в [JS01].

Доказательство NP-полноты задачи о клике было предложено Карпом (Karp); см. [Kar72]. Выполнив необходимое сведение (см. *раздел 9.3.3*), он установил, что задачи о клике, вершинном покрытии и независимом множестве тесно связаны, поэтому эвристические алгоритмы, которые решают одну из этих задач, также должны дать удовлетворительные решения для двух других.

В задаче о подграфе с наибольшей плотностью требуется найти такое подмножество вершин, чтобы порождаемый ими подграф имел максимально возможную среднюю степень вершин. Клика из k вершин, очевидно, является самым плотным подграфом этого размера, но неполные подграфы большего размера могут иметь более высокую степень вершин. Эта задача является NP-полной, но простые эвристические методы на основе последовательного удаления вершин с самой низкой степенью выдают удовлетворительные прибли-

зительные решения (см. [AITT00]). Интересное применение задачи о самом плотном подграфе, а именно обнаружение ссылочного спама в Интернете, описывается в работе [GKT05].

В работе [Has82] было доказано, что клику нельзя аппроксимировать с точностью до $n^{1/2-\varepsilon}$, за исключением того случая, когда $P = NP$ (и с точностью до $n^{1-\varepsilon}$ при более слабых предположениях).

Родственные задачи. Независимое множество (см. *раздел 16.2*), вершинное покрытие (см. *раздел 16.3*).

16.2. Независимое множество

Вход. Граф $G = (V, E)$.

Задача. Найти такое наибольшее подмножество S множества вершин V, в котором для каждого ребра $(x, y) \in E$ верно, что либо $x \notin E$, либо $y \notin E$ (рис. 16.2).

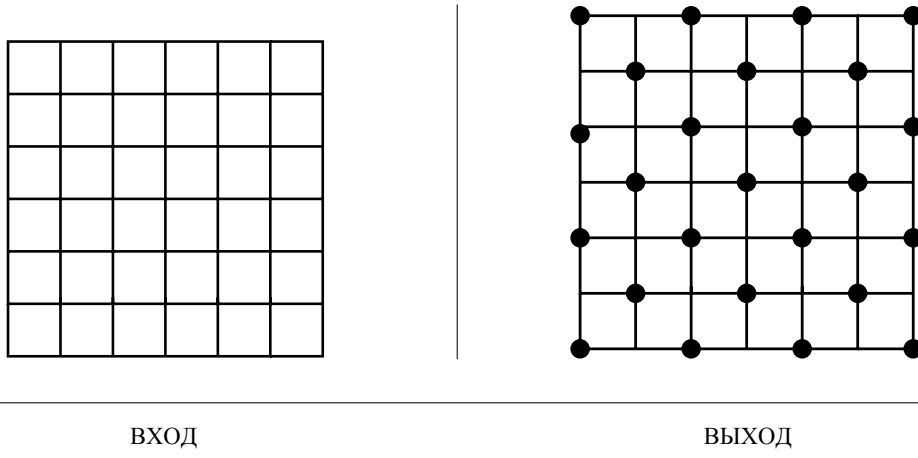


Рис. 16.2. Независимое множество

Обсуждение. Необходимость в поиске большого независимого множества вершин возникает в задаче выбора места для точек обслуживания. Важно не поместить два сервис-центра некоего предприятия слишком близко друг к другу, чтобы избежать конкуренции между ними. Для решения этой задачи мы можем создать граф, в котором вершины представляют возможные расположения точек обслуживания, а потом соединить ребрами пары точек, расположенных так близко, что они мешают друг другу. Максимальное независимое множество определяет наибольшее количество сервис-центров, которые мы можем расположить без внутренней конкуренции.

В независимых множествах (которые также называются *устойчивыми множествами*) отсутствуют конфликты между элементами, вследствие чего к ним часто прибегают в задачах теории кодирования и календарного планирования. Для использования независимого множества в теории кодирования определяем граф, вершины которого представляют набор возможных кодовых слов, и соединяем ребрами любые две вершины, кодовые слова которых похожи друг на друга настолько, что их можно перепутать при

шумовых помехах. Максимальное независимое множество этого графа определит код самой большой емкости для данного канала связи.

Задача о независимом множестве тесно связана с двумя другими NP-полными задачами:

- *с задачей о клике.* Дополнив независимое множество, мы получим клику. Дополнением графа $G = (V, E)$ является такой граф $G' = (V, E')$, для которого $(i, j) \in E'$ тогда и только, когда $(i, j) \notin E'$. Иными словами, мы убираем имеющиеся ребра и ставим ребра туда, где их не было. Максимальное независимое множество графа G представляет собой максимальную клику графа G', следовательно, эти две задачи алгоритмически идентичны. Таким образом, алгоритмы и реализации из *раздела 16.1* можно использовать и для решения задачи о независимом множестве;
- *с задачей вершинной раскраски.* Задача вершинной раскраски графа $G = (V, E)$ заключается в разбиении множества вершин V на подмножества таким образом, чтобы смежные вершины были разных цветов. Каждый цвет определяет независимое множество. Многие применения задачи о независимом множестве в задачах календарного планирования в действительности являются задачами раскраски, поскольку все работы, в конце концов, должны быть завершены.

  Один из эвристических подходов к поиску большого независимого множества состоит в применении любого алгоритма вершинной раскраски и использовании самого большого цветового класса. Отсюда следует, что все графы с небольшим количеством цветов (например, планарные или двудольные графы) имеют большие независимые множества.

Самый простой эвристический алгоритм поиска независимого множества можно описать так. Находим вершину с самой низкой степенью, добавляем ее в независимое множество, а потом удаляем эту и все смежные с ней вершины. Повторяя этот процесс, пока граф не окажется пустым, мы получим максимальное независимое множество, т. е. множество, которое нельзя увеличить простым добавлением вершин. Применение рандомизации или исчерпывающего перебора может дать независимые множества чуть большего размера.

Задача о независимом множестве в определенной степени является двойственной задаче паросочетания в графах. В первой задаче требуется найти большой набор вершин, не имеющих общих ребер, а во второй — большой набор ребер без общих вершин. Это наводит на мысль попробовать переформулировать NP-полную задачу о независимом множестве как задачу паросочетания, которая поддается эффективному решению.

Максимальное независимое множество для дерева можно найти за линейное время следующим способом: удаляются листья, удаленные листья добавляются к независимому множеству, удаляются все смежные узлы, а затем процедура повторяется для полученных деревьев до тех пор, пока дерево не станет пустым.

Реализации. Любую программу для вычисления максимальной клики в графе можно применить для поиска максимальных независимых множеств, просто дополнив граф. Поэтому все реализации, упомянутые в *разделе 16.1*, подходят и для решения задачи о независимом множестве.

Библиотека GOBLIN (**http://www.math.uni-augsburg.de/~fremuth/goblin.html**) содержит реализацию алгоритма метода ветвей и границ для поиска независимых множеств, которые в инструкции к программе называются устойчивыми множествами.

Алгоритм 787 на языке FORTRAN из коллекции алгоритмов ACM, реализованный Ресенде (Resende), представляет собой эвристическую процедуру GRASP для поиска независимого множества (см. [RFS98]). Эту реализацию можно загрузить с веб-сайта **http://www.research.att.com/~mgcr/src**.

Примечания

Доказательство NP-полноты задачи о независимом множестве было предоставлено Карпом (Karp); см. [Kar72]. Эта задача остается NP-полной для планарных кубических графов (см. [GJ79]). Для задачи о независимом множестве на двудольных графах существует эффективное решение (см. [Law76]). Сама задача не является тривиальной, поскольку бóльшая "часть" двудольного графа не обязательно представляет собой максимальное независимое множество.

Родственные задачи. Задача о клике (см. *раздел 16.1*), вершинная раскраска (см. *раздел 16.7*), вершинное покрытие (см. *раздел 16.3*).

16.3. Вершинное покрытие

Вход. Граф $G = (V, E)$.

Задача. Найти наименьшее подмножество $S \subset V$, для которого каждое ребро $(x, y) \in E$ содержит, по крайней мере, одну вершину из этого множества (рис. 16.3).

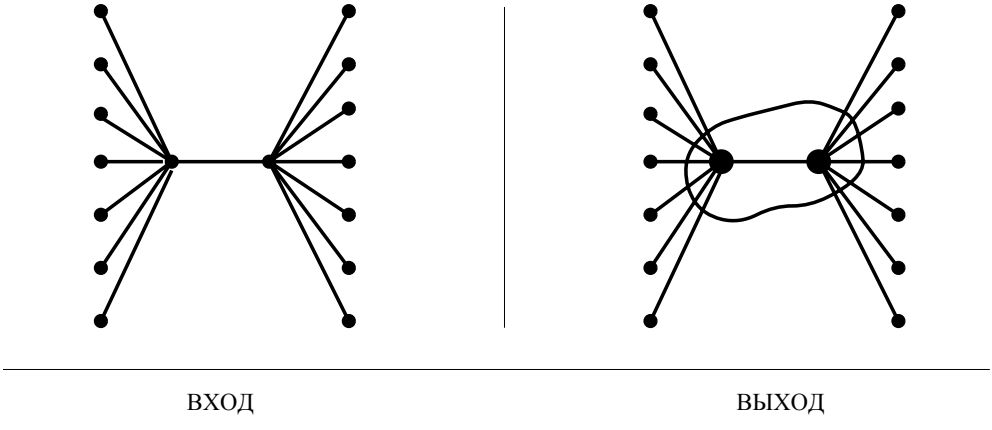


Рис. 16.3. Вершинное покрытие

Обсуждение. Задача о вершинном покрытии является частным случаем более общей задачи о покрытии множества, которая имеет на входе произвольную коллекцию подмножеств $S = (S_1, ..., S_n)$ универсального множества $U = \{1, ..., m\}$ и заключается в поиске наименьшего набора подмножеств из S, объединение которых будет равно U. Задача о вершинном покрытии возникает во многих приложениях, связанных с покупкой товаров, которые продаются в комплектах. Задача о покрытии множества обсуждается в *разделе 18.1*.

Преобразуем задачу о вершинном покрытии в задачу о покрытии множества. Пусть универсальное множество U представляет множество E ребер графа G, а множество S_i — набор ребер, инцидентных вершине i. Набор вершин определяет вершинное покрытие графа G тогда и только тогда, когда соответствующие подмножества определяют покрытие множества в данном экземпляре. Но так как каждое ребро может находиться только в двух разных подмножествах, то экземпляры задачи о вершинном покрытии проще, чем общая задача покрытия множества. Задача о вершинном покрытии является сравнительно простой среди NP-полных задач и поддается решению более эффективно, чем общая задача о покрытии множества.

Задачи о вершинном покрытии и независимом множестве тесно связаны друг с другом. Так как каждое ребро в множестве E по определению является инцидентным какой-либо вершине в любом покрытии S, то невозможно существование ребра, обе вершины которого находились бы в множестве (V–S). Таким образом, множество (V–S) должно быть независимым. Так как минимизация множества S равносильна максимизации множества (V–S), то эти две задачи эквивалентны. Это означает, что любая программа для решения задачи о независимом множестве может быть использована и для решения задачи о вершинном покрытии. Наличие двух способов подхода к решению задачи очень удобно, т. к. в некоторых контекстах один из них может оказаться проще.

Самый простой эвристический алгоритм для решения задачи о вершинном покрытии начинает работу с выбора вершины наивысшей степени, добавляет ее к покрытию, удаляет все смежные ребра, а потом повторяет процесс до тех пор, пока граф не станет пустым. На правильно построенных структурах данных этот алгоритм может быть выполнен за линейное время, и в большинстве случаев он выдает "довольно хорошее" покрытие. Но для некоторых входных экземпляров это покрытие может быть в $\lg n$ раз хуже, чем оптимальное.

К счастью, всегда можно найти вершинное покрытие, размер которого, самое большее, в два раза больше оптимального. Для этого находим *максимальное* паросочетание графа M, т. е. множество ребер, в котором ни одна пара ребер не имеет общей вершины и которое нельзя расширить, добавив к нему дополнительные ребра. Такое максимальное паросочетание можно создать инкрементальным образом: выбираем в графе произвольное ребро, удаляем все ребра, имеющие общую вершину с этим ребром, и повторяем процесс до тех пор, пока в графе не останется больше ребер. Взяв обе вершины каждого ребра в максимальном паросочетании, мы получим вершинное покрытие. Почему? Поскольку любое вершинное покрытие должно содержать хотя бы одну из двух вершин каждого ребра паросочетания только для того, чтобы покрыть ребра максимального паросочетания M, это покрытие больше минимального максимум в два раза.

Этот эвристический алгоритм можно настроить так, что он будет работать быстрее, если не в теории, то на практике. Мы можем выбирать ребра паросочетания, чтобы в конечном счете удалить как можно больше других ребер, тем самым уменьшив размер максимального паросочетания и, следовательно, количество пар вершин в вершинном покрытии. Кроме того, некоторые из вершин из паросочетания M в действительности могут оказаться ненужными, поскольку все инцидентные им ребра будут "охвачены" другими выбранными вершинами. Такие вершины можно идентифицировать и удалить, выполнив второй проход по полученному покрытию.

В задаче о вершинном покрытии требуется охватить все ребра, используя как можно меньшее количество вершин. Есть две другие задачи, в которых ставятся аналогичные цели:

- *задача о доминирующем множестве.* В этой задаче требуется найти такое наименьшее множество вершин D, для которого каждая вершина в подмножестве $(V–D)$ соединена ребром, по крайней мере, с одной вершиной в доминирующем множестве D. Каждое вершинное покрытие нетривиального связного графа также является и доминирующим множеством, но доминирующие множества могут быть намного меньшими, чем вершинные покрытия. Любая отдельная вершина представляет минимальное доминирующее множество полного графа K_n, в то время как для вершинного покрытия требуется $n-1$ вершина. Задача о доминирующем множестве обычно возникает в коммуникационных сетях, т. к. оно соответствует множеству узлов, которых достаточно для связи со всеми другими узлами.

  Задачи о доминирующем множестве можно с легкостью представить в виде экземпляров задачи о покрытии множества (см. *раздел 18.1*). Каждая вершина v_i определяет подмножество вершин, в которое входит она сама и все смежные с нею вершины. "Жадный" эвристический алгоритм находит аппроксимацию оптимального доминирующего множества для этого экземпляра за время $O(\lg n)$;

- *задача о реберном покрытии.* В этой задаче требуется найти наименьшее множество ребер такое, что каждая вершина графа инцидентна одному из ребер этого множества. Задачу эту можно эффективно решить, найдя паросочетание максимальной мощности (см. *раздел 15.6*), а потом выбрав произвольные ребра, чтобы учесть и вершины, не вошедшие в паросочетание.

Реализации. Любую программу поиска максимальной клики графа можно использовать для решения задачи о вершинном покрытии. Для этого нужно дополнить входной граф и выбрать вершины, которые не принадлежат клике. Поэтому мы отсылаем читателя к программам поиска клики, упомянутым в *разделе 16.1*.

Очень эффективным инструментом поиска вершинного покрытия является программа COVER (см. [RHG07]) на основе стохастического алгоритма локального поиска. Программу COVER можно загрузить с веб-сайта **http://www.nicta.com.au/people/richters/**.

Библиотека графов `JGraphT` на языке Java (**http://jgrapht.sourceforge.net**) содержит реализации "жадного" и 2-аппроксимирующего эвристических алгоритмов для поиска вершинного покрытия.

Примечания

Впервые NP-полнота задачи о вершинном покрытии была доказана Карпом (Karp); см. [Kar72]. Несколько эвристических методов, включая рандомизированное округление, составляют основу для 2-аппроксимирующих алгоритмов поиска вершинного покрытия. Хорошие описания этих 2-аппроксимирующих алгоритмов содержатся в работах [CLRS01], [Hoc96], [Pas97] и [Vaz04]. Пример "жадного" алгоритма, который может работать хуже оптимального в $\lg n$ раз, был впервые приведен в [Joh74] и изложен в книге [PS98]. Экспериментальные исследования эвристических методов для поиска вершинного покрытия описаны в [GMPV06], [GW97] и [RHG07].

Одна из важных нерешенных задач аппроксимирующих алгоритмов касается существования аппроксимирующего решения для задачи о вершинном покрытии, имеющего коэффициент лучший, чем 2. Хастад (Hastad) в своей работе [Has97] доказал, что для этой задачи не существует аппроксимирующего алгоритма, выдающего решение, имеющее коэффициент лучший, чем 1,1666.

Основным справочником по доминирующим множествам является монография [HHS98]. Эвристические методы решения задачи о связном доминирующем множестве рассматриваются в работе [GK98]. Невозможно получить аппроксимирующее решение задачи о доминирующем множестве, имеющее коэффициент лучший, чем $\Omega(\lg n)$ (см. [ACG+03]).

Родственные задачи. Независимое множество (см. *раздел 16.2*), вершинное покрытие (см. *раздел 16.3*).

16.4. Задача коммивояжера

Вход. Взвешенный граф G.

Задача. Найти цикл минимальной стоимости, проходящий через каждую вершину графа G ровно один раз (рис. 16.4).

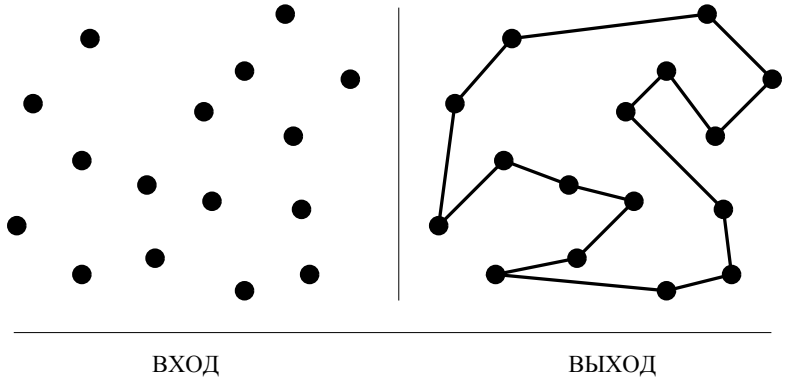


Рис. 16.4. Маршрут коммивояжера

Обсуждение. Задача коммивояжера — наиболее известная изо всех NP-полных задач. В качестве причин такой известности можно назвать прикладную ценность задачи и легкость ее популярного изложения. Представьте себе коммивояжера, планирующего поездку на автомобиле в несколько населенных пунктов. Он стремится разработать наиболее короткий маршрут, позволяющий посетить эти населенные пункты и возвратиться в исходную точку, т. е. минимизировать общую протяженность поездки?

Задача коммивояжера возникает во многих транспортных приложениях. Другим важным применением этой задачи является оптимизация перемещения инструментов на производстве. Рассмотрим, например, робот-манипулятор для пайки соединений на печатных платах. Наиболее эффективным маршрутом для такого манипулятора будет самый короткий маршрут, посещающий каждую точку пайки.

При решении задачи коммивояжера возникает несколько вопросов.

- *Является ли граф взвешенным?* Если граф маршрута является невзвешенным, или вес каждого ребра имеет одно из двух возможных значений, то задача сводится

к поиску гамильтонова цикла. Задача поиска гамильтонова цикла обсуждается в *разделе 16.5*.

- *Удовлетворяет ли вход аксиоме треугольника?* Наше интуитивное понятие о расстоянии основано на *аксиоме треугольника*. Эта аксиома утверждает, что $d(i, j) \leq d(i, k) + d(k, j)$ для всех вершин $i, j, k \in V$. Геометрические расстояния удовлетворяют аксиоме треугольника, т. к. кратчайший путь между двумя точками проходит по прямой линии между ними. Однако цены на коммерческие авиарейсы не удовлетворяют этой аксиоме, что и является причиной трудностей с выбором самого дешевого авиарейса между двумя точками. Эвристические методы решения задачи коммивояжера работают намного лучше с графами разумной степени сложности, удовлетворяющими аксиоме треугольника.
- *Входные данные представляют собой набор из n точек или взвешенный граф?* С геометрическими экземплярами часто легче работать, чем с графами. Так как пара точек определяет полный граф, то поиск подходящего маршрута не вызывает проблем. Вычисляя расстояния между точками лишь по мере надобности, можно сэкономить память, избавив себя от необходимости хранить матрицу расстояний размером $n \times n$. Геометрические экземпляры по своей сути удовлетворяют аксиоме треугольника, поэтому на них можно достичь производительности, гарантированной некоторыми эвристическими методами. Кроме прочего, можно воспользоваться такими геометрическими структурами данных, как kd-деревья, чтобы быстро найти непосещенные точки, расположенные близко друг к другу.
- *Разрешено ли многократное посещение одной и той же вершины?* Для многих приложений запрет на повторное посещение одной и той же вершины не имеет значения. Например, при авиаперелетах самый дешевый маршрут облета всех пунктов может проходить через один центральный аэропорт несколько раз. Обратите внимание, что этот вопрос не возникает, когда входной экземпляр удовлетворяет аксиоме треугольника.

  Задача коммивояжера, в которой допускается многократное посещение вершин, легко решается с помощью любой программы, предназначенной для решения общей версии этой задачи, на вход которой подается обновленная матрица стоимостей D, у которой ячейка $D(i, j)$ содержит кратчайшее расстояние от точки i к точке j. Эту матрицу можно создать, решив задачу поиска кратчайшего пути между всеми парами точек (см. *раздел 15.4*), и она удовлетворяет аксиоме треугольника.
- *Метрика расстояний симметрична?* Метрика расстояний асимметрична, когда существуют такие точки (x, y), для которых $d(x, y) \neq d(y, x)$. На практике асимметричную задачу коммивояжера решить намного труднее, чем симметричную. Старайтесь избегать таких необычных метрик расстояний. Имейте в виду, что существует способ преобразования асимметричных экземпляров задачи коммивояжера в симметричные, содержащие вдвое больше вершин (см. [GP07]). Вы можете воспользоваться им при решении вашей задачи, т. к. программы решения симметричных экземпляров работают намного лучше.
- *Насколько важно найти оптимальный маршрут?* В большинстве случаев достаточно решения, полученного эвристическим методом. Когда на первый план выходит оптимальность, можно применить любой из двух подходов. *Методы секущих*

плоскостей моделируют задачу в виде задачи целочисленного программирования, а потом решают ее соответствующими методами с ослабленными ограничивающими условиями. Если полученное оптимальное решение не является целочисленным, то добавляются дополнительные ограничивающие условия, чтобы обеспечить целочисленность решения. *Алгоритмы метода ветвей и границ* выполняют комбинаторный поиск, тщательно соблюдая при этом верхнюю и нижнюю границы стоимости маршрута. При умелом обращении такие алгоритмы могут решать задачи с тысячами вершин. Непрофессионалу для этого придется использовать самую лучшую существующую программу.

Почти любая версия задачи коммивояжера является NP-полной, поэтому правильным подходом к ее решению будет использование эвристического метода. Эти методы обычно выдают решение, отличающее от оптимального на несколько процентов, и этого обычно достаточно для практических целей. К сожалению, для задачи коммивояжера было предложено несколько десятков решений, и выбрать наиболее подходящее не так-то просто. Результаты экспериментов, излагаемые в литературе, несколько противоречивы. Но я рекомендую вам выбрать один из следующих эвристических подходов:

- *использование минимальных остовных деревьев.* Этот эвристический метод начинает с построения минимального остовного дерева, а потом выполняет обход в глубину полученного дерева. При этом каждое из $n-1$ ребер проходится ровно два раза: один раз при открытии вершин во время движения вниз и второй раз на обратном пути вверх. После этого определяется маршрут за счет упорядочивания вершин по времени их открытия. Если граф удовлетворяет аксиоме треугольника, то получившийся маршрут будет, самое большее, вдвое длиннее оптимального. Но на практике результат обычно еще лучше и превышает оптимальный на 15–20%. Кроме этого, время исполнения этого алгоритма ограничено временем исполнения для построения минимального остовного дерева, что в случае точек на плоскости равно $O(n\lg n)$ (см. *раздел 15.3*);
- *методы инкрементальной вставки.* Эвристические алгоритмы этого класса начинают работу с одной вершины, а потом вставляют новые точки в этот частичный маршрут по одной, пока маршрут не будет завершен. Из алгоритмов этого класса лучше всего, по-видимому, работает версия, основанная на вставке *самой дальней точки*: изо всех оставшихся точек в частичный маршрут T добавляется такая точка v, для которой:

$$\max_{v \in V} \min_{i=1}^{|T|} (d(v, v_i) + d(v, v_{i+1}))$$

  Условие "min" гарантирует, что мы вставляем вершину в позицию, которая добавляет к маршруту наименьшее расстояние, а "max" — что самая "худшая" из таких вершин выбирается первой. Этот подход работает потому, что сначала создается черновой маршрут и лишь потом уточняются его детали. Полученные таким образом маршруты обычно лишь на 5–10% длиннее оптимальных;
- *использование k-оптимальных маршрутов.* Значительно более мощными являются эвристические методы Кернигана-Лина (Kernighan-Lin heuristics). В этих методах сначала выбирается произвольный маршрут, в который потом вносятся локальные изменения с целью его улучшения. В частности, из маршрута удаляются подмноже-

ства, состоящие из k ребер, а оставшиеся после этого k цепочек снова соединяются, в расчете на получение более короткого маршрута. Маршрут называется k-оптимальным, когда из него больше нельзя удалить подмножество из k ребер и по-другому соединить оставшиеся цепочки. Один из эффективных способов улучшения результатов любого другого эвристического алгоритма задачи коммивояжера — выполнить над ним 2-оптимальное улучшение маршрута. В результате многочисленных экспериментов было установлено, что 3-оптимальные маршруты обычно отличаются от оптимальных лишь на несколько процентов. А для $k > 3$ время вычисления возрастает значительно быстрее, чем качество решения. Метод имитации отжига предоставляет альтернативный механизм обращения ребер для оптимизации решений, полученных эвристическими методами.

Реализации. Разработанная Эпплгейтом (Applegate), Биксби (Bixby), Чваталом (Chvatal) и Куком (Cook) (см. [ABCC07]) программа Concorde на языке ANSI C предназначена для решения симметричных задач коммивояжера и подобных задач оптимизации сетей. Эта программа установила несколько мировых рекордов и получила оптимальные решения для 106 из 110 экземпляров из библиотеки экземпляров задачи коммивояжера (TSPLIB, **http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/**), самый большой из которых содержит 15 112 узлов. Программу Concorde можно загрузить для академических исследований с веб-сайта **http://www.tsp.gatech.edu/concorde**. Это, бесспорно, самая лучшая реализация решения задачи коммивояжера. На веб-сайте разработчиков (**http://www.tsp.gatech.edu**) можно найти очень интересный материал по истории и практическим приложениям задачи коммивояжера.

Работа [LP07] Лоди (Lodi) и Пуннена (Punnen) является отличным обзором существующего программного обеспечения для решения задачи коммивояжера. Текущие ссылки на все упоминаемые в этом обзоре программы поддерживаются на веб-сайте **http://www.or.deis.unibo.it/research_pages/tspsoft.html**.

Библиотека TSPLIB содержит коллекцию стандартных сложных экземпляров задачи коммивояжера, которые возникают на практике. Лучшую версию библиотеки TSPLIB можно загрузить с веб-сайта **http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/**, хотя библиотека NetLib также содержит эти экземпляры.

Программа tsp_solve на языке C++, разработанная Чадом Гурвицем (Chad Hurwitz) и Робертом Крейгом (Robert Craig), предоставляет как эвристические, так и оптимальные решения. Программа может обрабатывать экземпляры, содержащие до 100 узлов. Загрузить ее можно по адресу **http://www.cs.sunysb.edu/~algorith** или запросить у Чада Гурвица по адресу **churritz@cts.com**. Библиотека GOBLIN (**http://www.math.uni-augsburg.de/~fremuth/goblin.html**) содержит реализации алгоритмов метода ветвей и границ для решения как симметричных, так и асимметричных версий задачи коммивояжера, а также реализации различных эвристических алгоритмов.

Алгоритм 608 на языке FORTRAN (см. [Wes83]) из коллекции алгоритмов ACM реализует эвристический метод для решения квадратичной задачи о назначениях, которая является более общей задачей, включающей в себя задачу коммивояжера, как частный случай. Алгоритм 750 на языке FORTRAN (см. [CDT95]) из этой же коллекции выдает точное решение для асимметричных экземпляров задачи коммивояжера. Подробности см. в *разделе 19.1.5*.

Примечания

В книге [ABCC07] Эпплгейта (Applegate), Биксби (Bixby), Чватала (Chvatal) и Кука (Cook) задокументированы методы, использованные авторами в их программах, решающих задачи коммивояжера за рекордное время, а также изложены теоретические основы задачи и ее история. Самым лучшим справочником по многочисленным версиям задачи коммивояжера является книга Гутина (Gutin) и Пуннена (Punnen) [GP07], заменившая старый и всеми любимый справочник [LLKS85].

Экспериментальные результаты применения эвристических методов для решения больших экземпляров задачи коммивояжера можно найти в [Ben92a], [GBDS80] и [Rei94]. Эти методы обычно позволяют получить решение, отличающееся от оптимального лишь на несколько процентов.

Эвристический алгоритм Кристофидеса (Christofides) (см. [Chr76]) представляет собой усовершенствованный вариант эвристического метода для построения минимального остовного дерева. На евклидовых графах он гарантирует маршрут, превышающий оптимальный не более чем в полтора раза. Время исполнения этого алгоритма равно $O(n^3)$, а узким местом является поиск совершенного паросочетания с минимальным весом (см. *раздел 15.6*). Эвристический алгоритм построения минимального остовного дерева был впервые изложен в работе [RSL77].

Аппроксимирующие методы для евклидовой задачи коммивояжера были разработаны Аророй (Arora) и Митчеллом (Mitchell) (см. [Aro98] и [Mit99]). Они выдают приблизительный результат за полиномиальное время с точностью до $1 + \varepsilon$ от оптимального для любого $\varepsilon > 0$. Эти методы представляют большой теоретический интерес, хотя их практическая ценность пока неизвестна.

История поиска оптимальных решений задачи коммивояжера свидетельствует о постоянном прогрессе в этой области. В 1954 г. Данциг (Dantzig), Фалкерсон (Fulkerson) и Джонсон (Johnson) решили экземпляр задачи коммивояжера для 42 городов Соединенных Штатов (см. [DFJ54]). В 1980 г. Падберг (Padberg) и Хонг (Hong) нашли решение экземпляра, состоящего из 318 вершин (см. [PH80]). А недавно Эпплгейт и др. (см. [ABCC07]) решили экземпляр задачи, размер которого в двадцать раз больше. Конечно, нельзя сбрасывать со счетов развитие аппаратного обеспечения, но значительная часть успехов объясняется появлением более удачных алгоритмов. Такой темп роста производительности свидетельствует о том, что при острой необходимости получение точных решений для NP-полных задач вполне возможно. К счастью или к сожалению, необходимость в них редко бывает острой.

Для больших экземпляров размер — не единственный критерий успешности решения. Можно с легкостью создать громадный граф, состоящий из одного дешевого цикла, для которого получить оптимальное решение не составит труда. Для множества точек, образующих выпуклый многоугольник на плоскости, минимальный маршрут коммивояжера определяется их выпуклой оболочкой (см. *раздел 17.2*), которую можно вычислить за время $O(n\lg n)$. Известны также другие простые частные случаи этой задачи.

Родственные задачи. Гамильтонов цикл (см. *раздел 16.5*), минимальное остовное дерево (см. *раздел 15.3*), выпуклая оболочка (см. *раздел 17.2*).

16.5. Гамильтонов цикл

Вход. Граф $G = (V, E)$.

Задача. Найти маршрут, состоящий из ребер графа, такой что каждая вершина посещается ровно один раз (рис. 16.5).

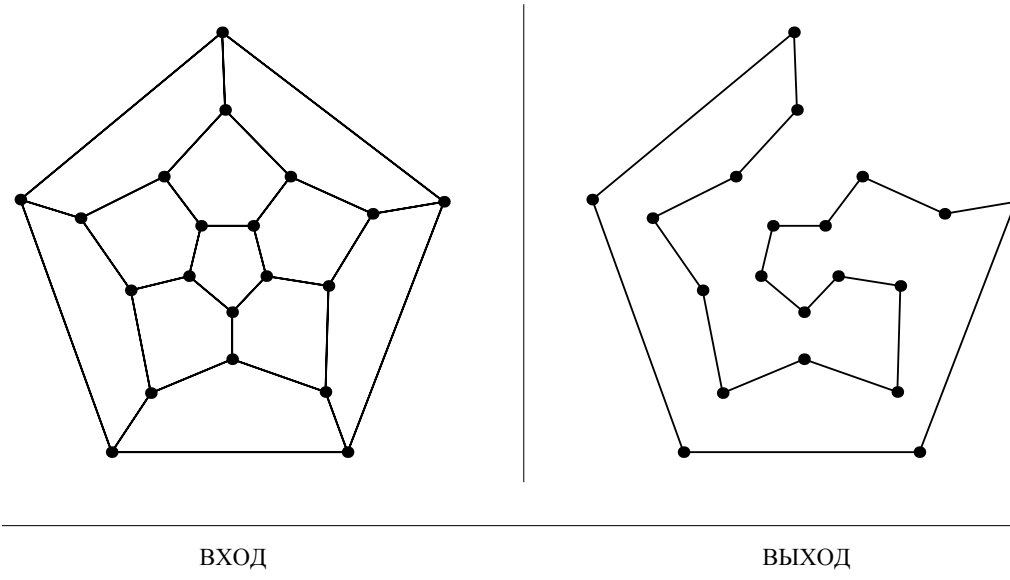


Рис. 16.5. Гамильтонов цикл

Обсуждение. Задача поиска гамильтонова цикла или пути в графе G является частным случаем задачи коммивояжера для графа G' с тем условием, что длина каждого ребра графа G равна 1 в графе G'. Расстоянию между вершинами, не соединенными ребрами, присваивается большее значение, например, 2. Такой взвешенный граф содержит маршрут коммивояжера стоимостью n в графе G' тогда и только тогда, когда граф G является гамильтоновым.

С задачей поиска гамильтонова пути тесно связана задача поиска самого длинного пути или цикла в графах, необходимость в котором часто возникает в задачах распознавания образов. В таких приложениях вершины графа соответствуют допустимым символам, а ребра соединяют пары символов, которые могут быть соседними. Самый длинный путь в этом графе будет наилучшим кандидатом для правильной интерпретации.

Обе задачи (поиска самого длинного пути и цикла) являются NP-полными, даже на невзвешенных графах. Тем не менее, существует несколько подходов к их решению, и для выбора наилучшего вы должны найти ответы на следующие вопросы.

- *Назначаются ли крупные штрафы за посещение вершин более одного раза?* Если мы переформулируем задачу поиска гамильтонова цикла вместо того, чтобы пытаться минимизировать общее количество вершин, посещенных при полном маршруте, то получим задачу оптимизации, что позволит использовать для ее решения эвристические и аппроксимирующие алгоритмы. Построив остовное дерево графа и выполнив по нему обход в глубину, как показано в *разделе 16.4*, мы получим маршрут, содержащий, самое большее, $2n$ вершин. Использование рандомизации или метода имитации отжига может значительно уменьшить этот число.
- *Является ли бесконтурным орграфом граф, в котором осуществляется поиск самого длинного пути?* Задачу поиска самого длинного пути в бесконтурном орграфе можно решить за линейное время с помощью методов динамического программи-

рования. К счастью, для этого можно использовать алгоритм поиска кратчайшего пути в бесконтурном орграфе (представленный в *разделе 15.4*), просто заменив *min* на *max*. Бесконтурные орграфы являются наиболее интересными экземплярами задачи поиска самого длинного пути, для которых существуют эффективные алгоритмы.

- *Является ли граф плотным?* Достаточно плотные графы всегда содержат гамильтоновы циклы. Более того, циклы на таких графах можно строить достаточно эффективно. В частности, любой граф, все вершины которого имеют степень, большую или равную $n/2$, должен быть гамильтоновым. Существуют и другие условия достаточности для наличия гамильтоновых циклов; подробности см. в *подразделе "Примечания"*.
- *Нужно ли посетить все вершины или требуется пройти по всем ребрам?* Убедитесь в том, что перед вами действительно стоит задача посещения вершин, а не ребер. При должной изобретательности иногда удается переформулировать задачу поиска гамильтонова цикла в терминах задачи поиска эйлерова цикла, в которой требуется пройти по каждому ребру графа. Пожалуй, самым известным таким экземпляром является задача создания последовательностей де Брейна, рассматриваемая в *разделе 15.7*. Польза от такой переформулировки состоит в том, что для решения задачи поиска эйлерова цикла и многих родственных ей задач существуют быстрые алгоритмы, в то время как задача поиска гамильтонова цикла является NP-полной.

Если вам действительно нужно знать, является ли ваш граф гамильтоновым, то единственным решением этой задачи является поиск с возвратом и разрежением поискового пространства. Обязательно проверьте граф на двусвязность (см. *раздел 15.8*). Если граф не двусвязный, то это означает, что в нем есть шарнир, удаление которого разъединит граф, и поэтому граф не может быть гамильтоновым.

Реализации. Описанная ранее конструкция (в которой вес ребер равен 1, а расстояние между несмежными вершинами равно 2) сводит задачу поиска гамильтонова цикла к симметричной задаче коммивояжера, удовлетворяющей аксиоме треугольника. Поэтому для решения задачи поиска гамильтонова цикла можно использовать программы для решения задачи коммивояжера, рассмотренные в *разделе 16.4*. Лучшей из них является программа Concorde, написанная на языке ANSI C и предназначенная для решения симметрических задач коммивояжера и подобных задач оптимизации сетей. Программу Concorde можно загрузить для академических исследований с веб-сайта **http://www.tsp.gatech.edu/concorde**. Это, бесспорно, самая лучшая реализация решения задачи коммивояжера.

Эффективная программа для решения задач поиска гамильтонова цикла была разработана на основе диссертации Вандергринда (Vandegriend); см. [Van98]. Код программы и текст работы можно загрузить с веб-сайта **http://webdocs.cs.ualberta.ca/~joe/Theses/vandegriend.html**.

Лоди (Lodi) и Пуннен (Punnen) написали прекрасный обзор существующего программного обеспечения для решения задачи коммивояжера, включая частный случай гамильтонова цикла (см. [LP07]). Действующие ссылки на все упоминаемые в этом обзоре программы поддерживаются на веб-сайте **http://www.or.deis.unibo.it/research_pages/tspsoft.html**.

Программа для ранжирования футбольных команд (речь идет об американском футболе — *прим. перев.*) из базы графов Standford GraphBase (см. *раздел 19.1.8*) использует многоуровневый "жадный" алгоритм для решения асимметричной задачи поиска самого длинного пути. Целью задачи является получение цепочки результатов футбольных матчей, чтобы установить превосходство одной футбольной команды над другой. Ведь если команда университета Вирджинии победила команду университета Иллинойса с разрывом в 30 очков, а команда университета Иллинойса победила команду университета Стоуни Брук с разрывом в 14 очков, тогда, по идее, можно считать, что если бы команда университета Вирджинии играла с командой университета Стоуни Брук, то она победила бы ее с разрывом в 44 очка, не так ли? Мы хотим найти самый длинный простой путь в графе, где вес ребра (x, y) обозначает превосходство в счете выигравшей команды x над проигравшей y.

Эффективная программа перечисления всех гамильтоновых циклов графа методом поиска с возвратом представлена в книге [NW78]. Подробности см. в *разделе 19.1.10*. Алгоритм 595 (см. [Mar83]) на языке FORTRAN из коллекции алгоритмов ACM представляет собой аналогичную программу, которую можно использовать для получения точного или приблизительного решения, управляя количеством возвратов. Подробности см. в *разделе 19.1.5*.

Примечания

По-видимому, задача поиска гамильтонова цикла впервые возникла в процессе изучения Эйлером задачи о ходе шахматного коня, хотя они были популяризированы в 1839 г. в головоломке "Вокруг света", изобретенной Гамильтоном. Обширный справочный материал по задаче коммивояжера, включающий в себя обсуждение задачи поиска гамильтонова цикла, представлен в [ABCC07], [GP07] и [LLKS85].

В большинстве учебников по теории графов обсуждаются условия достаточности для наличия гамильтоновых циклов. Моим любимым является учебник [Wes00].

В последнее время большое внимание привлекают к себе методы оптимизации лабораторных биологических процессов. При первом применении этих методов "биовычислений" Аделман (Adleman) (см. [Adl94]) решил задачу поиска гамильтонова пути на экземпляре с семью вершинами. К сожалению, этот подход требует экспоненциального количества молекул, и, зная число Авогадро, можно утверждать, что такие эксперименты невозможны для графов с количеством вершин свыше $n \approx 70$.

Родственные задачи. Эйлеров цикл (см. *раздел 15.7*), задача коммивояжера (см. *раздел 16.4*).

16.6. Разбиение графов

Вход. Граф $G = (V, E)$ (взвешенный) и целые числа k и m.

Задача. Разбить множество вершин графа на m приблизительно равных подмножеств таким образом, чтобы общая стоимость ребер во всех подмножествах не превышала k (рис. 16.6).

Обсуждение. Задача разбиения графа возникает во многих алгоритмах типа "разделяй и властвуй", эффективность которых основана на разбиении задачи на несколько меньших подзадач, с последующей "сборкой" решения из решений этих подзадач.

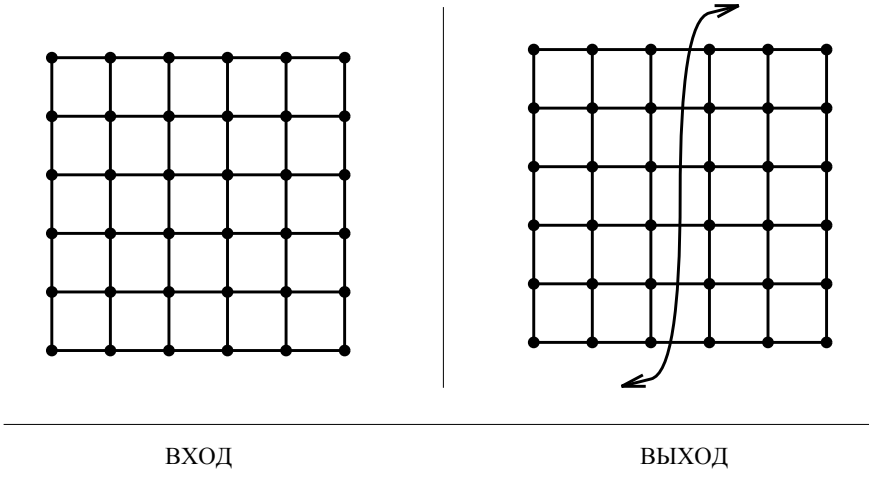


Рис. 16.6. Разбиение графа

Стремление к минимизации количества ребер, удаленных при разбиении графа, обычно упрощает задачу слияния решений подзадач в одно общее решение.

Задача разбиения графов также возникает, когда нужно собрать вершины в кластеры логических компонентов. Если ребра соединяют сходные объекты, то оставшиеся после разбиения кластеры должны отражать логически связанные группы. Большие графы часто разбиваются на части приемлемого размера, чтобы облегчить обработку данных или чтобы получить менее загроможденный рисунок.

Наконец, разбиение графов является важным шагом во многих параллельных алгоритмах. В качестве примера можно назвать метод конечных элементов, который используется для расчета физических явлений (таких как механическое напряжение или теплопередача) в геометрических моделях. Параллелизация этих вычислений требует разбиения моделей на равные части с небольшой общей границей. Это задача разбиения графа, т. к. топология геометрической модели обычно представляется с помощью графа.

В зависимости от требуемой целевой функции, могут возникать разные виды задачи разбиения графа. А именно:

- *минимальный разрез.* Наименьшее множество ребер, удаление которых разделит граф на части, можно эффективно найти, используя метод потока в сети или рандомизированные алгоритмы. Дополнительную информацию по алгоритмам связности см. в *разделе 15.8*. Но наименьший разрез может отделить только одну вершину, вследствие чего получившееся разбиение может оказаться очень несбалансированным;
- *разбиение графа.* При более практичном критерии разбиения мы ищем разрез небольшого размера, который делит граф на приблизительно равные части. К сожалению, эта задача является NP-полной. Однако на практике она хорошо поддается решению эвристическими методами, рассматриваемыми далее.

  Некоторые специализированные графы обязательно имеют небольшие множества вершин-разделителей, которые разбивают граф на сбалансированные части. Любое

дерево всегда имеет одну вершину, удаление которой разбивает дерево на части так, что ни одна из них не содержит больше, чем $n/2$ первоначальных n вершин. Эти компоненты не обязательно должны быть связными; рассмотрим, хотя бы, разделяющую вершину звездообразного дерева. Эту разделяющую вершину можно найти за линейное время посредством поиска в глубину. Каждый планарный граф имеет множество из $O(\sqrt{n})$ вершин, удаление которых не оставляет компонент, содержащих более $2n/3$ вершин. Наличие разделителей обеспечивает удобный способ разложения геометрических моделей, которые, как правило, часто представляются планарными графами;

- *максимальный разрез.* Для графов, представляющих электронные схемы, *максимальный разрез* графа (рис. 16.7) соответствует максимальному потоку данных, который может протекать в схеме. Таким образом, канал связи с самой высокой пропускной способностью должен покрывать разбиение множества вершин, определяемое максимальным разрезом. Задача поиска максимального разреза графа является NP-полной (см. [Kar72]), но на практике хорошо поддается решению эвристическими методами, аналогичными методам разбиения графа.

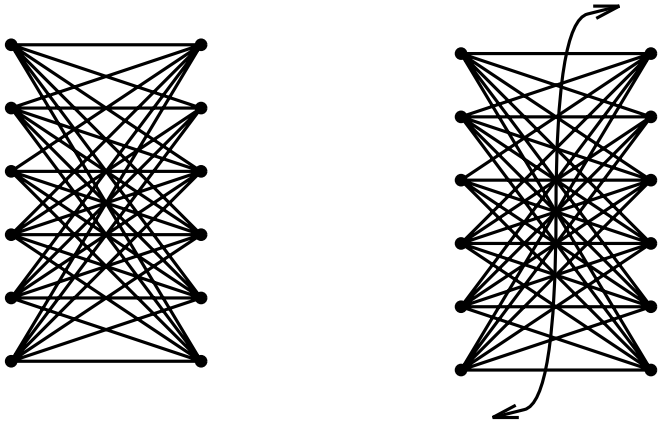


Рис. 16.7. Максимальный разрез графа

Основной подход к решению задач разбиения графа и поиска максимального разреза состоит в создании первоначального разбиения множества вершин (либо произвольным образом, либо в соответствии с некоторой стратегией, специфичной для приложения) с последующим перебором всех вершин и принятием решения по поводу каждой из них, не увеличится ли размер разреза, если ее переместить в другую часть графа. Решение о перемещении вершины v можно принять за время, пропорциональное ее степени, выяснив, какая часть разбиения содержит больше ее соседей. Конечно, наиболее подходящая часть для расположения вершины v может измениться после перемещения ее соседей, поэтому, скорее всего, понадобится несколько итераций, прежде чем процесс достигнет локального оптимума. Но даже в этом случае локальный оптимум может быть сколь угодно далек от глобального максимального разреза.

Существует много вариантов этой базовой процедуры, отличающихся от описанного порядком рассмотрения вершин или тем, что в перемещении участвуют целые класте-

ры вершин. Почти наверняка даст хороший результат использование какого-либо вида рандомизации, особенно метода имитации отжига. Если вершины нужно разбить на более чем две части, то процедуру разбиения следует применять рекурсивно.

В *спектральных* методах разбиения используются сложные приемы линейной алгебры. В частности, для разбиения матрицы на две части методом спектральной бисекции используется второй снизу собственный вектор *матрицы Лапласа* для графа. Спектральные методы обычно позволяют эффективно находить общую область для разбиения, но результаты можно улучшить, обработав полученный результат методом локальной оптимизации.

Реализации. Программа Chaco широко применяется для разбиения графов в приложениях параллельных вычислений. В ней используется несколько разных алгоритмов разбиения, включая метод Кернигана-Лина и спектральный метод. Программу Chaco можно загрузить с веб-сайта **http://www.cs.sandia.gov/~bahendr/chaco.html**.

Хорошей репутацией пользуется пакет METIS, который можно загрузить с веб-сайта **http://glaros.dtc.umn.edu/gkhome/views/metis**. Эта программа успешно применялась для разбиения графов, имеющих более миллиона вершин. Одна из доступных версий программы предназначена для выполнения на многопроцессорных компьютерах, а вторая для разбиения гиперграфов. Также заслуживают внимания программы Scotch (**http://www.labri.fr/perso/pelegrin/scotch**) и JOSTLE (**http://staffweb.cms.gre.ac.uk/~wc06/jostle/**).

Примечания

Основные эвристические методы для локального улучшения разбиения графа были представлены в [KL70] и [FM82]. Спектральные методы разбиения графов рассмотрены в [Chu97] и [PSL90]. Эмпирические результаты по эвристическим методам для разбиения графов изложены в [BG95] и [LR93].

Теорема о планарном разделителе и эффективный алгоритм поиска такого разделителя были представлены Липтоном и Тарьяном (см. [LT79] и [LT80]). Опыт реализации алгоритмов поиска разделителей планарных графов изложен в работах [ADGM04] и [HPS+05].

Можно ожидать, что любое случайное разбиение множества вершин приведет к удалению половины ребер графа, т. к. вероятность того, что две вершины, имеющие общее ребро, окажутся в разных частях разбиения, равна 1/2. Гоманс (Goemans) и Вильямсон (Williamson) в [GW95] представили аппроксимирующий алгоритм, основанный на методе полуопределенного программирования, выдающий решение задачи максимального разреза с точностью до 0,878. Более точный анализ этого алгоритма был выполнен Карловым (Karloff), см. [Kar96].

Родственные задачи. Реберная и вершинная связности (см. *раздел 15.8*), поток в сети (см. *раздел 15.9*).

16.7. Вершинная раскраска

Вход. Граф $G = (V, E)$.

Задача. Раскрасить множество вершин V, используя для этого минимальное количество цветов, таким образом, чтобы вершины i и j были разного цвета для всех ребер $(i, j) \in E$ (рис. 16.8).

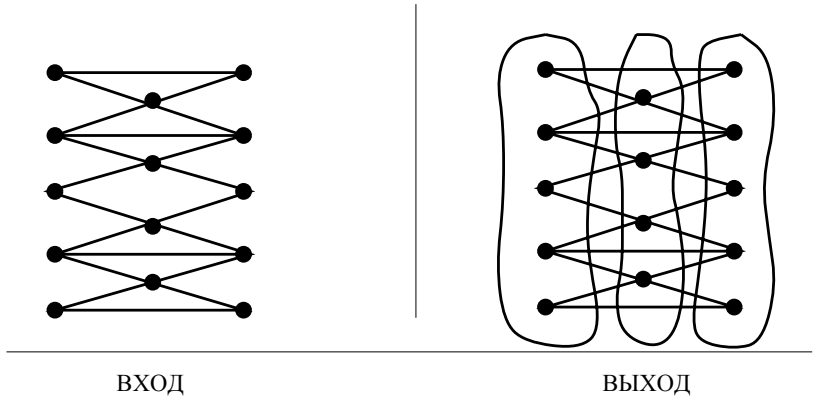


Рис. 16.8. Вершинная раскраска

Обсуждение. Задача вершинной раскраски возникает во многих приложениях календарного планирования и кластеризации. Классическим приложением раскраски графа является распределение регистров при оптимизации программы в процессе компилирования. Для каждой переменной в данном фрагменте программы существуют интервалы времени, в течение которых ее значение не должно меняться, в частности, между ее инициализацией и следующим обращением к ней. Следовательно, нельзя помещать в один и тот же регистр две переменные с пересекающимися интервалами времени. Для оптимального распределения регистров создаем граф, в котором вершины соответствуют переменным, и соединяем ребрами каждые две вершины, у которых переменные имеют пересекающиеся интервалы времени. Поскольку никакие две переменные с вершинами одного цвета не конфликтуют между собой, им можно выделить один и тот же регистр.

Никаких конфликтов не будет, если каждая вершина окрашена в свой цвет. Так как количество регистров процессора ограничено, то мы стремимся выполнить раскраску, используя как можно меньшее количество цветов. Минимальное количество цветов, достаточное для раскраски вершин графа, называется его хроматическим числом.

На практике встречается несколько частных случаев задачи раскраски, представляющих интерес. Для их распознавания постарайтесь найти ответы на следующие вопросы.

- *Можно ли раскрасить граф, используя только два цвета?* Важным частным случаем является проверка графа на *двудольность*, означающую, что его можно раскрасить, используя только два разных цвета. Двудольные графы естественно возникают в таких приложениях, как назначение заданий работникам. Быстрые и простые алгоритмы существуют для задач паросочетания (см. *раздел 15.6*), если на них наложено такое ограничение, как двудольность графа.

  Проверка графа на двудольность не составляет труда. Окрашиваем первую вершину в синий цвет, а потом выполняем обход графа в глубину. При открытии каждой новой, еще не окрашенной вершины, окрашиваем ее в цвет, противоположный цвету смежной с ней вершины, т. к. окраска в тот же самый цвет вызвала бы конфликт. Граф не может быть двудольным, если он содержит ребро (x, y), обе вершины которого окрашены в один цвет. В противном случае конечная раскраска будет двуцвет-

ной, и раскрашивание займет время $O(n + m)$. Реализацию этого алгоритма см. в *разделе 5.7.2*.

- *Является ли граф планарным? Все ли вершины графа имеют низкую степень?* Знаменитая теорема о четырех красках утверждает, что вершинную раскраску любого планарного графа можно выполнить, используя, самое большее, четыре разных цвета. Для четырехцветной раскраски планарных графов существуют эффективные алгоритмы, но выяснение, можно ли данный планарный граф раскрасить тремя цветами, является NP-полной задачей.

  Для шестицветной раскраски планарного графа существует очень простой алгоритм. Любой планарный граф содержит вершину, степень которой не больше пяти. Удаляем эту вершину и раскрашиваем граф рекурсивно. Удаленная вершина имеет не более пяти соседей, следовательно, ее всегда можно раскрасить одним из шести цветов, отличающимся от цвета соседних вершин. Метод работает, т. к. после удаления вершины из планарного графа получается планарный граф, и у него тоже должна быть вершина низкой степени, которую можно будет удалить. Эту же идею можно использовать для раскраски любого графа максимальной степени Δ, задействовав не больше $\Delta + 1$ цветов, за время $O(n\Delta)$.

- *Можно ли привести задачу к задаче реберной раскраски?* Некоторые задачи вершинной раскраски можно сформулировать в виде задачи *реберной раскраски*, в которой требуется раскрасить ребра графа таким образом, чтобы ни одна пара ребер с общей вершиной не была окрашена одинаковым цветом. Практическая выгода от такой переформулировки состоит в том, что для решения задачи реберной раскраски существует эффективный алгоритм, который возвращает почти оптимальную раскраску. Алгоритмы реберной раскраски рассматриваются в *разделе 16.8*.

Задача вычисления хроматического числа графа является NP-полной, поэтому, если требуется точное решение, нужно будет использовать метод перебора с возвратом, который может оказаться неожиданно эффективным при раскраске некоторых графов. Но задача поиска приблизительного решения, достаточно близкого к оптимальному, остается сложной, поэтому не следует ожидать никаких гарантий.

Наилучшими эвристическими алгоритмами для вершинной раскраски являются инкрементальные. Так же, как и в ранее упомянутом алгоритме для планарных графов, раскраска вершин выполняется последовательно, при этом цвета для раскраски текущей вершины выбираются в зависимости от уже использованных цветов для раскраски смежных вершин. Эти алгоритмы различаются способами выбора следующей вершины и выбора цвета для ее раскраски. Практика подсказывает, что вершины следует вставлять в порядке невозрастания их степеней. Дело в том, что для вершин с высокой степенью существует больше ограничений по выбору цвета, и вставлять их надо как можно раньше, чтобы не потребовался дополнительный цвет. Эвристический алгоритм Брелаза (Brelaz) (см. [Bre79]) динамически выбирает неокрашенную вершину, у которой смежные вершины окрашены в наибольшее количество разных цветов, и окрашивает ее неиспользованным цветом с наименьшим номером.

Полученные инкрементальными методами результаты можно улучшить, используя *обмен цветов*. Для этого в раскрашенном графе меняем местами два цвета, т. е., например, красные вершины раскрашиваем синим цветом, а синие — красным, сохраняя при

этом правильную вершинную раскраску. Теперь возьмем раскрашенный должным образом граф и удалим из него все вершины, кроме красных и синих. Мы можем перекрасить одну или несколько из получившихся компонент связности, сохраняя при этом правильную раскраску. После такого перекрашивания может оказаться, что какая-либо вершина, ранее смежная как с красными, так и с синими вершинами, является смежной только с синими вершинами, что позволит окрасить ее красным цветом.

Обмен цветов позволяет получить более качественную раскраску за счет увеличения времени обработки и сложности реализации. Алгоритмы, основанные на методе имитации отжига и использующие обмен цветов для перехода из одного состояния в другое, будут, скорее всего, еще более эффективными.

Реализации. Для раскраски графов есть два полезных интернет-ресурса. Страница раскраски графов (**http://web.cs.ualberta.ca/~joe/Coloring**) предоставляет обширную библиографию и программное обеспечение для генерирования и решения сложных экземпляров задачи раскраски графов. Страница Майкла Трика (Michael Trick) **http://mat.gsia.cmu.edu/COLOR/color.html** содержит хороший обзор приложений для раскраски графов, аннотированную библиографию, а также коллекцию из более чем 70 экземпляров задачи раскраски графов, возникающих в таких приложениях, как распределение регистров и проверка печатных плат. Оба веб-сайта также содержат реализацию на языке C алгоритма раскрашивания DSATUR.

На втором соревновании по реализации алгоритмов DIMACS в октябре 1993 г. (см. [JT96]) оценивались программы решения тесно связанных задач поиска клик и раскраски вершин графов. Программы и данные можно загрузить по адресу **ftp://dimacs.rutgers.edu**. Исходные коды находятся в каталоге pub/challenge/graph, а тестовые данные — в каталоге pub/djs, включая простой "полуисчерпывающий жадный" подход, используемый в алгоритме раскраски графов XRLF (см. [JAMS91]).

Программа GraphCol (**http://code.google.com/p/graphcol/**), написанная на языке C, реализует алгоритм раскраски графов, основанный на методах поиска с запретами и имитации отжига.

Библиотека Boost Graph Library (см. [SLL02]; **http://www.boost.org/libs/graph/doc**) содержит реализацию на языке C++ "жадного" инкрементального эвристического алгоритма для раскраски вершин графов. Библиотека GOBLIN (**http://www.math.uni-augsburg.de/~fremuth/goblin.html**) содержит реализацию алгоритма метода ветвей и границ для вершинной раскраски.

В книге [SDK83] представлены реализации на языке Pascal алгоритмов раскраски графов, основанных на методе перебора с возвратом, а также нескольких эвристических методов, включая метод инкрементального упорядочения по принципу "наибольшего в начало" и "наименьшего в конец" и метод обмена цветов. Подробности см. в *разделе 19.1.10*.

В книге [NW78] приводятся эффективные реализации на языке FORTRAN алгоритмов вычисления хроматических многочленов и раскраски графов, основанных на методе перебора с возвратом. Подробности см. в *разделе 19.1.10*.

Библиотека Combinatorica содержит реализации (на языке пакета Mathematica) алгоритмов для проверки двудольности графов, раскраски с помощью эвристических мето-

дов, вычисления хроматических многочленов и вершинной раскраски методом перебора с возвратом. Подробности см. в *разделе 19.1.9*.

Примечания

Отличным источником информации по эвристическим методам, включающим результаты экспериментов, является уже не новая книга [SDK83]. Классические эвристические методы для вершинной раскраски приводятся в работах [Bre79], [MMI72] и [Tur88]; последние результаты можно найти в [GH06] и [HDD03].

В своей работе [Wil84] Вильф (Wilf) доказал, что алгоритм для проверки произвольного графа на то, что его хроматическое число равно k, работающий методом перебора с возвратом, выполняется за постоянное время, зависящее от k, но не зависящее от n. Впрочем, это не представляет большого интереса, поскольку в действительности очень мало графов являются k-раскрашиваемыми. Известно несколько эффективных (но, тем не менее, имеющих экспоненциальное время исполнения) алгоритмов для вершинной раскраски. Обзор таких алгоритмов представлен в [Woe03].

Работа [Pas03] содержит информацию о доказуемо хороших аппроксимирующих алгоритмах вершинной раскраски. С одной стороны, сложность задачи получения аппроксимирующего решения с полиномиальным коэффициентом была доказана в работе [BGS95]. Но с другой стороны, существуют эвристические методы, гарантирующие нетривиальные результаты в зависимости от значений различных параметров. Одним из таких методов является алгоритм Вигдерсона (Wigderson) (см. [Wig83]), предоставляющий аппроксимирующее решение, имеющее коэффициент $n^{1-1/(\chi(G)-1)}$, где $\chi(G)$ является хроматическим числом графа G.

Теорема Брука утверждает, что для хроматического числа справедливо $\chi(G) \leq \Delta(G) + 1$, где $\Delta(G)$ — максимальная степень вершин графа G. Равенство имеет место только для циклов нечетной длины (с хроматическим числом 3) и полных графов.

Самой знаменитой задачей в истории теории графов является задача четырехцветной раскраски. Она была впервые поставлена в 1852 г. и окончательно решена в 1976 г. Аппелем (Appel) и Хакеном (Haken), причем для доказательства правильности решения был использован компьютер. Любой планарный граф можно раскрасить пятью цветами, применяя один из вариантов эвристического метода обмена цветов. Несмотря на существование решения задачи четырехцветной раскраски, задача проверки достаточности трех цветов для раскраски конкретного планарного графа является NP-полной. История задачи четырехцветной раскраски и доказательство изложены в книге [SK86]. Эффективный алгоритм для четырехцветной раскраски графа представлен в работе [RSST96].

Родственные задачи. Независимое множество (см. *раздел 16.2*), реберная раскраска (см. *раздел 16.8*).

16.8. Реберная раскраска

Вход. Граф $G = (V, E)$.

Задача. Найти наименьший набор цветов, требуемый для раскраски ребер графа G таким образом, чтобы ни одна пара ребер, имеющих общую вершину, не была окрашена одним цветом (рис. 16.9).

Обсуждение. Задача реберной раскраски графов возникает в приложениях календарного планирования, обычно, когда требуется минимизировать количество взаимно не

конфликтующих маршрутов, необходимых для выполнения данного набора заданий. Рассмотрим для примера ситуацию, когда необходимо составить расписание нескольких деловых встреч продолжительностью в один час, в каждой из которых участвуют два сотрудника. Чтобы избежать конфликтов, все встречи можно было бы запланировать на разное время, но разумнее назначить неконфликтующие встречи на одно и то же время. Для решения этой задачи создадим граф, в котором вершины представляют людей, а ребра соединяют тех, которые должны встретиться. Реберная раскраска этого графа определяет искомое расписание. Разные цвета представляют разные временные периоды, при этом все встречи одного цвета происходят одновременно.

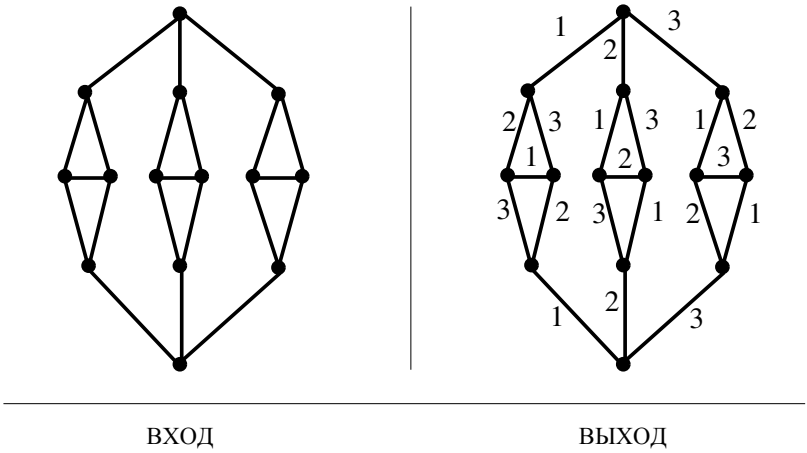


Рис. 16.9. Реберная раскраска

Национальная футбольная лига решает такую задачу реберной раскраски каждый сезон, чтобы составить расписание игр входящих в нее команд. Противники каждой команды определяются по результатам игр предыдущего сезона. Составление расписания игр является задачей реберной раскраски, усложненной дополнительными ограничивающими условиями, такими как необходимость в ответных матчах и традиционное требование встречи сильных соперников в понедельник вечером.

Минимальное количество цветов, требуемое для реберной раскраски графа, называется *реберно-хроматическим числом* (edge-chromatic number) или *хроматическим индексом* (chromatic index). Обратите внимание, что ребра цикла четной длины можно раскрасить двумя цветами, в то время как реберно-хроматическое число циклов нечетной длины равно трем.

С задачей реберной раскраски связана интересная теорема. Согласно теореме Визинга, для любого графа с максимальной степенью вершин Δ можно выполнить реберную раскраску, используя, самое большее, $\Delta + 1$ цвет. Чтобы понять это утверждение, обратите внимание на то, что любая реберная раскраска должна содержать, как минимум, Δ цветов, т. к. все ребра, инцидентные одной и той же вершине, должны быть раскрашены разными цвётами.

Доказательство теоремы Визинга является конструктивным, т. е. его можно представить в виде алгоритма поиска реберной раскраски из $\Delta + 1$ цветов с временем исполне-

ния $O(nm\Delta)$. Так как задача выяснения, можно ли использовать для раскраски на один цвет меньше, является NP-полной, то ее решение вряд ли стоит требуемых усилий.

Любую задачу реберной раскраски графа G можно преобразовать в задачу вершинной раскраски реберного графа $L(G)$, имеющего вершину графа $L(G)$ для каждого ребра в графе G, и ребро графа $L(G)$ тогда и только тогда, когда два ребра графа G инцидентны одной и той же вершине. Реберный граф можно создать за линейное время и можно раскрасить, используя любую программу вершинной раскраски.

Реализации. Реализация на языке C++ теоремы Визинга была осуществлена Яном Донгом (Yan Dong) в качестве курсового проекта, когда он учился у меня в университете Стоуни Брук. Загрузить эту программу можно с веб-страницы **http://www.cs.sunysb.edu/~algorith**.

Библиотека GOBLIN (**http://www.math.uni-augsburg.de/~fremuth/goblin.html**) содержит алгоритм метода ветвей и границ для реберной раскраски.

Список программ для вершинной раскраски, которые можно применить к реберному графу, построенному на основе вашего целевого графа, вы найдете в *разделе 16.7*. Библиотека Combinatorica (см. [PS03]) содержит реализации (на языке пакета Mathematica) алгоритмов реберной раскраски, работающих по тому же принципу, т. е. преобразующих целевой граф в реберный и применяющих к нему процедуры вершинной раскраски. Дополнительную информацию по Combinatorica см. в *разделе 19.1.9*.

Примечания

Обзоры теоретических исследований задачи реберной раскраски представлены в [FW77] и [GT94]. Доказательство, что реберную раскраску любого графа можно выполнить, используя для этого, самое большее, $\Delta + 1$ цвет, было независимо представлено Визингом (Vizing) (см. [Viz64]) и Гуптой (Gupta) (см. [Gup66]). Простое конструктивное доказательство этого результата приводят Мисра (Misra) и Гриз (Gries) в своей работе [MG92]. Несмотря на свою специфичность, задача вычисления реберно-хроматического числа является NP-полной (см. [Hol81]). Реберную раскраску двудольных графов можно выполнить за полиномиальное время (см. [Sch98]).

Представляя реберные графы в своей работе [Whi32], Уитни (Whitney) показал, что за исключением графов K_3 и $K_{1,3}$, любые два связных графа, реберные графы которых изоморфны, являются изоморфными. Интересным упражнением будет поиск доказательства того, что реберный граф эйлерова графа является как эйлеровым, так и гамильтоновым, в то время как реберный граф гамильтонова графа всегда будет только гамильтоновым.

Родственные задачи. Вершинная раскраска (см. *раздел 16.7*), календарное планирование (см. *раздел 14.9*).

16.9. Изоморфизм графов

Вход. Графы G и H.

Задача. Найти отображение f (или все возможные отображения) множества вершин графа G в множество вершин графа H такое, что граф G и граф H идентичны, т. е. (x, y) является ребром графа G тогда и только тогда, когда $(f(x), f(y))$ является ребром графа H (рис. 16.10).

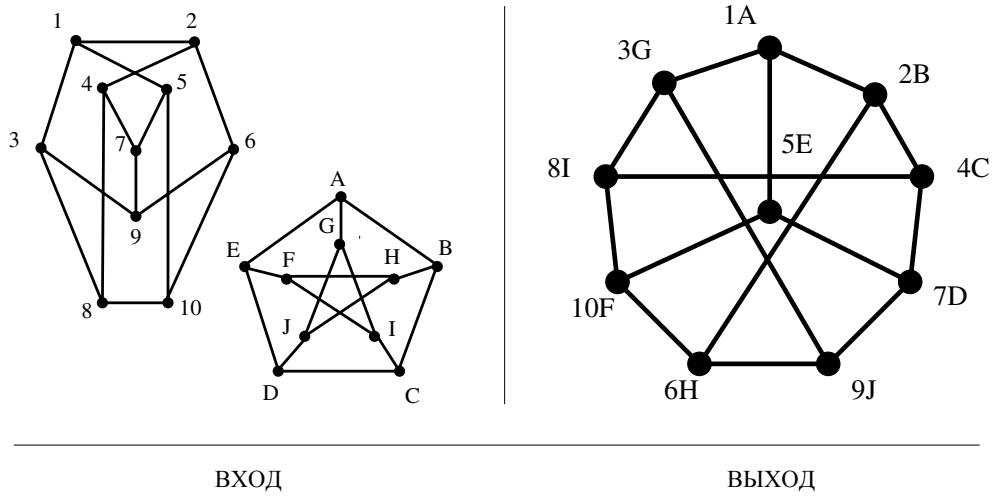


Рис. 16.10. Изоморфизм графов

Обсуждение. Задача выявления изоморфизма графов заключается в проверке графов на идентичность. Допустим, что требуется выполнить определенные операции над каждым графом из некоторой коллекции. Если мы сможем выяснить, что какие-то графы идентичны друг другу, мы будем игнорировать копии, чтобы не делать двойную работу.

Некоторые задачи распознавания образов можно с легкостью сформулировать в виде задачи выявления изоморфизма графов или подграфов. Например, структура химических соединений естественным образом описывается помеченными графами, в которых каждая вершина представляет отдельный атом. Поиск в базе данных всех молекул, содержащих определенную функциональную группу, является задачей выявления изоморфизма подграфов.

Уточним, что мы понимаем под идентичностью графов. Два помеченных графа $G = (V_g, E_g)$ и $H = (V_h, E_h)$ являются идентичными, если $(x, y) \in E_g$ тогда и только тогда, когда $(x, y) \in E_h$. Выявление изоморфизма заключается в поиске отображения вершин графа G на множество вершин графа H, при котором графы идентичны. Это отображение и называется *изоморфизмом*.

Другим важным применением изоморфизма графов является выявление симметрии. Отображение графа на самого себя называется автоморфизмом, а коллекция автоморфизмов (группа автоморфизмов) содержит много информации о симметричности графа. Например, полный граф K_n содержит $n!$ автоморфизмов (годится любое отображение), в то время как произвольный граф, скорее всего, будет иметь малое количество автоморфизмов, возможно, только один, поскольку граф G идентичен себе самому. На практике возникает несколько видов задачи изоморфизма графов. Чтобы их распознать, постарайтесь ответить на следующие вопросы.

- *Содержится ли граф G в графе H?* Вместо проверки на идентичность мы нередко должны выяснить, является ли данный граф G *подграфом* графа H. Такие задачи, как задача о клике, независимом множестве и гамильтоновом цикле, являются важными частными случаями задачи выявления изоморфизма подграфов.

Выражение "граф G содержится в графе H" имеет два разных значения в теории графов. В задаче выявления *изоморфизма подграфа* требуется выяснить, содержит ли граф H подмножество ребер и вершин, которое является изоморфным графу G. А в задаче выявления изоморфизма порожденного подграфа требуется выяснить, содержит ли граф H подмножество ребер и вершин, после удаления которого останется подграф, изоморфный графу G. В задаче выявления изоморфизма порожденного подграфа требуется, чтобы все ребра графа G присутствовали в графе H, и чтобы в графе H не было никаких "не-ребер" графа G. Клика является экземпляром обоих вариантов задачи выявления изоморфизма подграфов, в то время как гамильтонов цикл является примером лишь простого изоморфизма подграфов.

Следует учитывать это различие при работе над вашим приложением. Задачи выявления изоморфизма подграфов обычно сложнее, чем задачи выявления изоморфизма графов, а задачи выявления изоморфизма порожденных подграфов еще сложнее. Единственным разумным подходом к решению таких задач является метод перебора с возвратом.

- *Помечены графы или нет?* Во многих приложениях вершины и/или ребра графов помечаются каким-либо атрибутом, который нужно учитывать при выявлении изоморфизма. Например, при сравнении двудольных графов, содержащих вершины двух типов, скажем, "рабочий" и "задание", любой изоморфизм, уравнивающий задание с рабочим, будет лишен смысла.

  Метки и связанные с ними ограничивающие условия можно включить в любой алгоритм метода перебора с возвратом. Кроме этого, такие ограничивающие условия значительно ускоряют поиск, создавая намного больше возможностей для разрежения пространства поиска при каждом случае несовпадения меток двух вершин.

- *Являются ли деревьями графы, которые проверяются на изоморфизм?* Для проверки на изоморфизм некоторых частных случаев графов, таких как деревья и планарные графы, существуют более быстрые алгоритмы. Возможно, самым важным случаем задачи выявления изоморфизма является выявление изоморфизма деревьев. Эта задача возникает при сопоставлении языковых структур и синтаксическом анализе. Для описания структуры текста часто используется дерево синтаксического разбора. Два таких дерева будут изоморфными, если представляемые ими тексты имеют одинаковую структуру.

  Эффективные алгоритмы выявления изоморфизма деревьев начинают работу с листьев обоих деревьев и продвигаются к центру. Каждой вершине дерева присваивается метка, представляющая набор вершин во втором дереве, который, возможно, изоморфен поддереву с корнем в этой вершине, с учетом ограничений, накладываемых метками и степенями вершин. Например, первоначально все листья дерева T_1 потенциально эквивалентны всем листьям дерева T_2. Теперь, продвигаясь внутрь, мы можем разбить смежные с листьями вершины дерева T_1 на классы, в зависимости от количества смежных с ними листьев и других узлов. Отслеживая метки поддеревьев, мы можем убедиться, что имеем одинаковое распределение помеченных поддеревьев для T_1 и T_2. Любое несовпадение означает, что $T_1 \neq T_2$, в то время как по завершении процесса все вершины оказываются разбиты на классы эквивалентности, определяющие все изоморфизмы.

- *Сколько имеется графов?* Во многих приложениях обработки данных требуется выполнить поиск всех экземпляров графа с определенной структурой в большой базе данных. Уже упоминавшаяся задача отображения химических структур относится к этому типу задач. Такие базы данных обычно содержат большое количество сравнительно небольших графов. В связи с этим возникает необходимость индексирования базы данных графов по небольшим подструктурам (от пяти до десяти вершин каждая) и выполнения дорогостоящих проверок графов на изоморфность только с теми, которые содержат такие же подструктуры, что и граф запроса.

Для решения задачи выявления изоморфизма графов не предложено ни одного алгоритма с полиномиальным временем исполнения и в то же время не выяснено, является ли эта задача NP-полной. Вместе с задачей разложения на множители целых чисел (см. *раздел 13.8*), это одна из нескольких важных алгоритмических задач, вычислительная сложность которых до сих пор неизвестна, даже приблизительно. Принято считать, что если P≠NP, то задача выявления изоморфизма занимает промежуточное положение между P- и NP-полной.

Но хотя не известно алгоритмов для решения наихудших случаев задачи за полиномиальное время, задача выявления изоморфизма на практике *обычно* не очень сложна. Основной алгоритм выполняет перебор с возвратом всех $n!$ возможных замен меток вершин графа h на метки вершин графа g, а потом проверяет графы на идентичность. Конечно же, пространство поиска можно разредить, убрав все перестановки с одинаковым префиксом при первом же несовпадении ребер, обе вершины которых имеют этот префикс.

Но настоящим ключом к эффективной проверке на изоморфизм будет предварительное разбиение множества вершин на "классы эквивалентности" таким образом, чтобы было невозможно перепутать две вершины из разных классов. Все вершины в каждом классе эквивалентности должны иметь одинаковое значение какого-либо инварианта, не зависящего от меток. Возможно использование одного из следующих инвариантов:

- *степень вершины.* Самый простой способ разбиения множества вершин на части основан на их степени, т. е. на количестве ребер, инцидентных каждой вершине. Две вершины с разными степенями не могут быть одинаковыми. Такое простое разбиение может быть очень полезным, но не в случае с регулярными графами (т. е. графами, у которых все вершины имеют одинаковую степень);
- *матрица кратчайших путей.* Для каждой вершины v матрица кратчайших путей между всеми парами (см. *раздел 15.4*) определяет мультимножество из $n-1$ расстояний, представляющих расстояния между вершиной v и каждой из остальных вершин. Любые две одинаковые вершины должны определять одинаковые мультимножества расстояний, поэтому мы можем разбить вершины на классы эквивалентности, определяющие одинаковые мультимножества расстояний;
- *количество путей длиной k*. Возведение в k-ю степень матрицы смежности графа G дает матрицу, в которой ячейка $G^k[i,j]$ содержит количество путей от вершины i к вершине j. Для каждой вершины и для каждого значения k эта матрица определяет количество путей, которое можно использовать для разбиения вершин на классы, как и расстояния в предыдущем случае. Можно перепробовать все $1 \le k \le n$ или да-

же лежащие за пределами этого диапазона, и использовать любое отклонение как критерий для разбиения.

Используя эти инварианты, часто удается разбить граф на много небольших классов эквивалентности. После такого разбиения вершин не составит труда довести работу до конца, используя метод перебора с возвратом. Каждой вершине в качестве метки присваивается имя ее класса эквивалентности, и задача решается как задача паросочетания в помеченном графе. Выявить изоморфизм в высокосимметричных графах сложнее, чем в случайных, по причине снижения эффективности таких эвристических методов разбиения вершин на множество классов эквивалентности.

Реализации. Наиболее известной программой для проверки графов на изоморфизм является программа nauty, представляющая собой набор очень эффективных процедур на языке C для поиска группы автоморфизма графа с вершинной раскраской. Программа также может выдавать канонически помеченный граф, изоморфный данному, чтобы содействовать проверке на изоморфизм. На основе nauty была разработана первая программа для генерирования всех 11-вершинных графов, не имеющих изоморфных графов, которая также может тестировать большинство графов с меньше чем 100 вершинами быстрее, чем за одну секунду. Программа nauty перенесена на разные операционные системы и компиляторы языка C. Загрузить ее можно с веб-сайта **http://cs.anu.edu.au/~bdm/nauty/**. Теоретические основы программы nauty изложены в работе [McK81].

Библиотека средств для сравнения графов VFLib содержит реализации нескольких разных алгоритмов для проверки на изоморфизм как графов, так и подграфов. Библиотека была очень тщательно протестирована (см. [FSV01]). Загрузить ее можно с веб-сайта **http://amalfi.dis.unina.it/graph**.

Пакет программного обеспечения GraphGrep (см. [GS02]) предназначен для поиска запрашиваемого графа в большой базе данных графов. Загрузить пакет можно с веб-сайта **http://www.cs.nyu.edu/shasha/papers/graphgrep**.

Реализации алгоритмов на языке C++ для проверки изоморфизма графов и подграфов как для деревьев, так и для графов из книги [Val02] предоставляются для общего пользования. Эти реализации основаны на кодах библиотеки LEDA (см. *раздел 19.1.1*); загрузить их можно с веб-сайта **http://www.lsi.upc.edu/~valiente/algorithm**.

В книге [KS99], в дополнение к более общим операциям теории графов, представлены программы для выявления изоморфизма графов. Эти программы, написанные на языке C, можно загрузить с веб-сайта **http://www.math.mtu.edu/~kreher/cages/Src.html**.

Примечания

Выявление изоморфизма графов — важная задача теории сложности. Среди монографий, посвященных выявлению изоморфизма, можно выделить работу [Hof82] и книгу [KST93]. Книга [Val02] посвящена, преимущественно, алгоритмам выявления изоморфизма деревьев и подграфов. Подход к проверке на изоморфизм, применяемый в книге [KS99], основан на теории групп. Обзор систем и алгоритмов анализа данных, представленных в виде графов, приводится в книге [CH06]. Результаты сравнения производительности разных алгоритмов выявления изоморфизма графов и подграфов содержатся в работе [FSV01].

Существуют алгоритмы с полиномиальным временем исполнения для выявления полиморфизма планарных графов (см. [HW74]) и для графов, у которых максимальная степень

вершин ограничена константой (см. [Luk80]). Эвристический метод на основе кратчайшего пути между всеми парами был представлен в работе [SD76], хотя существуют неизоморфные графы, имеющие точно такое же множество расстояний (см. [BH90]). Алгоритм с линейным временем исполнения для выявления изоморфизмов в помеченных и непомеченных деревьях представлен в книге [AHU74]:

Задача называется *изоморфно-полной*, если она доказуемо такая же сложная, как и задача изоморфизма. Задача проверки на изоморфизм двудольных графов является изоморфно-полной, т. к. любой граф можно преобразовать в двудольный, заменив каждое из его ребер двумя ребрами, соединенными с новой вершиной. Очевидно, что исходные графы являются изоморфными тогда и только тогда, когда изоморфны графы, полученные в результате такого преобразования.

Родственные задачи. Задача кратчайшего пути (см. *раздел 15.4*), поиск подстрок (см. *раздел 18.3*).

16.10. Дерево Штейнера

Вход. Граф $G = (V, E)$, подмножество вершин $T \in V$.

Задача. Найти наименьшее дерево, соединяющее все вершины T (рис. 16.11).

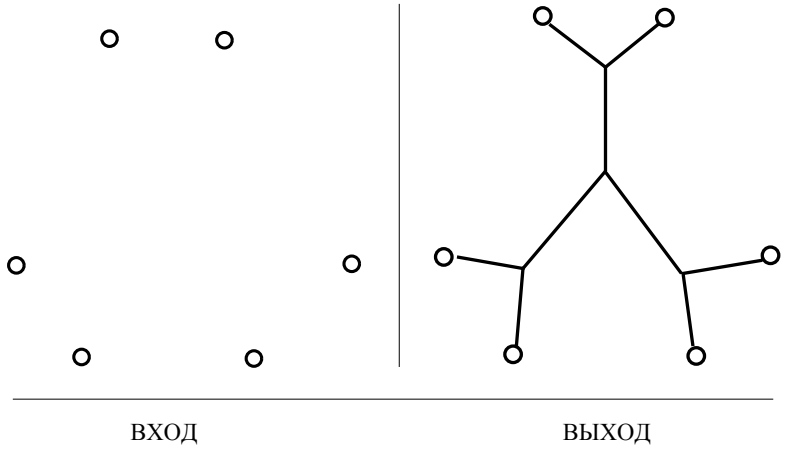


Рис. 16.11. Дерево Штейнера

Обсуждение. Задача построения деревьев Штейнера часто возникает при разработке коммуникационных сетей, т. к. минимальное дерево Штейнера описывает, как соединить данное множество станций, используя кабель наименьшей протяженности. Аналогичные задачи возникают при разработке водопроводных или вентиляционных и отопительных сетей и при разработке сверхбольших интегральных микросхем (СБИС). В области разработки СБИС типичной задачей построения дерева Штейнера является соединение набора площадок, например, с проводом заземления, при соблюдении ограничивающих условий, таких как стоимость материала, время распространения сигнала или емкостное сопротивление.

Задача построения дерева Штейнера отличается от задачи построения минимального остовного дерева (см. *раздел 15.3*) тем, что в ней разрешается создавать или выбирать

промежуточные соединительные узлы, чтобы уменьшить стоимость дерева. Прежде чем приступать к построению дерева Штейнера, ответьте на несколько вопросов.

- *Сколько точек требуется соединить?* Дерево Штейнера для двух вершин — это просто кратчайший путь между ними (см. *раздел 15.4*). А дерево Штейнера для всех вершин, когда $S = V$, просто определяет минимальное остовное дерево графа. Несмотря на наличие таких простых частных случаев, общая задача построения дерева Штейнера является NP-полной и в широком диапазоне ограничивающих условий.
- *Являются ли входные данные набором точек или графом расстояний?* Геометрические версии задачи построения дерева Штейнера принимают в качестве входа набор точек, как правило, расположенных на плоскости, и заключаются в поиске наименьшего дерева, соединяющего эти точки. Однако возникает затруднение следующего вида. Набор допустимых промежуточных точек не задается как часть входа, а должен быть получен из исходного набора точек. Эти промежуточные точки должны удовлетворять определенным геометрическим условиям, благодаря которым набор точек-кандидатов становится конечным. Например, в минимальном дереве Штейнера степень каждой точки Штейнера равна 3, а угол между любыми двумя ребрами такой точки должен быть ровно 120°.
- *Можно ли воспользоваться какими-нибудь ограничениями, наложенными на ребра?* Многие задачи монтажа электропроводки соответствуют геометрическим версиям задачи, в которых все ребра должны располагаться либо горизонтально, либо вертикально. Задача такого типа называется *линейной задачей Штейнера*. В задаче построения линейного дерева Штейнера на углы между ребрами и степени вершин накладываются иные ограничения, чем в задаче построения евклидова дерева. В частности, все углы должны быть кратны 90°, а максимальная степень любой вершины не должна превышать 4.
- *Действительно ли нужно оптимальное дерево?* В некоторых приложениях задачи построения дерева Штейнера (например, при проектировании печатных плат или коммуникационных сетей) большой объем вычислений для построения оптимального дерева Штейнера вполне оправдан. В таких случаях применяются исчерпывающие методы поиска, такие как перебор с возвратом или метод ветвей и границ. Существует много возможностей для прореживания пространства поиска с использованием различных геометрических ограничивающих условий.

  Тем не менее, задача построения дерева Штейнера остается сложной. Поэтому, прежде чем пытаться разрабатывать свою собственную реализацию, попробуйте применить уже существующие, описанные далее в *подразделе "Реализации"*.
- *Как восстановить точки Штейнера, о которых не было известно вначале?* В приложениях из области классификации и эволюции возникает очень специфичный тип дерева Штейнера. *Филогенетическое дерево* иллюстрирует относительную схожесть разных объектов или биологических видов. Каждый объект обычно представляется листом дерева, а промежуточные вершины соответствуют точкам разветвления между классами объектов. Например, в эволюционном дереве видов животного мира листья могут представлять *человека*, *собаку*, *змею*, а внутренние узлы соответствуют таксонам *(животные*, *млекопитающие*, *пресмыкающиеся)*. Дерево с корнем

животные, в котором *человек* и *собака* находятся в таксоне *млекопитающие*, означает, что человек имеет более близкое родство с собаками, чем со змеями.

Было разработано много разных алгоритмов для создания филогенетических деревьев, которые отличаются друг от друга моделируемыми данными и критериями оптимизации. Разные комбинации реконструирующего алгоритма и метрики выдают разные ответы, поэтому определение "правильного" метода в каждом конкретном случае весьма субъективно. Разумным решением будет применение одного из упомянутых далее стандартных пакетов реализаций и сравнительный анализ результатов обработки данных всеми входящими в него реализациями.

К счастью, существует эффективный эвристический алгоритм для построения деревьев Штейнера, хорошо работающий на всех версиях задачи. Создаем граф, моделирующий вход задачи, и устанавливаем вес ребра (i, j) равным расстоянию от точки i до точки j. Находим минимальное остовное дерево для этого графа. Оно гарантированно является хорошей аппроксимацией как для евклидовых деревьев, так и для линейных деревьев Штейнера.

Самым худшим случаем для получения приблизительного минимального остовного дерева для евклидова дерева Штейнера является входной экземпляр из трех точек, образующих равносторонний треугольник. В этом случае минимальное остовное дерево содержит две стороны треугольника (с общей длиной ребер равной 2), в то время как минимальное дерево Штейнера соединяет все три точки с использованием еще одной, внутренней, и общая длина его ребер равна $\sqrt{3}$. Это соотношение $\sqrt{3}/2 \approx 0{,}866$ всегда достижимо, и на практике минимальное остовное дерево обычно имеет размер, на несколько процентов отличающийся от размера оптимального дерева Штейнера. Отношение размера линейного дерева Штейнера к размеру минимального остовного дерева всегда равно $2/3 \approx 0{,}667$.

Такое минимальное остовное дерево всегда можно улучшить, вставив в него точку Штейнера в любом месте, где угол между инцидентными какой-либо вершине ребрами минимального остовного дерева меньше 120°. Вставив эту точку и откорректировав ребра, можно приблизить решение к оптимальному еще на несколько процентов. Подобная оптимизация возможна и для линейных остовных деревьев.

Заметим, что нас интересует только поддерево, соединяющее листья. Если к входу задачи добавить вершины, не являющиеся конечными, то, возможно, потребуется откорректировать минимальное остовное дерево. В таком случае нужно будет оставить только те ребра, которые лежат на (уникальном) пути между какой-либо парой конечных узлов. Полный набор таких ребер можно найти за время $O(n)$, выполнив обход в ширину всего дерева, начиная с любого листа.

Альтернативный эвристический алгоритм для графов основан на методе кратчайших путей. Начинаем с дерева, состоящего из кратчайшего пути между двумя конечными узлами. Для каждого оставшегося конечного узла t выбираем кратчайший путь к вершине v внутри дерева и добавляем этот путь к дереву. Качество и временная сложность этого эвристического алгоритма зависят от порядка вставки конечных узлов и от способа вычисления кратчайших путей, но вероятность получить простое и эффективное решение достаточно высока.

Реализации. Варм (Warme), Винтер (Winter) и Закариасен (Zachariasen) разработали пакет GeoSteiner, содержащий программы построения как евклидова, так и линейного дерева Штейнера на плоскости (см. [WWZ00]). Пакет можно также использовать для решения родственной задачи построения минимального остовного дерева в гиперграфах. Утверждается, что с его помощью были получены оптимальные решения для экземпляров, состоящих из 10 000 точек. Это, пожалуй, самый лучший пакет для решения геометрических экземпляров задачи построения дерева Штейнера. Загрузить пакет можно с веб-сайта **http://www.diku.dk/hjemmesider/ansatte/martinz/geosteiner/**.

Пакет FLUTE предназначен для быстрого построения линейных деревьев Штейнера. Программа предоставляет управляемый пользователем параметр для контроля соотношения между качеством решения и временем исполнения.

Библиотека GOBLIN (**http://www.math.uni-augsburg.de/~fremuth/goblin.html**) содержит реализации как эвристических, так и поисковых алгоритмов для построения деревьев Штейнера в графах.

Пакеты PHYLIP (**http://evolution.genetics.washington.edu/phylip.html**) и PAUP (**http://paup.csit.fsu.edu/**) широко применяются для построения филогенетических деревьев. Оба пакета содержат реализации свыше 20 алгоритмов для создания филогенетических деревьев. Хотя многие из них предназначены для работы с моделями молекулярных последовательностей, некоторые методы принимают в качестве входа произвольные матрицы расстояний.

Примечания

В числе последних монографий, посвященных деревьям Штейнера, можно назвать [HRW92] и [PS02]. Книга [DSR00] содержит коллекцию сравнительно свежих обзоров по всем аспектам построения деревьев Штейнера. Более старый обзор задачи можно найти в [Kuh75]. Результаты исследований эвристических методов работы с деревьями Штейнера изложены в [SFG82] и [Vos92].

Задача построения евклидовых деревьев Штейнера была впервые поставлена Ферма, который изучал вопрос поиска на плоскости точки p такой, что сумма расстояний до трех данных точек минимальна. Эта задача была решена Торричелли до 1640 г. По-видимому, над общей задачей для n точек работал также Штейнер, и поэтому ему по ошибке приписывается авторство решения данной задачи. Подробное и интересное изложение истории задачи представлено в работе [HRW92].

Гилберт (Gilbert) и Поллак (Pollak) (см. [GP68]) были первыми, кто выдвинул предположение, что отношение размера минимального дерева Штейнера к размеру минимального остовного дерева всегда $\geq \sqrt{3}/2 \approx 0{,}866$. После двадцати лет активных исследований, отношение Гильберта-Поллака было, наконец, доказано Ду (Du) и Хвангом (Hwang); см. [DH92]. Евклидово минимальное остовное дерево для n точек можно создать за время $O(n\lg n)$ (см. [PS85]).

Аппроксимирующая схема с полиномиальным временем исполнения для деревьев Штейнера в k-мерном евклидовом пространстве была представлена в работе [Aro98]. А в работе [RZ05] приведен метод получения аппроксимирующего решения задачи построения деревьев Штейнера для графов, имеющего коэффициент 1,55.

Доказательство сложности задачи построения дерева Штейнера для графов (см. [Kar72]) содержится в [Eve79a]. Описания точных алгоритмов построения деревьев Штейнера для графов можно найти в [Law76]. Сложность задачи построения дерева Штейнера для

евклидовых и манхэттеновских метрик была доказана в работах [GGJ77] и [GJ77]. Неизвестно, является ли задача построения евклидова дерева Штейнера NP-полной, в силу проблем с представлением расстояний.

Можно провести аналогию между минимальными деревьями Штейнера и структурами с минимальной энергией в некоторых физических системах. Предположение, что такие системы (например, мыльные пленки на проволочных рамках) "решают" задачу построения дерева Штейнера, обсуждается в [Mie58].

Родственные задачи. Минимальное остовное дерево (см. *раздел 15.3*), поиск кратчайшего пути (см. *раздел 15.4*).

16.11. Разрывающее множество ребер или вершин

Вход. Ориентированный граф $G = (V, E)$.

Задача. Найти наименьший набор ребер E' или вершин V', удаление которого сделает граф ациклическим (рис. 16.12).

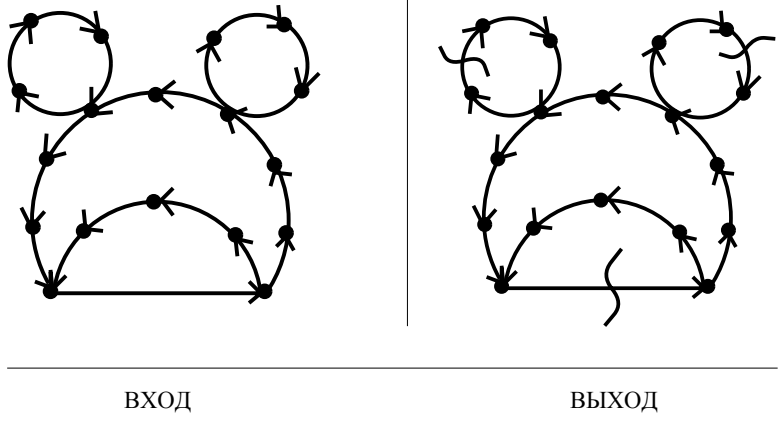


Рис. 16.12. Разрывающее множество ребер

Обсуждение. Необходимость в поиске разрывающего множества возникает из-за того, что многие задачи легче поддаются решению на бесконтурных орграфах, чем на ориентированных графах общего вида. Рассмотрим, например, задачу планирования расписания работ с ограничивающими условиями очередности, т. е. задача A должна выполняться перед задачей B. Когда все ограничивающие условия не противоречат друг другу, то получается бесконтурный орграф, и для соблюдения условий вершины можно упорядочить посредством топологической сортировки (см. *раздел 15.2*). Но как составить расписание, когда существуют циклические ограничивающие условия, например, задание A нужно выполнить перед заданием B, которое нужно выполнить перед заданием C, которое нужно выполнить перед заданием A?

Найдя разрывающее множество, мы определяем наименьшее количество ограничивающих условий, которые нужно отбросить, чтобы получить допустимое расписание.

В задаче *разрывающего множества ребер* (или дуг) мы отбрасываем отдельные ограничивающие условия очередности. А в задаче *разрывающего множества вершин* мы отбрасываем целые задания вместе со связанными с ними ограничениями.

Аналогичным образом устраняется состояние гонок в электронных схемах. Задача разрывающего множества также называется задачей поиска *максимального ациклического подграфа.*

Еще одно приложение связано с определением рейтинга спортсменов. Допустим, нам требуется определить рейтинг участников шахматных или теннисных соревнований. Для этого мы можем создать ориентированный граф, в котором ребро идет от вершины x к вершине y, если игрок x победил игрока y. По идее, игрок более высокого класса должен победить игрока низшего класса, хотя противоположный (неожиданный) результат наблюдается довольно часто. Естественным определением рейтинга будет топологическое упорядочение, полученное после удаления из графа минимального разрывающего множества ребер, представляющих неожиданные результаты матча.

Решая задачу разрывающего множества, ответьте на следующие вопросы.

- *Нужно ли отбросить какие-либо ограничивающие условия?* Если граф уже является бесконтурным орграфом, что можно выяснить с помощью топологической сортировки, то никакие изменения не требуются. Один из способов поиска разрывающего множества состоит в модифицировании алгоритма топологической сортировки таким образом, чтобы при обнаружении конфликта удалялось проблемное ребро или вершина. Но это разрывающее множество может оказаться намного больше требуемого, т. к. на ориентированных графах задачи поиска разрывающего множества ребер и разрывающего множества вершин являются NP-полными.
- *Как найти подходящее разрывающее множество ребер?* Эффективный эвристический алгоритм с линейным временем исполнения создает вершинное упорядочение, а потом удаляет каждую дугу, идущую в неправильном направлении. При любом упорядочении вершин направление, по крайней мере, половины дуг должно быть одинаковым (слева направо или справа налево), поэтому в качестве разрывающего множества следует взять то, которое меньше.

  Как правильно выбрать начальную вершину? Вполне естественно отсортировать вершины по их реберной разбалансированности, т. е. по разности между степенью захода и степенью исхода. При другом подходе в качестве начальной берется случайная вершина v. Любая вершина x, определяемая входящим ребром (x, v), будет помещена слева от вершины v. Аналогичным образом, любая вершина x, определяемая исходящим ребром (v, x), будет помещена справа от вершины v. Теперь можно рекурсивно выполнить эту процедуру на левом и правом подмножествах, чтобы завершить упорядочение вершин.
- *Как найти хорошее подходящее множество вершин?* Только что описанная эвристическая процедура возвращает упорядоченный набор вершин с некоторым количеством обратных ребер. Нам нужно найти небольшое множество вершин, покрывающее эти обратные ребра. То есть мы имеем задачу о вершинном покрытии, эвристические методы решения которой рассматриваются в *разделе 16.3*.
- *Как поступить, когда нужно разорвать все циклы в неориентированном графе?* Задача поиска разрывающих множеств в неориентированных графах существенно

отличается от задачи их поиска в орграфах. Деревья являются ациклическими неориентированными графами, и каждое дерево из n вершин содержит ровно $n-1$ ребро. Таким образом, мощность наименьшего разрывающего множества ребер любого неориентированного графа G будет $|E|-(n-c)$, где c — количество компонент связности графа G. Обратные ребра, обнаруженные при обходе в глубину графа G, можно рассматривать как минимальное разрывающее множество ребер.

Задача поиска разрывающего множества вершин для неориентированных графов является NP-полной. Эвристическая процедура для ее решения использует обход в глубину для поиска самого короткого цикла графа. Все вершины найденного цикла удаляются из графа, и в оставшемся графе вновь выполняется поиск цикла минимальной длины, который снова удаляется. Эта процедура поиска и удаления циклов повторяется до тех пор, пока граф не станет ациклическим. Оптимальное разрывающее множество вершин должно содержать хотя бы одну вершину, принадлежащую каждому из этих циклов (у которых нет общих вершин), и поэтому качество полученного приблизительного решения определяется средней длиной удаленных циклов.

Иногда имеет смысл улучшить эвристические решения, используя рандомизацию или метод имитации отжига. Для перехода из одного состояния в другое можно изменять порядок вершин, обменивая пары вершин местами или вставляя/удаляя вершины множества, являющегося кандидатом на роль разрывающего.

Реализации. Алгоритм 815 из коллекции алгоритмов ACM (см. *раздел 19.1.6*) представляет собой эвристическую процедуру GRASP для поиска разрывающего множества как вершин, так и ребер (см. [FPR01]). Эти коды на языке FORTRAN можно также загрузить с веб-сайта **http://www.research.att.com/~mgcr/src.**

Библиотека GOBLIN (**http://www.math.uni-augsburg.de/~fremuth/goblin.html**) содержит реализацию аппроксимирующего алгоритм поиска минимального разрывающего множества дуг.

Программа econ_order из базы графов Standford GraphBase (см. *раздел 19.1.8*) выполняет перестановку рядов и столбцов матрицы таким образом, чтобы минимизировать суммы чисел под главной диагональю. Использование матрицы смежности в качестве входа и удаление всех ребер под главной диагональю позволяет получить ациклический граф.

Примечания

Обзор решений задачи разрывающего множества представлен в [FPR99]. Обсуждение доказательства сложности задачи поиска минимального разрывающего множества (см. [Kar72]) можно найти в [AHU74] и [Eve79a]. Как задача разрывающего множества вершин, так и задача разрывающего множества ребер остается сложной даже в том случае, когда степень захода и исхода всех вершин не больше двух (см. [GJ79]).

В работе [BBF99] представлен аппроксимирующий алгоритм, выдающий решение для задачи разрывающего множества вершин, имеющее коэффициент 2. Разрывающее множество ребер в ориентированных графах можно аппроксимировать с коэффициентом $O(\log n \log\log n)$ (см. [ENSS98]). В работе [CFR06] рассматривается эвристическая про-

цедура определения рейтинга участников соревнований. Эксперименты с эвристическими методами описаны в журнале [Koe05].

Интересное применение задачи разрывающего множества дуг в экономике представлено в книге [Knu94]. Для каждой пары A, B секторов экономики дается информация об объеме потока финансов из сектора A в сектор B. Требуется упорядочить секторы таким образом, чтобы выяснить, какие из них являются преимущественно поставщиками для других секторов, а какие поставляют товар, в основном, потребителям.

Родственные задачи. Уменьшение ширины ленты матрицы (см. *раздел 13.2*), топологическая сортировка (см. *раздел 15.2*), календарное планирование (см. *раздел 14.9*).

ГЛАВА 17

Вычислительная геометрия

Вычислительная геометрия — это область дискретной математики, в которой изучаются алгоритмы решения геометрических задач. Ее возникновение совпало с возникновением таких прикладных областей, как компьютерная графика и системы автоматизированного проектирования и производства, а также с более широким применением компьютеров в различных научных отраслях. Все эти факторы стимулируют развитие вычислительной геометрии. В течение последних двадцати лет это развитие происходило бурными темпами, в результате чего на свет появилось множество полезных алгоритмов, программ, исследовательских работ и учебников. Среди книг по вычислительной геометрии можно выделить следующие:

- [dBvKOS00] — самое лучшее введение в общую теорию вычислительной геометрии и ее основные алгоритмы;
- [O'R01] — прекрасное практическое введение в вычислительную геометрию. В нем особое внимание уделяется тщательной и правильной реализации алгоритмов. Код на языках C и Java можно загрузить с веб-страницы **http://maven.smith.edu/~orourke/books/compgeom.html**;
- [PS85] — несмотря на солидный возраст, эта книга остается хорошим введением в вычислительную геометрию. В ней подробно описаны алгоритмы построения выпуклой оболочки, построения диаграмм Вороного и выявления пересечений;
- [GO04] — недавно изданный сборник статей содержит подробный обзор положения дел во всех областях дискретной и вычислительной геометрии.

Симпозиум ассоциации ACM по вычислительной геометрии (ACM Symposium on Computational Geometry) проводится ежегодно в последних числах мая или в первых числах июня. Хотя на нем представляются, в основном, теоретические результаты, научное сообщество прилагает усилия по расширению присутствия прикладных и экспериментальных работ, используя для этого видеообзоры и стендовые доклады.

Существует постоянно расширяющаяся база реализаций алгоритмов вычислительной геометрии. Как правило, в каталоге указываются конкретные реализации, но вам следует обратить особое внимание на обширную библиотеку CGAL (Computational Geometry Algorithms Library, библиотека алгоритмов вычислительной геометрии). Она содержит реализации большого количества алгоритмов на языке C++, разработанные как часть крупного общеевропейского проекта. Каждому, кто серьезно интересуется вычислительной геометрией, следует ознакомиться с этой библиотекой (**http://www.cgal.org**).

17.1. Элементарные задачи вычислительной геометрии

Вход. Точка p и отрезок l или два отрезка, l_1 и l_2.

Задача. Выяснить, находится ли точка p на отрезке l, а если не находится, то по какую сторону от него она расположена. Выяснить, пересекаются ли отрезки l_1 и l_2 (рис. 17.1).

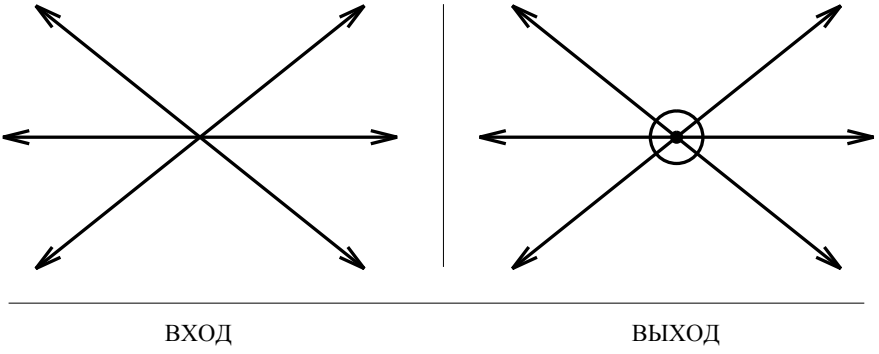


Рис. 17.1. Пересечение прямых

Обсуждение. Элементарные задачи вычислительной геометрии имеют свои "подводные камни" даже в таких простых случаях, как поиск точки пересечения двух прямых. Эта задача сложнее, чем кажется на первый взгляд. Например, какой результат нужно возвратить, если прямые параллельны? Как поступить, если прямые совпадают, и пересечением является не точка, а вся прямая? Если одна из прямых расположена горизонтально, в процессе решения уравнений придется выполнить деление на нуль. Если прямые почти параллельны, точка пересечения находится так далеко от начала координат, что при вычислении ее местоположения возникнет арифметическое переполнение. Ситуация значительно усложняется при поиске пересечения двух отрезков, поскольку тогда возникает много частных случаев, которые нужно отслеживать.

Если вы новичок в области реализации алгоритмов вычислительной геометрии, я рекомендую вам изучить книгу [O'R01], в которой содержатся практические советы и законченные реализации основных алгоритмов вычислительной геометрии и структур данных.

Мы имеем дело с двумя разными понятиями: геометрической вырожденностью и численной устойчивостью. Под *вырожденностью* (degeneracy) понимаются частные случаи, требующие специальных подходов, например, такие, в которых две прямые имеют несколько точек пересечения. Существует три основных подхода к решению проблемы вырожденности:

- *игнорировать ее.* Выдвигаем рабочую гипотезу, что наша программа будет работать правильно только в тех случаях, когда никакие три точки не коллинеарны, никакие три прямые не пересекаются в одной точке, пересечение отрезков не происходит в их конечных точках и т. д. Это, пожалуй, самый распространенный подход, который я могу рекомендовать для краткосрочных проектов, если допустимы частые отказы программы. Недостаток такого подхода обусловлен тем фактом, что самые инте-

ресные данные снимаются с линий решетки и поэтому неизбежно являются сильно вырожденными;

- *подделать невырожденность*. Случайным образом внесите беспорядок в данные, чтобы сделать их невырожденными. Перемещая каждую точку на небольшое расстояние в случайном направлении, можно исключить вырожденность данных, не создавая при этом слишком много новых проблем. Это первое, что вы должны попробовать, когда решите, что частота отказов вашей программы превышает допустимый уровень. Недостаток этого подхода состоит в том, что случайные изменения могут исказить входные данные неприемлемым для вашего приложения образом. Существуют способы внесения "символических" возмущений в данные для непротиворечивого исключения вырожденности, но корректное применение этих способов требует серьезных исследований;
- *обрабатывать вырожденность*. Программу можно сделать более устойчивой, добавив в нее специальный код для обработки каждого частного случая. Такой подход может быть оправдан, если его осуществить в начале разработки программы, но недопустимо добавлять "заплатки" при каждом сбое программы. Если вы решительно настроены на применение этого подхода, будьте готовы вложить в его реализацию значительные усилия.

В геометрических вычислениях часто применяются арифметические операции с плавающей точкой, из-за чего могут возникнуть такие проблемы, как переполнение и потеря точности. Существуют три основных подхода к обеспечению численной устойчивости:

- *использование целочисленных арифметических операций*. Принудительное размещение всех представляющих интерес точек на целочисленной решетке позволит выполнять точные проверки двух чисел на равенство или двух отрезков на пересечение. Нужно отдавать себе отчет в том, что точка пересечения двух прямых не всегда будет находиться строго на решетке. Тем не менее, это самый простой и лучший способ при условии, что он дает приемлемые результаты;
- *использование действительных чисел двойной точности*. При операциях над числами двойной точности с плавающей точкой вам может повезти настолько, что вы избежите численных ошибок. Однако лучше всего для представления данных использовать действительные числа одинарной точности, а двойную точность применять для промежуточных вычислений;
- *использование арифметических операций произвольной точности*. Алгоритм, использующий такой подход, несомненно даст правильные результаты, но будет работать очень медленно. Однако он завоевывает все большую популярность в научном сообществе. Тщательный анализ может свести к минимуму необходимость в арифметических операциях с высокой точностью и, следовательно, потерю производительности. Тем не менее вы должны понимать, что операции с высокой точностью выполняются на несколько порядков медленнее, чем стандартные арифметические операции с плавающей точкой.

Разработка устойчивого программного обеспечения для решения задач вычислительной геометрии связана со значительными трудностями. На практике самым лучшим подходом будет создание приложений на основе небольшого набора алгоритмов вы-

числительной геометрии, выполняющих большой объем вычислений низкого уровня. В число таких алгоритмов входят следующие:

- *вычисление площади треугольника.* Хотя хорошо известно, что площадь $A(t)$ треугольника $t = (a, b, c)$ равна половине произведения его основания на высоту, вычисление длины основания и высоты требует кропотливых вычислений с применением тригонометрических функций. Лучше вычислить *двойную* площадь, используя следующую формулу:

$$2 \cdot A(t) = \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = a_x b_y - a_y b_x + a_y c_x - a_x c_y + b_x c_y - c_x b_y$$

  Эта формула обобщает задачу вычисления $d!$-кратного объема многогранника в d-мерном пространстве. Таким образом, в трехмерном пространстве шестикратный ($3! = 6$) объем четырехгранника $t = (a, b, c, d)$ будет равен:

$$6 \cdot A(t) = \begin{vmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{vmatrix}$$

  Обратите внимание, что эти значения могут быть отрицательными, поэтому в дальнейших вычислениях следует использовать их модули. Определители обсуждаются в *разделе 13.4*.

  Концептуально самым простым способом вычисления площади многоугольника (или многогранника) будет выполнение его триангуляции с последующим суммированием площадей всех полученных треугольников. Реализацию изящного алгоритма, в котором не используется триангуляция, см. в [O'R01] и [SR03];

- *выяснение местоположения точки.* Как расположена точка c относительно прямой l? Чтобы дать корректный ответ на этот вопрос, следует представить прямую l в виде направленной линии, проходящей через точку a раньше, чем через точку b, и выяснить, слева или справа от этой линии находится точка c.

  Решить эту задачу можно, используя знак определителя, соответствующего площади треугольника, вычисленной способом, приведенным выше. Если площадь треугольника $t(a, b, c) > 0$, значит, точка c находится слева от $\overline{ab}$. Если площадь треугольника $t(a, b, c) = 0$, значит, точка c расположена на $\overline{ab}$. Наконец, если площадь треугольника $t(a, b, c) < 0$, значит, точка c находится справа от $\overline{ab}$.
  Этот подход легко обобщается для трех измерений, где знак вычисленного определителя, соответствующего площади, показывает местоположение точки d над ориентированной плоскостью (a, b, c), под плоскостью или на ней;

- *проверка пересечения прямой и отрезка.* Предшествующий способ выяснения расположения точки можно использовать и для проверки пересечения прямой с отрезком. Такое пересечение имеет место тогда и только тогда, когда одна конечная точка отрезка находится слева от прямой, а вторая — справа. Проверка пересечения

отрезков является сходной, но более сложной задачей. Ответ на вопрос, пересекаются ли два отрезка, если они имеют общую концевую точку, зависит от конкретного приложения. Это типичная ситуация, в которой возникают проблемы с вырожденностью;

- *выяснение, находится ли точка внутри круга*. Необходимость в проверке, находится ли точка d внутри круга, определяемого точками а, Ь и с, возникает во всех алгоритмах триангуляции Делоне. Соответствующий алгоритм может быть использован в качестве надежного способа сравнения расстояний. Предполагая, что обозначение точек *a, b,* и *c* идет против часовой стрелки, вычислим определитель *incircle*:

$$\text{incircle}(a,b,c,d) = \begin{vmatrix} a_x & a_y & a_x^2+a_y^2 & 1 \\ b_x & b_y & b_x^2+b_y^2 & 1 \\ c_x & c_y & c_x^2+c_y^2 & 1 \\ d_x & d_y & d_x^2+d_y^2 & 1 \end{vmatrix}$$

Значение определителя *incircle* будет нулевым, если все четыре точки лежат на круге, положительным, если точка *d* лежит внутри круга, и отрицательным, если точка *d* находится вне круга.

Прежде чем пытаться создать свою собственную реализацию, ознакомьтесь с уже существующими, включая указанные в *разделе "Реализации"*.

Реализации. Библиотеки CGAL (**www.cgal.org**) и LEDA (см. *раздел 19.1.1*) содержат реализации алгоритмов вычислительной геометрии, написанные на языке C++. С библиотекой LEDA легче работать, но библиотека CGAL полнее и, к тому же, бесплатна. Если вы разрабатываете серьезное геометрическое приложение, вам непременно следует ознакомиться с возможностями этих библиотек, прежде чем создавать свою собственную реализацию.

В книге [O'R01] представлены реализации на языке С большинства алгоритмов, обсуждаемых в этом разделе. Подробности см. в *разделе 19.1.10*. Хотя эти реализации были созданы, в основном, для ознакомительных целей, а не для коммерческого применения, они надежны и годятся для небольших приложений.

Библиотека Core (**http://cs.nyu.edu/exact/**) содержит API-интерфейс, в котором используется подход EGC (Exact Geometric Computation, точные геометрические вычисления) к созданию численно устойчивых алгоритмов. Эту библиотеку можно использовать с небольшими изменениями в любой программе на языке C/C++, с легкостью поддерживая три уровня точности: машинную точность, произвольную точность и гарантированную точность.

Устойчивую реализацию основных алгоритмов вычислительной геометрии на языке C++, представленную в работе [She97], можно загрузить по адресу **http://www.cs.cmu.edu/~quake/robust.html**.

Примечания

Книга [O'R01] — прекрасное практическое введение в вычислительную геометрию. В ней особое внимание уделяется тщательной и правильной реализации алгоритмов. Библиотека LEDA (см. [MN99]) является еще одним отличным образцом для подражания для тех, кто собирается разрабатывать собственные реализации.

В работе [Yap04] дан замечательный обзор методов построения устойчивых реализаций. Когда писались эти строки, на ее основе готовилась книга (см. [MY07]). В работе [KMP+04] дано графическое представление проблем, возникающих при использовании арифметических операций над действительными числами в алгоритмах вычислительной геометрии, таких как поиск выпуклой оболочки. Управляемое внесение возмущений в данные (см. [MOS06]) является новым подходом к обеспечению устойчивых вычислений, пользующимся значительным вниманием. В работах [She97] и [FvW93] представлены результаты подробных исследований стоимости арифметических операций произвольной точности для геометрических вычислений. Если использовать эти операции с должной осторожностью, можно обеспечить приемлемую эффективность вычислений при их полной устойчивости.

Родственные задачи. Выявление пересечений (см. *раздел 17.8*), конфигурации прямых (см. *раздел 17.15*).

17.2. Выпуклая оболочка

Вход. Множество S из n точек в d-мерном пространстве.

Задача. Найти наименьший выпуклый многоугольник (или многогранник), содержащий все точки множества S (рис. 17.2).

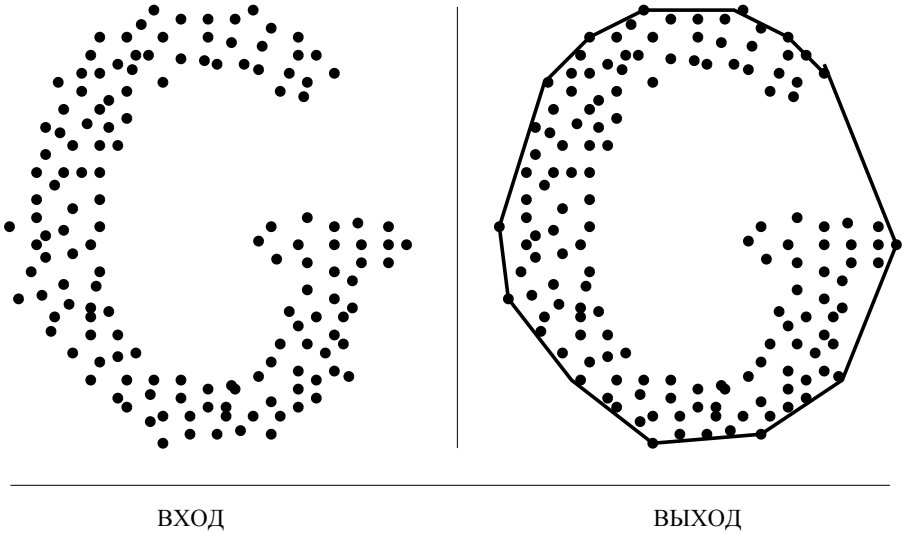


Рис. 17.2. Выпуклая оболочка

Обсуждение. Задача построения выпуклой оболочки набора точек является самой важной элементарной задачей вычислительной геометрии, точно так же, как задача сортировки является самой важной задачей комбинаторных алгоритмов. Важность этой задачи обусловлена тем, что создание выпуклой оболочки позволяет получить приблизительное представление о форме или размере набора данных.

Построение выпуклой оболочки является первым шагом предварительной обработки в большинстве алгоритмов вычислительной геометрии. Рассмотрим, например, задачу поиска диаметра набора точек, иначе говоря, поиска пары точек с наибольшим рас-

стоянием между ними. Диаметр равен расстоянию между двумя точками, расположенными на выпуклой оболочке. Алгоритм вычисления диаметра (имеющий время исполнения $O(n\lg n)$) начинает работу с создания выпуклой оболочки, после чего для каждой вершины оболочки он находит на оболочке вершину, наиболее удаленную от данной. Так называемый "метод штангенциркуля" можно использовать для быстрого перемещения от одного конца диаметра оболочки к другому, двигаясь вдоль оболочки по часовой стрелке.

Существует почти такое же количество алгоритмов для построения выпуклой оболочки, как и алгоритмов сортировки. При выборе наиболее подходящего из них ответьте на следующие вопросы.

- *С каким количеством измерений вы имеете дело?* С выпуклыми оболочками в двух- и даже трехмерном пространстве работать довольно легко. Но некоторые предположения, справедливые для небольшого количества измерений, становятся недействительными при увеличении количества измерений. Например, в двумерном пространстве любой многоугольник с n вершинами имеет ровно n ребер. Но с увеличением количества измерений всего лишь до трех, зависимость между количеством вершин и граней становятся более сложной. Например, куб имеет восемь вершин и шесть граней, а восьмигранник — шесть вершин и восемь граней. Поэтому при выборе структур данных, представляющих оболочки, принципиально важно, требуется ли нам всего лишь найти точки оболочки или мы должны построить многогранник. Следует иметь в виду такие особенности многомерных пространств, если при решении своей задачи вы будете вынуждены работать с ними.

  Существуют простые алгоритмы для построения выпуклой оболочки с временем исполнения $O(n\log n)$ для двумерных и трехмерных случаев. С увеличением размерности задача становится более сложной. Для создания выпуклых оболочек в многомерных пространствах используется базовый алгоритм, называющийся алгоритмом *заворачивания подарка* (gift-wrapping algorithm). Обратите внимание, что трехмерный выпуклый многогранник состоит из двумерных *граней*, соединенных одномерными *ребрами*. Каждое ребро соединяет ровно две грани. Заворачивание подарка начинается с определения начальной грани, связанной с самой нижней вершиной, после чего от этой грани выполняется поиск в ширину, чтобы найти следующие грани. Каждое ребро e, принадлежащее грани, должно быть общим с какой-то другой гранью. Перебрав все n точек, мы можем выяснить, какая точка определяет другую грань ребра e. Образно говоря, мы "оборачиваем" точки по одной грани за раз, сгибая оберточную бумагу по ребрам, пока она не закроет первую точку.

  Для обеспечения эффективности алгоритма необходимо, чтобы каждое ребро исследовалось только один раз. Время исполнения корректно реализованного алгоритма для d измерений равно $O(n\Phi_{d-1} + \Phi_{d-2}\lg\Phi_{d-2})$, где Φ_{d-1} — количество граней, а Φ_{d-2} — количество ребер в выпуклой оболочке. Но если выпуклая оболочка очень сложна, то время исполнения может ухудшиться до $O(n^{\lfloor d/2 \rfloor+1})$. Я рекомендую использовать готовые реализации, а не пытаться разработать свою собственную.

- *Данные представлены в виде вершин или в виде полупространств?* Задача поиска пересечения набора n полупространств в d-мерном пространстве (каждое из которых содержит начало координат) двойственна задаче поиска выпуклых оболочек из

n точек в d-мерном пространстве. Таким образом, для решения обеих задач достаточно одного алгоритма. Требуемое преобразование рассматривается в *разделе 17.15*. Когда внутренняя точка не задана, задача поиска пересечения полуплоскостей перестает быть двойственной задаче поиска выпуклой оболочки, поскольку соответствующий экземпляр не всегда возможен (пересечение полуплоскостей может оказаться пустым).

- *Сколько точек может содержать оболочка?* Если набор точек сгенерирован псевдослучайным образом, большинство точек будет, скорее всего, находиться внутри оболочки. Практическую эффективность программ построения выпуклых оболочек на плоскости можно повысить, используя то обстоятельство, что самая левая, самая правая, самая верхняя и самая нижняя точки должны находиться на выпуклой оболочке. Таким образом, мы имеем набор из трех или четырех разных точек, определяющих треугольник или четырехугольник. Любая точка внутри этой области *не может* находиться на выпуклой оболочке и, следовательно, ее можно отбросить путем линейного перебора точек. В идеале, после этого останется небольшое количество точек, которые можно просмотреть с помощью алгоритма построения выпуклой оболочки.

  Этот прием можно применять в многомерных пространствах, но с увеличением количества измерений его эффективность снижается.

- *Как выяснить форму данного набора точек?* Хотя выпуклая оболочка дает приблизительное представление о форме, многие подробности теряются. Например, выпуклая оболочка для набора точек в форме буквы G (см. рис. 17.2) неотличима от выпуклой оболочки для буквы O. Более общими структурами, которые можно параметризировать, чтобы сохранить "невыпуклость" множества точек, являются *альфа-очертания* (alpha shapes). Ссылки на справочный материал и реализации соответствующих алгоритмов приводятся далее.

Основным алгоритмом построения выпуклой оболочки на плоскости является сканирование по Грэхему. Алгоритм Грэхема начинает работу с точки p, заведомо находящейся на выпуклой оболочке (например, с точки, имеющей самое меньшее значение координаты x), и сортирует остальные точки в порядке возрастания полярного угла, измеряемого против часовой стрелки, относительно этой точки. Начиная с выпуклой оболочки, состоящей из точки p и точки v, имеющей наименьший полярный угол, алгоритм выполняет перебор всех точек вокруг точки v против часовой стрелки с добавлением новых точек в оболочку. Если угол между очередной точкой и последним ребром оболочки меньше, чем 180°, то эта точка добавляется в оболочку. А если этот угол больше 180°, то цепочка вершин, начинающаяся с последнего ребра оболочки, подлежит удалению для сохранения выпуклости. Общее время исполнения алгоритма равно $O(n\lg n)$, т. к. узким местом является сортировка точек вокруг v.

Базовую процедуру алгоритма Грэхема можно использовать для создания простого (не имеющего самопересечений) многоугольника, проходящего через все заданные точки. Для этого сортируем точки вокруг вершины v, но вместо проверки углов соединяем точки в том порядке, в каком они отсортированы. В результате получим многоугольник без самопересечений, но, нередко, с большим количеством некрасивых выступающих частей.

Алгоритм заворачивания подарка значительно упрощается в двумерном пространстве, когда каждая "грань" становится ребром, а каждое "ребро" — вершиной многоугольника, и "поиск в ширину" выполняется вдоль оболочки, по или против часовой стрелки. Время исполнения алгоритма заворачивания подарка для двух измерений равно $O(nh)$, где h — количество вершин в выпуклой оболочке. Я рекомендую использовать алгоритм Грэхема за исключением случаев, когда вы *точно* знаете заранее, что оболочка не содержит слишком много вершин.

Реализации. Библиотека CGAL (**www.cgal.org**) включает в себя реализации на языке C++ разнообразных алгоритмов построения выпуклой оболочки для произвольного количества измерений. Альтернативные реализации на языке C++ алгоритмов построения выпуклых оболочек содержатся в библиотеке LEDA (см. *раздел 19.1.1*).

Популярной реализацией алгоритма построения выпуклых оболочек в пространствах с небольшим количеством измерений является программа Qhull (см.[BDH97]), оптимизированная для работы в диапазоне от двух до восьми измерений. Программа написана на языке C и может также создавать триангуляции Делоне, диаграммы Вороного, а также пересечения полупространств. Программа Qhull широко используется в научных приложениях и доступна на веб-сайте **http://www.qhull.org/**.

В книге [O'R01] дана устойчивая реализация алгоритма Грэхема для двух измерений и реализация с временем исполнения $O(n^2)$ инкрементального алгоритма построения выпуклых оболочек в трех измерениях. Программы написаны на языках C и Java. Подробности см. в *разделе 19.1.10*.

Замечательные программы для построения альфа-очертаний, основанные на работе [EM94], доступны на веб-сайте **http://biogeometry.duke.edu/software/alphashapes/**. Реализацию Hull для построения выпуклых оболочек в многомерных пространствах также можно использовать для построения альфа-очертаний. Программа доступна на веб-сайте **http://www.netlib.org/voronoi/hull.html**.

Для перечисления вершин пересекающихся полупространств с большим количеством измерений требуются другие реализации. Программа lhs (**http://cgm.cs.mcgill.ca/~avis/C/lrs.html**) является арифметически устойчивой реализацией на языке ANSI C алгоритма поиска в обратном направлении Ависа-Фукуды для решения задач перечисления вершин и построения выпуклых оболочек. Так как обход многогранников выполняется неявным образом и они явно не сохраняются в памяти, то иногда можно решить даже задачи с очень большим размером вывода.

Примечания

Роль плоских выпуклых оболочек в вычислительной геометрии сравнима с ролью сортировки в комбинаторике. Подобно сортировке, задача построения выпуклой оболочки является фундаментальной, и разнообразные подходы ведут к созданию интересных или оптимальных алгоритмов. Алгоритмы Quickhull и mergehull являются примерами алгоритмов построения выпуклой оболочки, на создание которых разработчиков вдохновили алгоритмы сортировки (см. [PS85]). Простая конструкция из точек на параболе, рассмотренная в *разделе 9.2.4*, сводит задачу построения выпуклой оболочки к задаче сортировки, так что теоретическая нижняя граница сортировки позволяет сделать вывод, что для построения плоской выпуклой оболочки требуется время $\Omega(n\lg n)$. Более сильная нижняя граница установлена в работе [Yao81].

Подробное обсуждение алгоритмов Грэхема (см. [Gra72]) и Джарвиса (см. [Jar73]) содержится в книгах [dBvKOS00], [CLRS01], [O'R01] и [PS85]. Алгоритм для построения плоской выпуклой оболочки (см. [KS86]) выдает оптимальное решение за время $O(n\lg h)$, где h — количество вершин оболочки, которое обеспечивает наилучшую производительность как алгоритма Грэхема, так и алгоритма заворачивания подарка и является (теоретически) оптимальным вариантом. Вычисление планарных выпуклых оболочек можно эффективно выполнять "на месте", т. е. без выделения дополнительной памяти (см. [BIK+04]). В работе [Sei04] представлен самый свежий обзор алгоритмов построения выпуклой оболочки и их вариантов, в особенности для многомерных пространств.

Альфа-очертания, обсуждаемые в работе [EKS83], дают хорошее представление о форме набора точек. Обобщение оболочек этого типа для трехмерного пространства и соответствующая реализация приведены в работе [EM94].

Алгоритмы поиска в обратном направлении для построения выпуклых оболочек эффективно работают и в многомерных пространствах (см. [AF96]). При удачном построении отображения в пространство с большим количеством измерений (см. [ES86]) задачу создания диаграммы Вороного в d-мерном пространстве можно свести к задаче создания выпуклой оболочки в $(d+1)$-мерном пространстве. Подробности см. в *разделе 17.4*.

Алгоритмы, предназначенные для работы с выпуклыми оболочками, используют специальные структуры данных, которые позволяют выполнять вставку и удаление произвольных точек и при этом непрерывно отображают текущую выпуклую оболочку. Первая из таких динамических структур данных (см. [OvL81]) поддерживала вставки и удаление за время $O(\lg^2 n)$. Позже стоимость этих операций была понижена до почти логарифмического амортизированного времени (см. [Cha01]).

Родственные задачи. Сортировка (см. *раздел 14.1*), диаграммы Вороного (см. *раздел 17.4*).

17.3. Триангуляция

Вход: Набор точек или многогранник.

Задача. Разбить внутреннюю часть набора точек или многогранника на треугольники (рис. 17.3).

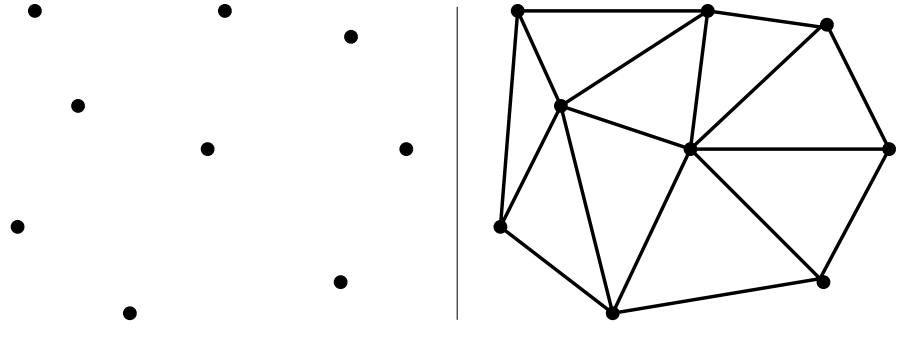


Рис. 17.3. Триангуляция

Обсуждение. Обычно первым шагом в обработке сложных геометрических объектов является разбиение их на простые геометрические фигуры. Это обстоятельство делает триангуляцию фундаментальной задачей вычислительной геометрии. Самым простым двумерным геометрическим объектом является треугольник, а самым простым трехмерным — четырехгранник. В число классических применений триангуляции входят анализ методом конечных элементов и компьютерная графика.

Особенно интересным применением триангуляции является интерполирование поверхности или интерполирование функций. Допустим, что имеются значения высоты горы в некотором множестве точек. Как установить приблизительную высоту горы в заданной точке q? Мы можем проецировать точки, в которых замерена высота, на плоскость, а потом выполнить триангуляцию. Плоскость будет разбита на треугольники, позволяющие приблизительно вычислить высоту точки q путем интерполирования высот вершин треугольника, содержащего эту точку. Кроме того, триангуляция и соответствующие значения высот определяют поверхность горы, пригодную для создания компьютерного изображения.

Триангуляция на плоскости выполняется путем соединения точек непересекающимися отрезками до тех пор, пока добавление новых отрезков не становится невозможным. Вы должны ответить на следующие вопросы.

- *Триангулируется набор точек или многоугольник?* Обычно приходится триангулировать набор точек, как в рассмотренной ранее задаче интерполирования поверхности. Для решения этой задачи нужно сначала создать выпуклую оболочку для данного набора точек, а потом разбить внутренюю область полученной оболочки на треугольники.

  Самый простой алгоритм обработки набора точек, имеющий время исполнения $O(n\lg n)$, сначала сортирует точки по x-координате. Потом отсортированные точки выбираются слева направо, согласно алгоритму построения выпуклой оболочки, описанному в *разделе 4.1*. При этом выполняется триангуляция путем добавления ребра к каждой новой отрезанной от оболочки точке.

- *Имеет ли значение внешний вид треугольников триангуляции?* Обычно вход можно разбить на треугольники многими разными способами. Рассмотрим набор из n точек в выпуклой оболочке на плоскости. Самый простой способ выполнить триангуляцию этих точек — добавить к выпуклой оболочке диагонали, идущие от первой точки до всех остальных. Но в таком случае обычно появляются "узкие" треугольники.

  Во многих приложениях необходимо избегать "узких" треугольников или, что тоже самое, минимизировать количество очень острых углов триангуляции. Триангуляция набора точек методом Делоне минимизирует максимальный угол по всем возможным вариантам триангуляции. Правда, это не совсем то, что нам требуется, но, в принципе, весьма близко. Вообще, триангуляция методом Делоне имеет достаточно много других интересных свойств (включая то, что она является двойственной задаче построения диаграмм Вороного), чтобы я рекомендовал предпочитать ее другим методам триангуляции. Кроме того, используя описанные далее реализации, такую триангуляцию можно выполнить за время $O(n\lg n)$.

- *Как можно улучшить результат триангуляции?* Каждое внутреннее ребро любой триангуляции принадлежит двум треугольникам. Четыре вершины, определяющие эти два треугольника, образуют выпуклый или невыпуклый четырехугольник. Достоинство выпуклого варианта заключается в том, что при замене внутреннего ребра на ребро, связывающее две другие вершины, получается другая триангуляция.

  Иначе говоря, мы имеем локальную операцию замены ребер, посредством которой можно модифицировать и, не исключено, улучшать данную триангуляцию. В самом деле, триангуляцию Делоне можно получить из любой триангуляции, удаляя "узкие" треугольники до тех пор, пока не будут исчерпаны все возможности для локальных улучшений.

- *Какова размерность задачи?* Трехмерные задачи обычно гораздо сложнее, чем двумерные. Обобщение процедуры триангуляции до трех измерений заключается в разбиении пространства на четырехвершинные четырехгранники путем добавления непересекающихся граней. Трудность заключается в том, что некоторые многогранники нельзя разбить на четырехгранники, не добавляя дополнительные вершины. Кроме этого, задача выяснения, существует ли такое разбиение на четырехгранники, является NP-полной, поэтому вы можете смело добавлять новые вершины, чтобы упростить задачу.

- *Каковы ограничивающие условия на входные данные?* При триангуляции многоугольника или многогранника имеется естественное ограничение, запрещающее добавлять ребра, пересекающие какие-либо внешние грани. Вообще говоря, может существовать набор препятствий, которые нельзя пересекать добавляемыми ребрами. Самой лучшей триангуляцией при таких условиях будет, скорее всего, так называемая *триангуляция Делоне с ограничениями*.

- *Можно ли добавлять дополнительные точки или перемещать существующие?* Когда внешний вид треугольников не имеет значения, может оказаться выгодным добавление небольшого количества промежуточных точек в набор данных с целью облегчить создание триангуляции, удовлетворяющей заданному условию, например, требованию отсутствия очень острых углов. Как упоминалось ранее, триангуляцию некоторых многогранников невозможно выполнить без добавления дополнительных точек в исходные данные.

Чтобы выполнить триангуляцию выпуклого многоугольника за линейное время, просто выберем случайную начальную вершину v и соединим ее ребрами со всеми остальными вершинами многоугольника. Так как многоугольник выпуклый, то мы можем быть уверены в том, что добавляемые ребра не будут пересекать стороны многоугольника и что все они будут находиться внутри многоугольника. Самый простой алгоритм для выполнения общей триангуляции многоугольника проверяет каждое из $O(n^2)$ возможных ребер на пересечение с граничными или ранее вставленными ребрами и вставляет его только в случае отсутствия такого пересечения. Существуют пригодные для практического применения алгоритмы с временем исполнения $O(n\lg n)$ и представляющие теоретический интерес алгоритмы с линейным временем исполнения. Подробности см. в *подразделах "Реализации"* и *"Примечания"*.

Реализации. Пакет Triangle, разработанный Джонатаном Шевчуком (Jonathan Shewchuk), содержит набор процедур на языке C для генерирования триангуляций Делоне,

триангуляций Делоне с ограничениями (т. е. триангуляций, в которых некоторые ребра вставляются в принудительном порядке), а также триангуляций Делоне, обеспечивающих качественный результат (т. е. таких, в которых очень острые углы исключаются с помощью вставки промежуточных точек). Этот высокопроизводительный и устойчивый пакет широко используется для анализа методом конечных элементов. Если бы мне требовалась реализация алгоритма двумерной триангуляции, то я начал бы с программ из пакета Triangle. Загрузить пакет можно с веб-страницы **http://www.cs.cmu.edu/~quake/triangle.html**.

Программа Sweep2, написанная на языке C, прекрасно подходит для создания двумерных диаграмм Вороного и триангуляций Делоне. С ней особенно удобно работать, когда вам требуется лишь триангуляция Делоне точек на плоскости. Программа основана на алгоритме заметающей прямой (sweepline alogirthm) для создания диаграмм Вороного (см. [For87]).

Создание сеток для методов конечных элементов является обширной областью для исследований. Веб-страница Стива Оуэна (Steve Owen) Meshing Research Corner, (**http://www.andrew.cmu.edu/user/sowen/mesh.html**) содержит исчерпывающий обзор литературы в этой области, а также ссылки на многочисленные реализации. Особенно рекомендую программу QMG, доступную по адресу **http://www.cs.cornell.edu/Info/People/vavasis/qmg-home.html**.

Как библиотека CGAL (**www.cgal.org**), так и библиотека LEDA (см. *раздел 19.1.1*) содержат реализации на языке C++ самых разных алгоритмов триангуляции в двух- и трехмерных пространствах, включая триангуляции Делоне с ограничениями.

В многомерных пространствах триангуляции Делоне представляют собой частный случай выпуклых оболочек. Популярной реализацией алгоритма построения выпуклых оболочек в пространствах с небольшим количеством измерений является программа Qhull (см. [BDH97]), оптимизированная для работы в диапазоне от двух до восьми измерений. Программа написана на языке C и может также создавать триангуляции Делоне, диаграммы Вороного, а также пересечения полупространств. Программа Qhull широко используется в научных приложениях и доступна на веб-сайте **http://www.qhull.org/**. В качестве альтернативы доступна программа Hull, предназначенная для построения выпуклых оболочек в многомерных пространствах (**http://www.netlib.org/voronoi/hull.html**).

Примечания

После длительных исследований Шазель (Chazelle) создал линейный алгоритм триангуляции простого многоугольника (см. [Cha91]). Но реализация этого алгоритма является исключительно трудной задачей, так что он больше подходит в качестве доказательства существования триангуляционного разбиения. Первый алгоритм триангуляции многоугольников с временем исполнения $O(n\lg n)$ был представлен в работе [GJPT78]. После создания этого алгоритма (но раньше появления алгоритма Шазеля) Тарьян и ван Вик разработали еще один алгоритм с таким же временем исполнения (см. [TW88]). Обзор последних результатов в области триангуляции наборов точек и многоугольников представлен в [Ber04a].

Для исследователей, интересующихся генерированием сеток и решеток, проводится ежегодная конференция International Meshing Roundtable. Отличными обзорами методов генерирования сеток являются работы [Ber02] и [Ede06].

Алгоритмы с линейным временем исполнения для триангуляции монотонных многоугольников давно известны (см. [GJPT78]) и составляют базу для алгоритмов триангуляции простых многоугольников. Монотонным называется многоугольник, для которого существует такое направление, что любая прямая, идущая в этом направлении, пересекает многоугольник, самое большее, в двух точках.

Активно исследуемый класс оптимальных триангуляций включает в себя решения, стремящиеся минимизировать суммарную длину использованных ребер. В работе [MR06] было доказано, что такая задача является NP-полной. После этого внимание исследователей переключилось на доказуемо хорошие аппроксимирующие алгоритмы. Триангуляцию с минимальным весом выпуклого многоугольника можно выполнить за время $O(n^3)$, используя динамическое программирование (см. *раздел 8.6.1*).

Родственные задачи. Диаграммы Вороного (см. *раздел 17.4*), разбиение многоугольников (см. *раздел 17.11*).

17.4. Диаграммы Вороного

Вход. Множество S точек $p_1, ..., p_n$.

Задача. Разбить пространство на области вокруг каждой точки таким образом, чтобы все точки в области вокруг точки p_i были ближе к этой точке, чем к любой другой точке множества S (рис. 17.4).

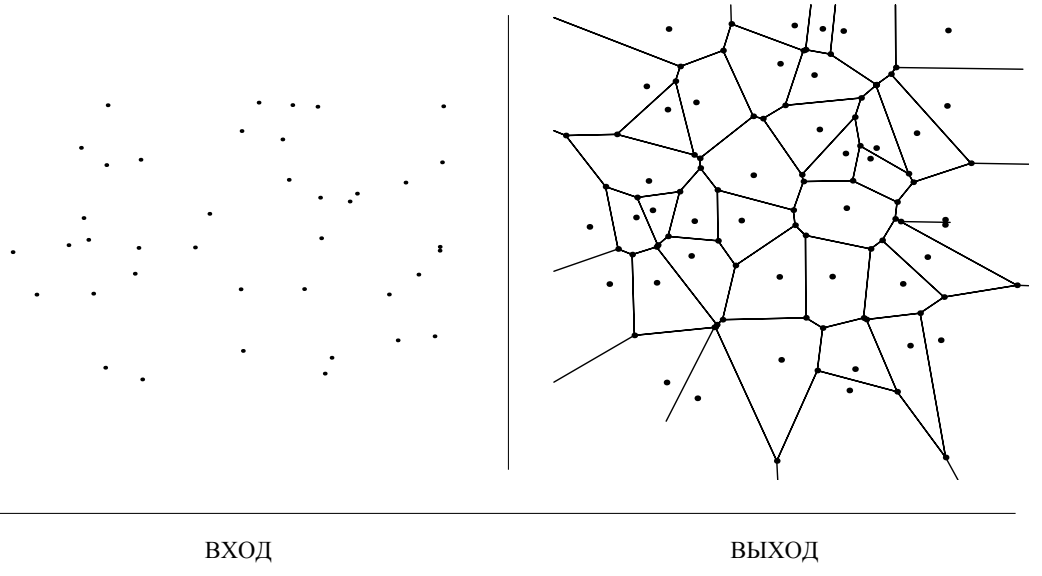


Рис. 17.4. Диаграмма Вороного

Обсуждение. Диаграмма Вороного представляет области влияния вокруг точек из данного набора. Если эти точки соответствуют расположению ресторанов быстрого питания, то диаграмма Вороного разбивает пространство на ячейки, окружающие каждый ресторан. Для человека, живущего в конкретной ячейке, соответствующий ресторан является ближайшим местом, где можно заказать горячий бутерброд.

Диаграммы Вороного имеют множество различных применений:

- *поиск ближайшей точки.* Поиск ближайшего соседа точки q из фиксированного множества S точек сводится к задаче поиска ячейки в диаграмме Вороного, которая содержит данную точку. Подробности см. в *разделе 17.5*;
- *размещение точек обслуживания.* Предположим, что владелец сети ресторанов быстрого питания хочет открыть еще один ресторан. Чтобы свести к минимуму конкуренцию с уже имеющимися в этой местности ресторанами, следует разместить новый ресторан как можно дальше от ближайшего ресторана. Искомое место всегда будет находиться в вершине диаграммы Вороного, и его можно найти за линейное время, выполнив поиск по всем вершинам;
- *наибольший пустой круг.* Допустим, что требуется найти большой неосвоенный земельный участок, чтобы построить фабрику. Условие, определяющее выбор места для ресторана быстрого питания, применимо и при указании месторасположения нежелательных объектов. Иначе говоря, они должны располагаться как можно дальше от любого интересующего нас места. Вершина Вороного определяет центр наибольшего пустого круга вокруг точек;
- *разработка маршрута.* Если объекты множества S являются препятствиями, которых следует избегать, то ребра диаграммы Вороного определяют возможные маршруты с максимальным расстоянием до препятствий. Таким образом, самый "безопасный" путь среди препятствий будет проходить по ребрам диаграммы Вороного;
- *улучшение триангуляции.* Выполняя триангуляцию набора точек, мы обычно хотим получить треугольники без очень острых углов и стараемся избегать "узких" треугольников. Триангуляция Делоне максимизирует минимальный угол по всем возможным триангуляциям. Кроме того, она легко формулируется, как двойственная задача для диаграммы Вороного. Подробности см. в *разделе 17.3*.

Каждое ребро диаграммы Вороного является отрезком серединного перпендикуляра двух точек в множестве S, т. к. это линия, которая делит плоскость посередине между двумя точками. Самым простым методом создания диаграмм Вороного является рандомизированное инкрементальное построение. Чтобы добавить новую точку в диаграмму, мы находим содержащую ее ячейку и добавляем серединные перпендикуляры, которые отделяют эту точку от других, определяющих области, подвергающиеся влиянию. Если точки вставляются в произвольном порядке, то каждая такая вставка повлияет, скорее всего, лишь на небольшое количество областей.

Однако самым лучшим методом является алгоритм Форчуна, основанный на методе заметающей прямой. Его главное достоинство состоит в том, что для него существуют устойчивые реализации. Алгоритм отображает набор точек на плоскости на набор конусов в трехмерном пространстве таким образом, что диаграмма Вороного определяется отображением конусов обратно на плоскость. Преимущества алгоритма Форчуна: оптимальное время исполнения $O(n\log n)$, легкость реализации и отсутствие необходимости сохранять всю диаграмму, если мы можем использовать ее в процессе заметания.

Между выпуклыми оболочками в $(d+1)$-мерном пространстве и триангуляциями Делоне (или, что эквивалентно, диаграммами Вороного) в d-мерном пространстве суще-

ствует интересная связь. Отобразив каждую точку из $E^d(x_1, x_2, ..., x_d)$ на точки $(x_1, x_2, ..., x_d, \sum_{i=1}^{d} x_i^2)$, создав выпуклую оболочку этого $(d+1)$-мерного набора точек, а потом отобразив его обратно в d-мерное пространство, мы получим триангуляцию Делоне. Дополнительная информация содержится в *подразделе "Примечания"*, однако изложенный способ создания диаграмм Вороного в многомерных пространствах является самым лучшим. Программы построения выпуклых оболочек в многомерном пространстве рассматриваются в *разделе 17.2*.

На практике возникает несколько важных вариантов стандартной диаграммы Вороного:

- *неевклидовы метрики расстояний.* Вспомним, что диаграммы Вороного разбивают пространство на зоны влияния вокруг каждой заданной точки. До сих пор мы предполагали, что влияние измеряется евклидовым расстоянием, но в некоторых приложениях это не так. Например, если люди добираются до ресторана на машине, то требуемое время зависит не от расстояния, а от того, как проложены дороги. Существуют эффективные алгоритмы для создания диаграмм Вороного при разных метриках, а также при наличии объектов неправильной формы;
- *диаграммы мощности.* Эти структуры разбивают пространство на зоны влияния вокруг точек, которые могут иметь разную мощность. Рассмотрим, например, сеть радиостанций, работающих на одной частоте. Зона влияния каждой станции зависит как от мощности ее передатчика, так и от расположения и мощности соседних станций;
- *диаграммы k-го порядка и диаграммы для самых дальних точек.* Идею разбиения пространства на зоны с каким-то общим свойством можно распространить за пределы диаграмм Вороного для ближайших точек. Все точки в одной ячейке диаграммы Вороного k-го порядка имеют один и тот же набор ближайших точек из множества S. В диаграммах для самых дальних точек все точки внутри определенной области имеют одну и ту же самую дальнюю точку из множества S. Расположение точек (см. *раздел 17.7*) на этих структурах позволяет быстро находить требуемые точки.

Реализации. Для создания двумерных диаграмм Вороного и триангуляций Делоне широко используется написанная на языке C программа Sweep2. С ней очень просто работать, если вам требуется только диаграмма Вороного. Программа основана на алгоритме заметающей прямой для создания диаграмм Вороного (см. [For87]). Загрузить ее можно с веб-страницы **http://www.netlib.org/voronoi**.

Как библиотека CGAL (**www.cgal.org**), так и библиотека LEDA (см. *раздел 19.1.1*) содержат реализации на языке C++ самых разных алгоритмов для создания диаграмм Вороного и триангуляций Делоне в двух- и трехмерных пространствах.

Диаграммы Вороного в пространствах с большим количеством измерений и диаграммы Вороного для самых дальних точек можно создавать как частный случай выпуклых оболочек в многомерном пространстве. Популярной реализацией алгоритма построения выпуклых оболочек в пространствах с небольшим количеством измерений является программа Qhull (см.[BDH97]), оптимизированная для работы в диапазоне от двух

до восьми измерений. Программа написана на языке C и может создавать триангуляции Делоне, диаграммы Вороного, а также пересечения полупространств. Программа Qhull широко используется в научных приложениях и доступна на веб-сайте **http://www.qhull.org/**. Альтернативным вариантом является программа Hull для создания выпуклых оболочек высокой размерности (**http://www.netlib.org/voronoi/hull.shar**).

Примечания

Современное название диаграммы Вороного получили по имени математика Г. Вороного, который сообщил о них в своем докладе в 1908 г. Однако Дирихле изучал этот вопрос в 1850 г., вследствие чего эти диаграммы иногда называют разбиениями Дирихле. Наиболее полное обсуждение диаграмм Вороного и их приложений содержится в книге [OBSC00]. Отличные обзоры диаграмм Вороного и их вариантов, таких как диаграммы мощности, даются в работах [Aur91] и [For04]. Первый алгоритм создания диаграмм Вороного за время $O(n\lg n)$ основан на методе "разделяй и властвуй". Он приведен в работе [SH75]. Подробное изложение алгоритма Форчуна (см. [For87]) для создания диаграмм Вороного за время $O(n\lg n)$, а также обсуждение связи между триангуляциями Делоне и $(d+1)$-мерными выпуклыми оболочками (см. [ES86]) можно найти в [dBvKOS00] и [O'R01].

Для создания диаграммы Вороного k-го порядка пространство разбивается на области таким образом, что каждая точка данной области находится ближе всего к одному и тому же набору из k точек. Представленный в работе [ES86] алгоритм позволяет создавать диаграммы Вороного k-го порядка за время $O(n^3)$. Выясняя местоположение точек в этой структуре, можно найти k ближайших соседей заданной точки за время $O(k+\lg n)$. Обсуждение диаграмм Вороного k-го порядка содержится в книгах [O'R01] и [PS85].

Задачу наименьшей охватывающей окружности можно решить за время $O(n\lg n)$, используя диаграммы Вороного $(n-1)$-го порядка. Более того, существует алгоритм с линейным временем исполнения, использующий методы линейного программирования в пространствах с небольшим количеством измерений. Линейный алгоритм создания диаграммы Вороного для выпуклого многоугольника приводится в работе [AGSS89].

Родственные задачи. Поиск ближайшего соседа (см. *раздел 17.5*), выяснение местоположения точки (см. *раздел 17.7*), поиск области (см. *раздел 17.6*).

17.5. Поиск ближайшей точки

Входа. Множество S из n точек в d-мерном пространстве; точка q.

Задача. Найти точку в S, ближайшую к точке q (рис. 17.5).

Обсуждение. Во многих геометрических приложениях возникает необходимость быстро найти точку, ближайшую к заданной точке. Классическим примером такой ситуации является разработка диспетчерской системы пожаротушения. Получив вызов, диспетчер должен найти ближайшую к пожару станцию, чтобы минимизировать время прибытия пожарной команды. Аналогичная ситуация возникает в любом приложении, где требуется сопоставление клиентов с поставщиками услуг.

Задача поиска ближайшего соседа также имеет важность в области классификации. Допустим, что у нас имеется база данных избирателей, в которой указаны возраст, рост, вес, образование, уровень дохода и т. п. Про каждого избирателя также известно,

какой из двух политических партий — демократической или республиканской — он симпатизирует. Нам требуется создать классификатор, позволяющий прогнозировать, как данный избиратель будет голосовать. Каждый избиратель в базе данных представлен точкой, помеченной названием партии, в d-мерном пространстве. Простой классификатор можно создать, присваивая новой точке ту же метку, что и у ее ближайшего соседа.

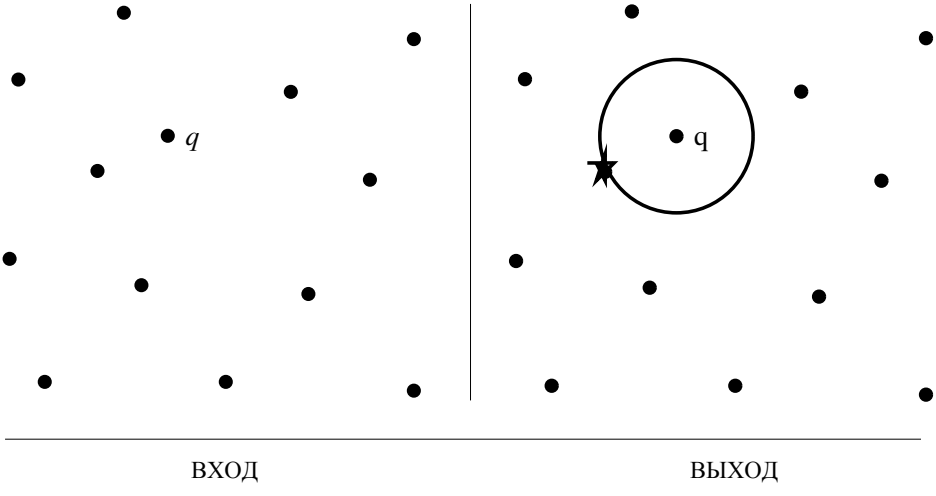


Рис. 17.5. Поиск ближайшей точки

Такие классификаторы, основанные на методе поиска ближайшего соседа, применяются достаточно широко. При сжатии изображений методом векторного квантования изображение разбивается на участки размером 8×8 пикселов. Метод использует заранее составленную библиотеку из нескольких тысяч элементов размером 8×8 пикселов и заменяет каждый участок изображения наиболее похожим элементом из библиотеки. Такой элемент представляется точкой в 64-мерном пространстве, ближайшей к точке, которая представляет рассматриваемый участок изображения. Сжатие, сопровождаемое некоторой потерей качества изображения, достигается заменой 64 пикселов на идентификатор самого похожего библиотечного элемента. Решая задачу поиска ближайшего соседа, ответьте на следующие вопросы.

- *Каков размер пространства поиска?* Если набор данных состоит из небольшого количества точек (скажем, $n \leq 100$) или если планируется небольшое количество запросов, то самый простой подход будет самым лучшим. Сравниваем точку q с каждой из n точек набора данных. Более сложные методы стоит рассматривать только в том случае, когда требуется быстрое выполнение запросов для большого количества точек.
- *Какова размерность пространства?* По мере возрастания количества измерений поиск ближайшего соседа становится все труднее. Представленная в *разделе 12.6* структура данных kd-деревьев очень хорошо подходит для работы в пространствах с небольшим количеством измерений, даже с плоскостями. Тем не менее при количестве измерений, превышающем 20, поиск в kd-деревьях вырождается практически до линейного поиска. Поиск в многомерных пространствах становится трудным по

той причине, что с увеличением размерности шар радиусом r, представляющий все точки на расстоянии $\leq r$ от центра, имеет все меньший объем по сравнению с объемом куба. Таким образом, любая структура данных, основанная на разбиении множества точек на подмножества внутри охватывающих сфер, будет становиться все менее эффективной по мере увеличения количества измерений.

Диаграммы Вороного в двумерном пространстве (см. *раздел 17.4*) предоставляют эффективную структуру данных для поиска ближайшего соседа. Диаграмма Вороного на плоскости разбивает эту плоскость на ячейки таким образом, что каждая ячейка, содержащая точку p, содержит все множество точек, расположенных ближе к точке p, чем к любой другой точке множества S. Задача поиска ближайшего соседа точки q сводится к задаче поиска ячейки диаграммы Вороного, содержащей точку q, и указания точки данных, связанной с этой ячейкой. Хотя диаграммы Вороного можно создавать в многомерных пространствах, с увеличением количества измерений их размер быстро возрастает до такой степени, что они становятся непригодными для использования.

- *Действительно ли нужен ближайший сосед?* Поиск точного ближайшего соседа в многомерном пространстве представляет трудную задачу, для которой, скорее всего, нет решения лучше, чем линейный поиск (т. е. поиск методом исчерпывающего перебора). Но для решения этой задачи существуют эвристические алгоритмы, которые могут довольно быстро найти точку, расположенную достаточно близко от заданной точки.

  Один из подходов заключается в *понижении размерности* (dimension reduction). Существуют способы отображения любого набора из n точек в d-мерном пространстве на пространство, имеющее $d' = O(\lg n/\varepsilon^2)$ измерений, таким образом, что расстояние до ближайшего соседа в пространстве с меньшим количеством измерений превышает расстояние до действительно ближайшего соседа примерно в $(1 + \varepsilon)$ раз. Отображение точек на произвольную гиперплоскость, имеющую d' измерений, в пространстве E^d, скорее всего, принесет желаемые результаты.

  Альтернативным подходом является внесение элемента случайности при поиске в структуре данных. Структура данных kd-деревьев позволяет выполнять эффективный поиск ячейки, содержащей точку q, т. е. ячейки, граничные точки которой являются хорошими кандидатами на роль близких соседей. Теперь допустим, что мы ищем точку q', которая является результатом небольшого случайного перемещения точки q. Мы попадем в другую ячейку поблизости, одна из граничных точек которой может оказаться еще более близким соседом точки q. Многократное повторение подобных рандомизированных запросов позволит нам эффективно потратить на улучшение результата столько времени, сколько мы можем себе позволить.

- *Набор данных является статическим или динамическим?* Будут ли в вашем приложении выполняться операции вставки или удаления точек данных? Если вставка и удаление выполняются нечасто, то после каждой такой операции имеет смысл заново строить структуру данных. В противном случае следует использовать версию kd-дерева, поддерживающую операции вставки и удаления.

Граф ближайшего соседа для множества S из n точек связывает каждую вершину со своим ближайшим соседом. Такой граф является подграфом триангуляции Делоне, и

поэтому его можно вычислить за время $O(n\log n)$. Это довольно хороший результат, т. к. только поиск ближайшей пары точек множества S занимает время $\Theta(n\log n)$.

Задача поиска пары ближайших точек сводится к сортировке в одномерном пространстве. В отсортированном наборе чисел пара ближайших точек соответствует двум числам, расположенным рядом друг с другом, поэтому нам нужно только найти минимальное расстояние между $n-1$ смежными точками. Предельный случай возникает, когда расстояние между ближайшими точками равно нулю, откуда следует, что элементы не являются уникальными.

Реализации. Библиотека ANN на языке C++ содержит реализации алгоритмов для точного и аппроксимирующего поиска ближайшего соседа в пространствах произвольной размерности. Эти реализации дают хорошие результаты для поиска среди сотен тысяч точек в пространствах, имеющих до 20 измерений. Библиотека поддерживает все L^p-нормы, включая евклидово и манхэттенское расстояния. Загрузить библиотеку можно по адресу **http://www.cs.umd.edu/~mount/ANN/**. Если бы мне пришлось решать задачу поиска ближайшего соседа, я начал бы с этой библиотеки.

Апплеты Java из коллекции Spatial Index Demos (**http://donar.umiacs.umd.edu/quadtree/**) иллюстрируют различные варианты kd-деревьев. Алгоритмы, реализуемые в апплетах, рассматриваются в книге [Sam06]. Программа KDTREE 2 содержит реализации kd-деревьев на C++ и FORTRAN 95 для эффективного поиска ближайшего соседа в многомерных пространствах. Подробности см. по адресу **http://arxiv.org/abs/physics/0408067**.

Программа Ranger (см. [MS93]) представляет собой инструмент для визуализации и экспериментов с поиском ближайшего соседа и поиском в ортогональных областях в наборах данных с большим количеством измерений с использованием многомерных деревьев поиска. Программа поддерживает четыре разных типа структур данных: примитивные kd-деревья, медианные kd-деревья, неортогональные kd-деревья и VP-деревья. Программу можно загрузить из хранилища алгоритмов по адресу **http://www.cs.sunysb.edu/~algorith**.

Специализированная программа Nearpt3 предназначена для поиска ближайшего соседа в очень больших наборах точек в трехмерном пространстве. Код можно загрузить с веб-страницы **http://wrfranklin.org/Research/nearpt3**.

В *разделе 17.4* представлена информация о полной коллекции реализаций алгоритмов построения диаграмм Вороного. В частности, библиотеки CGAL (**www.cgal.org**) и LEDA (см. *раздел 19.1.1*) содержат реализации на языке C++ алгоритмов построения диаграмм Вороного, а также алгоритмов выяснения местоположения точек на плоскости для их эффективного использования при поиске ближайшего соседа.

Примечания

В работе [Ind04] представлен отличный обзор алгоритмов, основанных на методе случайных отображений для аппроксимирующего поиска ближайшего соседа в многомерных пространствах. Как теоретические, так и экспериментальные результаты (см. [IM04] и [BM01] соответственно) указывают на то, что этот метод обладает довольно высокой точностью.

Несколько иную теоретическую основу имеет программа ANN для аппроксимирующего поиска ближайшего соседа (см. [AM93, AMN+98]). На основе набора данных создается

структура разреженного взвешенного графа, после чего поиск ближайшего соседа осуществляется выполнением "жадного" обхода от произвольной точки до точки запроса. Ближайшая точка, найденная в результате нескольких таких попыток, объявляется ближайшим соседом. Подобные структуры данных оказываются полезными и при решении других задач в многомерных пространствах. Предметом пятых соревнований DIMACS был поиск ближайшего соседа (см. [GJM02]).

Самым лучшим справочником по kd-деревьям и другим пространственным структурам данных является книга [Sam06]. В ней подробно рассмотрены все основные (и многие второстепенные) варианты этих структур. Книга [Sam05] представляет собой краткий обзор той же темы. Метод случайных смещений точки запроса был представлен в работе [Pan06].

Хорошие описания задачи поиска пары ближайших точек на плоскости (см. [BS76]) можно найти в книгах [CLRS01] и [Man89]. Вместо простого выбора точек из триангуляции Делоне в этих алгоритмах применяется метод "разделяй и властвуй".

Родственные задачи. Kd-деревья (см. *раздел 12.6*), диаграммы Вороного (см. *раздел 17.4*), поиск в области (см. *раздел 17.6*).

17.6. Поиск в области

Вход. Множество S из n точек в пространстве E^d и область Q.

Задача. Выяснить, какие точки множества S находятся в области Q (рис. 17.6).

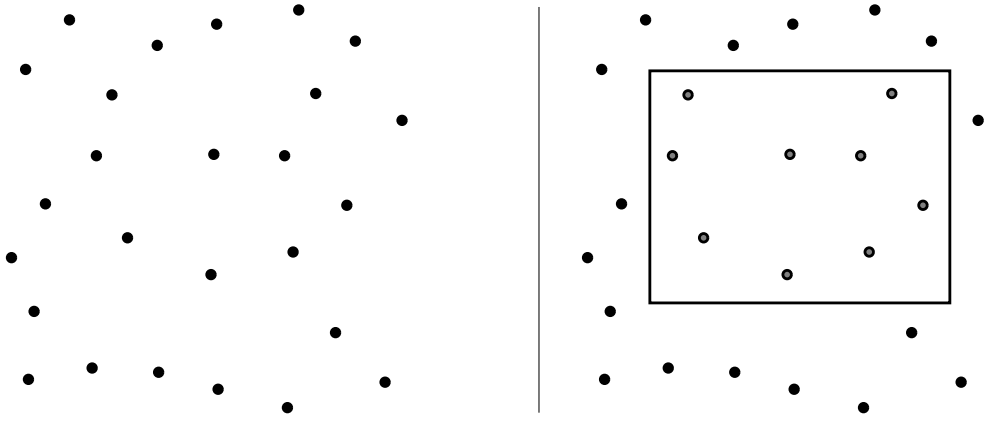


Рис. 17.6. Поиск в области

Обсуждение. Задачи поиска в области возникают в приложениях баз данных и географических информационных систем. Любой объект базы данных, имеющий d числовых полей (и описывающий, например, человека с его ростом, весом, уровнем дохода и т. д.), можно представить в виде точки в d-мерном пространстве. *Область запроса* описывает область в пространстве, а наша задача состоит в поиске всех точек или выяснении количества точек в этой области. Например, запрос на поиск всех людей с го-

довым доходом от 0 до 10 000 долларов, ростом между 185 и 215 см и весом от 25 до 60 кг определяет область, содержащую тощих людей с тощим кошельком.

Уровень трудности поиска в области зависит от нескольких факторов.

- *Велико ли количество выполняемых запросов?* Самый простой подход — проверка каждой из n точек на попадание в многоугольник Q. Этот метод прекрасно работает при небольшом количестве запросов. Алгоритмы для проверки вхождения точки в данный многоугольник рассматриваются в *разделе 17.7*.
- *Какова форма многоугольника?* Легче всего выполнять поиск в прямоугольниках, параллельных осям координат, т. к. проверка точки на вхождение в такую область сводится к проверке попадания каждой координаты в требуемый диапазон. Пример поиска в ортогональной области показан на рис. 17.6.

  При поиске в невыпуклом многоугольнике полезно разбить его на выпуклые области или, что еще лучше, треугольники и выполнить поиск в каждой из этих областей. Этот метод хорошо работает, потому что проверка точки на вхождение в выпуклый многоугольник является достаточно простой задачей. Алгоритмы разбиения невыпуклого многоугольника на несколько выпуклых многоугольников рассматриваются в *разделе 17.11*.
- *Какова размерность пространства?* Общий подход к поиску в области состоит в создании kd-дерева для входного набора точек (см. *раздел 12.6*). Затем выполняется обход в глубину этого kd-дерева, причем каждый узел дерева разворачивается только тогда, когда соответствующий прямоугольник пересекает область запроса. В случае очень больших или смещенных областей запроса может потребоваться обход всего дерева, но, вообще говоря, kd-деревья дают эффективное решение. Для двумерного пространства известны алгоритмы с более высокой производительностью в наихудшем случае, но на плоскости производительность kd-деревьев является вполне приемлемой. А в многомерных пространствах поиска они являются единственным возможным решением задачи.
- *Возможны ли операции вставки и удаления?* Красивый и практичный подход к решению задачи поиска в области и ряда других геометрических задач поиска основан на триангуляциях Делоне. В триангуляции Делоне ребра соединяют каждую точку p с близлежащими точками. Для поиска в области мы сначала выясняем местоположение точки на плоскости (см. *раздел 17.7*), чтобы быстро найти треугольник в интересующей нас области. Потом мы выполняем поиск в глубину вокруг какой-либо вершины этого треугольника, разрежая пространство поиска в каждом случае, когда посещаем точку, расположенную слишком далеко, чтобы иметь неоткрытых соседей, представляющих для нас интерес. Эта процедура должна быть эффективной, т. к. общее количество посещенных точек приблизительно пропорционально количеству точек в области запроса.

  Одним из достоинств этого подхода является легкость, с которой можно откорректировать триангуляцию Делоне после вставки или удаления точки, используя для этого операции замены ребер. Подробности см. в *разделе 17.3*.
- *Можно ли ограничиться подсчетом количества точек в области или требуется идентифицировать их?* Для многих приложений достаточно только подсчитать ко-

личество искомых точек в области, а не возвращать сами точки. Например, из базы данных, упомянутой в начале раздела, мы можем узнать, каких людей больше, худых и бедных или толстых и богатых. Часто возникает необходимость найти самую плотную или самую разреженную область в пространстве поиска, и эту задачу можно решить с помощью подсчета точек в ней.

Хорошая структура данных для эффективного поиска ответов на составные запросы поиска в области основана на упорядочении набора точек по признаку доминирования. Говорят, что точка x доминирует над точкой y, если точка y располагается снизу и слева от точки x. Пусть DOM(p) является функцией, которая подсчитывает количество точек во множестве S, над которыми доминирует точка p. Количество точек m в ортогональном прямоугольнике, определяемом координатами $x_{\min} \le x \le x_{\max}$ и $y_{\min} \le y \le y_{\max}$, вычисляется по такой формуле:

$$m = DOM(x_{\max}, y_{\max}) - DOM(x_{\max}, y_{\min}) - DOM(x_{\min}, y_{\max}) + DOM(x_{\min}, y_{\min})$$

Четвертый член вносит поправку на точки нижнего левого угла, которые были вычтены дважды.

Чтобы эффективно отвечать на произвольные запросы по выяснению доминирования, разделяем пространство на n^2 прямоугольников, проводя горизонтальную и вертикальную прямые через каждую из n точек. В любом прямоугольнике набор доминируемых точек одинаков для каждой точки, поэтому их можно вычислить заранее для нижнего левого угла каждого прямоугольника, сохранить и возвращать в ответ на запрос для любой точки в данном прямоугольнике. Таким образом, поиск в области сводится к двоичному поиску и выполняется за время $O(\lg n)$. К сожалению, эта структура данных имеет квадратичную сложность по памяти. Но эту идею можно использовать с kd-деревьями, чтобы создать структуру, расходующую память более эффективно.

Реализации. Библиотеки CGAL (**www.cgal.org**) и LEDA (см. *раздел 19.1.1*) используют динамическую структуру данных на основе триангуляции Делоне для поддержки круглых, треугольных и ортогональных областей поиска. Обе библиотеки также содержат реализации древовидных структур, которые поддерживают поиск в ортогональных областях за время $O(k + \lg^2 n)$, где n — сложность разбиения, k — количество точек в прямоугольной области.

Библиотека ANN на языке C++ содержит реализации алгоритмов для точного и аппроксимирующего поиска ближайшего соседа в пространствах произвольной размерности. Эти реализации дают хорошие результаты для поиска среди сотен тысяч точек в пространствах, имеющих до 20 измерений. Библиотека позволяет выполнять запросы поиска ближайшего соседа постоянного радиуса по всем L^p-нормам расстояний, которые можно использовать для аппроксимации запросов поиска круглых и прямоугольных областей в нормах L^1 и L^2 соответственно. Библиотеку ANN можно загрузить по адресу **http://www.cs.umd.edu/~mount/ANN/**.

Программа Ranger (см. [MS93]) представляет собой инструмент для визуализации и экспериментов с поиском ближайшего соседа и поиском в ортогональных областях в наборах данных с большим количеством измерений с использованием многомерных деревьев поиска. Программа поддерживает четыре разных типа структур данных: примитивные kd-деревья, медианные kd-деревья, неортогональные kd-деревья и

VP-деревья. Для каждой из этих структур данных программа поддерживает запросы в пространствах, имеющих до 25 измерений, в любой метрике Минковского. Программу можно загрузить из хранилища алгоритмов по адресу **http://www.cs.sunysb.edu/~algorith**.

Примечания

Хорошие описания структур данных с временем исполнения $O(\lg n + k)$ в наихудшем случае для поиска в ортогональных областях (см. [Wil85]) представлены в книгах [dBvKOS00] и [PS85]. Там же можно найти описания kd-деревьев, применяемых при поиске точек в прямоугольниках на плоскости. Производительность программ, использующих эти деревья, может быть очень низкой. Так, в работе [LW77] описывается двумерный экземпляр запроса, для которого потребовалось время $O\sqrt{n}$, чтобы выяснить, что прямоугольник пустой.

Задача значительно усложняется при поиске в неортогональных областях, т. е. в областях, не являющихся прямоугольниками со сторонами, параллельными осям координат. Поиск пересечений полуплоскостей имеет сложность по времени $O(\lg n)$, а по памяти — линейную (см. [CGL85]). В случае поиска в простых областях (таких как треугольник) нижние границы препятствуют созданию структур, эффективных для наихудших случаев. Обсуждение этой темы и обзор посвященных ей работ представлены в [Aga04].

Родственные задачи. Kd-деревья (см. *раздел 12.6*), выяснение местоположения точки (см. *раздел 17.7*).

17.7. Местоположение точки

Вход. Плоскость, разбитая на многоугольники, и точка q.

Задача. Выяснить, какая область плоскости содержит точку q (рис. 17.7).

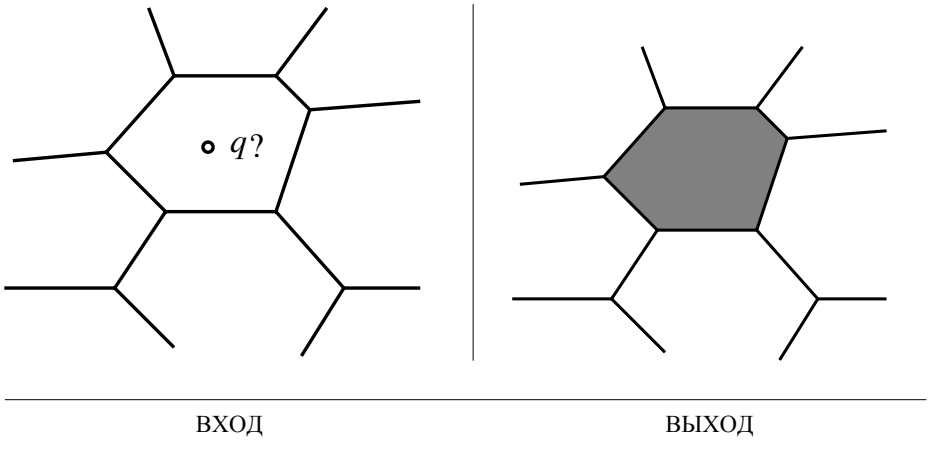


Рис. 17.7. Выяснение местоположения точки

Обсуждение. Выяснение местоположения точки на плоскости является фундаментальной подзадачей вычислительной геометрии, обычно возникающей, как составная часть решения больших геометрических задач. В типичной полицейской диспетчерской

службе город разбивается на несколько участков (районов). Имея карту участков и точку запроса (место преступления), диспетчер должен найти участок, содержащий данную точку. Это и есть задача выяснения местоположения точки на плоскости. Возможны следующие варианты задачи:

- *Находится ли данная точка внутри многоугольника P?* В самом простом варианте задачи выяснения местоположения точки на плоскости участвуют только две области, внутри и вне многоугольника P, и требуется выяснить, в какой из них находится данная точка. Для невыпуклых многоугольников, имеющих много узких выступов, поиск ответа может оказаться очень трудным. Процедура получения решения состоит в следующем. Из точки проводим отрезок, заканчивающийся далеко за пределами многоугольника, и подсчитываем количество ребер многоугольника, пересекаемых этим отрезком. Точка будет находиться внутри многоугольника тогда и только тогда, когда это число нечетное. Когда отрезок проходит через вершину, а не ребро, ответ очевиден из контекста, т. к. мы подсчитываем количество пересечений границы многоугольника. Проверка всех n сторон многоугольника на пересечение с отрезком занимает время $O(n)$. Для выпуклых многоугольников существуют более быстрые алгоритмы на основе двоичного поиска со временем исполнения $O(\lg n)$.
- *Сколько потребуется запросов?* Выяснение местоположения точки внутри многоугольника всегда можно производить отдельно для каждой области в данном разбиении. Однако выполнение большого количества таких запросов на одном и том же разбиении неэкономично. Намного лучшим решением будет создание поверх этого разбиения решеточной или древовидной структуры данных, позволяющей быстро попасть в нужную область. Такие структуры подробно рассматриваются далее в этом разделе.
- *Какова сложность областей разбиения?* Если области, на которые разбита плоскость, являются произвольными многоугольниками, требуются довольно сложные проверки на вхождение точки в область. Выполнив предварительную триангуляцию всех многоугольников разбиения, мы сведем проверку на вхождение точки в область к проверке на вхождение точки в треугольник. Эти проверки можно сделать очень быстрыми и простыми за счет небольших затрат на сохранение имени многоугольника для каждого треугольника. Дополнительная выгода от такой триангуляции заключается в том, что чем меньше размер областей разбиения, тем выше производительность решеточных или древовидных суперструктур. Но при триангуляции следует соблюдать определенную осторожность, чтобы избежать получения треугольников с очень острыми углами (см. *раздел 17.3*).
- *Насколько регулярны размер и форма областей разбиения?* Если все треугольники имеют приблизительно одинаковый размер и форму, можно использовать самый простой способ выяснения местоположения точки, при котором на область разбиения накладывается решетка размером $k \times k$ из горизонтальных и вертикальных прямых, идущих через одинаковые интервалы. Для каждой из k^2 прямоугольных ячеек поддерживается список всех областей разбиения, которые хотя бы частично покрываются данной ячейкой. Для выяснения местоположения точки в таком сеточном файле выполняется двоичный поиск или поиск в хэш-таблице, чтобы найти ячейку, содержащую точку q, а потом выполняется поиск в каждой области, покрываемой ячейкой, чтобы выяснить, какая из них содержит данную точку.

Производительность таких сеточных файлов может быть очень хорошей, при условии, что каждая треугольная область перекрывает лишь небольшое количество прямоугольников (тем самым минимизируя потребности в памяти), а каждый прямоугольник перекрывает лишь небольшое количество треугольников (тем самым минимизируя время поиска). Производительность этого метода зависит от регулярности областей разбиения. Определенной гибкости можно добиться, располагая горизонтальные линии решетки не на одинаковых расстояниях друг от друга, а в зависимости от расположения областей разбиения. Рассматриваемый далее метод полос является развитием этой идеи и гарантирует эффективную временную сложность за счет квадратичной сложности по памяти.

- *Какова размерность пространства?* Для трех и более измерений наиболее подходящим методом выяснения местоположения точки почти наверняка будет использование kd-дерева. Эти деревья могут оказаться лучшим решением, когда плоскость разбита на области, слишком нерегулярные, чтобы можно было применить сеточные файлы.

  Kd-деревья (см. *раздел 12.6*) иерархически разбивают пространство на прямоугольные блоки. В каждом узле дерева текущий прямоугольник разбивается на небольшое количество (обычно 2, 4 или 2^d для d измерений) меньших прямоугольников. Прямоугольник, принадлежащий листу, помечается списком областей, содержащихся в нем хотя бы частично. Процесс выяснения местоположения точки начинается с корня дерева и движется вниз по дереву, пока не будет найден прямоугольник, который содержит точку q. Когда поиск доходит до листа, проверяется каждая содержащаяся в нем подходящая область, чтобы выяснить, какая из них включает в себя точку q. Так же, как и в случае с сеточными файлами, мы надеемся, что каждый лист будет содержать небольшое количество областей, и что каждая область не пересекает слишком много листовых прямоугольников.

- *Близко ли целевая ячейка?* Простым методом выяснения местоположения точки, который хорошо работает в пространствах, имеющих гораздо больше двух измерений, является обход. Начинаем поиск с произвольной точки p в произвольной ячейке, расположенной (предположительно) недалеко от точки q. Строим луч от p к q и находим сторону (грань) ячейки, пересекаемую этим лучом. В триангуляционных разбиениях такие запросы выполняются за постоянное время.

  Переходя к следующей ячейке сквозь эту грань, мы еще на шаг приблизимся к цели. Для достаточно регулярных d-мерных разбиений ожидаемая длина пути будет $O(n^{1/d})$, но линейной в наихудшем случае.

Самым простым алгоритмом, гарантирующим время поиска $O(\lg n)$ в наихудшем случае, является *метод полос*, в котором через каждую вершину проводятся горизонтальные прямые, создающие $n + 1$ полос между ними. Поскольку полосы определяются горизонтальными прямыми, полосу, содержащую конкретную точку, можно найти, выполнив двоичный поиск по y-координате точки q. Поскольку никакая полоса не содержит вершин, то область в полосе, содержащую точку q, можно найти с помощью двоичного поиска по ребрам, пересекающим эту полосу. Некоторая трудность заключается в том, что для каждой полосы необходимо сопровождать двоичное дерево поиска, что в наихудшем случае дает сложность по памяти, равную $O(n^2)$. Метод, расхо-

дующий память более эффективно, основан на создании иерархической триангуляционной структуры на областях разбиения. Он тоже обеспечивает сложность по времени, равную $O(\lg n)$, и рассматривается в *подразделе "Примечания"*.

Эффективные методы вычислительной геометрии для наихудших случаев либо требуют большого объема памяти, либо очень сложны для реализации. Но для большинства приложений рекомендуется использовать kd-деревья.

Реализации. Как библиотека CGAL (**www.cgal.org**), так и библиотека LEDA (см. *раздел 19.1.1*) содержат реализации на языке C++ самых разных алгоритмов разбиения плоскости. В библиотеке CGAL отдается предпочтение стратегии "переход и обход" (jump-and-walk), хотя также представлена реализация алгоритма поиска с логарифмической сложностью для наихудшего случая. Библиотека LEDA содержит программы выяснения местоположения точки с использованием частично устойчивых (partially persistent) деревьев поиска с ожидаемым временем исполнения $O(\lg n)$.

Библиотека ANN на языке C++ содержит реализации алгоритмов для точного и аппроксимирующего поиска ближайшего соседа в пространствах произвольной размерности. Программы из этой библиотеки можно использовать для быстрого поиска точки на границе ближайшей ячейки, с которой следует начинать обход. Загрузить библиотеку можно по адресу **http://www.cs.umd.edu/~mount/ANN/**.

Пакет Arrange на языке C предназначен для размещения многоугольников на плоскости или на сфере. Многоугольники могут быть вырожденными, и в таком случае работа сводится к размещению отрезков. В пакете используется рандомизированный инкрементальный алгоритм и поддерживается эффективное выяснение местоположения точки в разбиении. Пакет Arrange был разработан Майклом Голдвассером (Michael Goldwasser) и доступен на веб-сайте **http://euler.slu.edu/~goldwasser/publications**.

В книгах [O'R01] и [SR03] представлены процедуры проверки расположения точки в простом многоугольнике.

Примечания

Отличный обзор последних достижений в области решения задачи выяснения местоположения точки, как теоретических, так и практических, представлен в [Sno04]. Очень подробные описания детерминистических структур данных, используемых при выяснении местоположения точки на плоскости, содержатся в книгах [dBvKOS00] и [PS85].

В работе [TV01] задача выяснения местоположения точки на плоскости используется в качестве учебного примера разработки алгоритмов вычислительной геометрии на языке Java. Экспериментальное исследование алгоритмов выяснения местоположения точки на плоскости изложено в работе [EKA84]. Лучшим был признан метод группирования, аналогичный методу сеточного файла.

Элегантный метод улучшения треугольников (см. [Kir83]) заключается в создании иерархической структуры триангуляционных разбиений поверх имеющегося разбиения таким образом, что каждый треугольник на данном уровне пересекает постоянное количество треугольников следующего уровня. Так как размер каждой триангуляции составляет определенную долю от размера следующей триангуляции, то общая сложность по памяти вычисляется суммированием геометрической прогрессии и, следовательно, является линейной. Кроме этого, высота иерархической структуры равна $O(\lg n)$, что обеспечивает быстрое время исполнения запросов. Альтернативный алгоритм с такими же временными

пределами изложен в работе [EGS86]. Описанный ранее метод полос был разработан Добкином (Dobkin) и Липтоном (Lipton) (см. [DL76]) и представлен в книге [PS85]. Описания алгоритмов, проверяющих вхождение точки в простой многоугольник, содержатся в работах [Hai94], [O'R01], [PS85] и [SR03].

В последнее время наблюдается интерес к динамическим структурам данных, которые поддерживают быстрое инкрементальное обновление разбиения плоскости (после вставки и удаления ребер и вершин), а также быстрое выяснение местоположения точки. Изучение этой области можно начать с [CT92], а обновленный справочный материал можно найти в [Sno04].

Родственные задачи. Kd-деревья (см. *раздел 12.6*), диаграммы Вороного (см. *раздел 17.4*), поиск ближайшего соседа (см. *раздел 17.5*).

17.8. Выявление пересечений

Вход. Множество S отрезков (прямых) l_1, ..., l_n или пара многоугольников (многогранников) P_1 и P_2.

Задача. Выяснить, какие отрезки пересекаются. Найти пересечение объектов P_1 и P_2 (рис. 17.8).

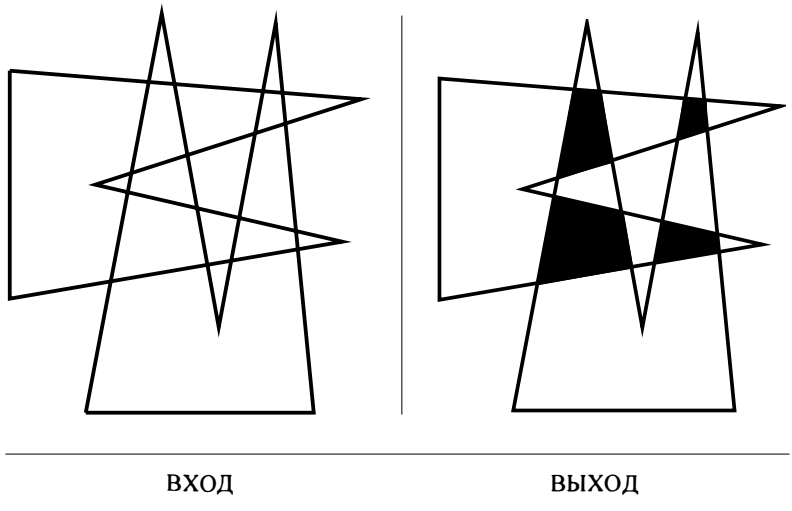


Рис. 17.8. Выявление пересечений

Обсуждение. Выявление пересечения является фундаментальной задачей вычислительной геометрии, имеющей много применений. Например, предположим, что мы моделируем здание в виртуальной реальности компьютерной игры. Иллюзия реальности исчезнет, как только виртуальный персонаж пройдет сквозь стену. Чтобы обеспечить соблюдение физических ограничений, мы должны немедленно обнаруживать пересечение многогранных моделей и информировать о нем игрока или ограничивать его действия.

Еще одним применением выявления пересечений является контроль проектирования СБИС. Небольшой дефект конструкции, такой как пересечение двух дорожек, может

вызвать короткое замыкание микросхемы, но такие ошибки можно с легкостью выявить до сдачи проекта в производство, используя программы для выявления пересечения отрезков.

Вы должны ответить на следующие вопросы, возникающие в задаче выявления пересечений.

- *Нужно вычислить местоположение пересечения или достаточно лишь выявить сам факт его существования?* Задача выявления пересечения решается значительно легче, чем вычисление его местоположения, и во многих случаях этого достаточно. Для приложений виртуальной реальности важным может быть только сам факт столкновения со стеной, а не координаты точки, в которой это произойдет.
- *Нужно выявить пересечение прямых или отрезков?* Разница состоит в том, что любые две непараллельные прямые пересекаются в одной и только одной точке. Все точки пересечения прямых можно вычислить за время $O(n^2)$, сравнив каждую пару прямых. Как показано в *разделе 17.15*, создание конфигурации прямых предоставляет больше информации, чем просто поиск точек пересечения.

  Поиск всех точек пересечения n отрезков является значительно более трудной задачей. Даже проверка двух отрезков на пересечение оказывается не такой уж простой операцией (см. *раздел 17.1*). Конечно, можно было бы явно проверить каждую пару отрезков и таким образом найти все пересечения за время $O(n^2)$, но для случаев с небольшим количеством точек пересечения существуют более быстрые алгоритмы.
- *Каково ожидаемое количество точек пересечения?* При контроле проектирования СБИС мы ожидаем, что набор отрезков будет иметь небольшое количество точек пересечения, если, вообще, они найдутся. В этом случае нам нужен алгоритм, чувствительный к выводу, т. е. имеющий время исполнения, пропорциональное количеству точек пересечения.

  Такие чувствительные к выводу алгоритмы для выявления пересечений отрезков уже существуют. Время исполнения самого быстрого из них равно $O(n\lg n + k)$, где k — количество пересечений. Эти алгоритмы основаны на методе заметающей прямой.
- *Видна ли точка x из точки y?* В некоторых случаях требуется узнать, можно ли в заполненном препятствиями пространстве видеть точку y из точки x. Эту задачу можно сформулировать в виде задачи выявления пересечения отрезков: пересекает ли отрезок между точками x и y какое-либо препятствие? Задачи выяснения видимости возникают при планировании перемещений роботов (см. *раздел 17.14*) и при исключении скрытых поверхностей в компьютерной графике.
- *Являются ли пересекающиеся объекты выпуклыми?* Существуют очень хорошие алгоритмы для выявления пересечения многоугольников. В этой ситуации большую важность приобретает вопрос о выпуклости многоугольников. Пересечение выпуклого n-угольника с выпуклым m-угольником можно выявить за время $O(n+m)$ с помощью алгоритма, основанного на методе заметающей прямой, описанного далее. Это возможно благодаря тому, что результат пересечения двух выпуклых многоугольников — выпуклый многоугольник, содержащий, самое большее, $n+m$ вершин.

Однако пересечение двух невыпуклых многоугольников лишено этих достоинств. Рассмотрим пересечение двух "расчесок", показанное на рис. 17.8. Как видно из рисунка, пересечение невыпуклых многоугольников может быть фрагментированным.

Задача поиска пересечения многогранников сложнее задачи поиска пересечения многоугольников, т. к. многогранники могут пересекаться даже тогда, когда их ребра не пересекаются. В качестве примера такой ситуации рассмотрим иголку, пронзающую внутреннюю область грани. Тем не менее, при поиске пересечения как многоугольников, так и многогранников возникают одинаковые вопросы.

- *Выполняется ли многократный поиск пересечений с одними и теми же основными объектами?* В примере с прохождением сквозь стену в виртуальном здании неподвижные объекты не изменяются от эпизода к эпизоду. Движется только виртуальный персонаж, а пересечения происходят редко.

  В таких случаях типичным подходом будет аппроксимация объектов более простыми охватывающими объектами, например, параллелепипедами. Пересечение двух таких охватывающих объектов означает, что содержащиеся в них объекты *могут* пересекаться, и для выяснения этого вопроса требуется дополнительная обработка. Но проверка на пересечение простых параллелепипедов происходит намного эффективнее, чем более сложных объектов, поэтому в случае небольшого количества пересечений этот метод выгоден. Возможны различные варианты этой схемы, но основная идея позволяет заметно повысить производительность в сложной обстановке.

Для эффективного поиска пересечений нескольких отрезков или пересечений и объединений двух многоугольников можно использовать алгоритмы, основанные на методе заметающей прямой. Эти алгоритмы отслеживают важные события в процессе перемещения вертикальной прямой слева направо по набору входных данных. В самом левом положении прямая не пересекает никаких объектов, но в процессе ее движения вправо происходит следующая последовательность событий:

- *вставка.* Найден левый конец отрезка, и этот отрезок может пересекать какой-либо другой отрезок прямой;
- *удаление.* Найден правый конец отрезка. Это означает, что заметающая прямая полностью прошла данный отрезок, и его можно исключить из дальнейшего рассмотрения;
- *пересечение.* Если для активных отрезков, пересекаемых заметающей прямой, поддерживается упорядочивание по расположению (сверху вниз), то следующее пересечение должно произойти между парой соседних отрезков. После такого пересечения относительный порядок этих двух отрезков меняется на обратный.

Для слежения за процессом требуются две структуры данных. Под будущие события отводится очередь событий или очередь с приоритетами, упорядоченная по x-координате всех возможных будущих событий: вставок, удалений и пересечений. Базовые реализации очередей с приоритетами рассматриваются в *разделе 12.2*. Настоящее представлено *горизонтом* — упорядоченным списком отрезков, пересекающих текущую позицию заметающей прямой. Для сопровождения горизонта можно использовать любую словарную структуру данных, например, сбалансированное дерево.

Чтобы адаптировать этот подход для поиска пересечения или объединения многоугольников, изменяется обработка трех основных типов событий. Этот алгоритм можно значительно упростить для выявления пересечения пар выпуклых многоугольников, т. к., во-первых, заметающая прямая пересекает, самое большее, четыре ребра многоугольников, что делает горизонт ненужным, и, во-вторых, не требуется сортировать очередь событий, т. к. мы можем начать обработку с самой левой вершины каждого многоугольника и двигаться направо в соответствии с естественным порядком вершин многоугольника.

Реализации. Как библиотека LEDA (см. *раздел 19.1.1*), так и библиотека CGAL (**www.cgal.org**) содержит реализацию на языке C++ самых разных алгоритмов для выявления пересечений отрезков и многоугольников. В частности, обе библиотеки предоставляют реализацию алгоритма Бентли-Оттманна (Bentley-Ottmann), основанного на методе заметающей прямой (см. [BO79]), для поиска всех k точек пересечения между n отрезками прямых на плоскости за время $O((n + k)\lg n)$.

Устойчивая программа на языке C для поиска пересечения двух выпуклых многоугольников представлена в книге [O'R01]. Подробности см. в *разделе 19.1.10*.

Группа GAMMA из университета Северной Каролины разработала несколько эффективных библиотек для обнаружения столкновений, из которых самой последней является библиотека SWIFT++ (см. [EL01]). Эта библиотека поддерживает выявление пересечений, вычисление приблизительных и точных расстояний между объектами, а также выявление контакта между парой объектов в сценах, состоящих из жестких многогранных моделей. Подробная информация обо всех этих библиотеках, включая условия загрузки и использования, доступна на веб-сайте **http://www.cs.unc.edu/~geom/collide/**.

Поиск взаимного пересечения набора полупространств является частным случаем поиска пересечения выпуклых оболочек. Лучшей программой для работы с выпуклыми оболочками в многомерных пространствах является Qhull (см. [BDH97]).

Эта программа широко используется в научных приложениях и доступна на веб-сайте **http://www.qhull.org/**.

Примечания

Отличный обзор алгоритмов поиска пересечений таких геометрических объектов, как отрезки, многоугольники и многогранники представлен в работе [Mou04]. Книги [dBvKOS00], [CLRS01] и [PS85] содержат главы, в которых обсуждается задача поиска пересечений геометрических объектов. Хорошее обсуждение частного случая поиска пересечений и объединений прямоугольников, ориентированных вдоль осей координат (задача, которая часто возникает при разработке СБИС), представлено в книге [PS85].

Алгоритм поиска пересечения отрезков с временем исполнения $O(n\lg n + k)$ приведен в работе [CE92]. Всесторонний обзор более простых, рандомизированных алгоритмов с такими же временными границами представлен в работе [Mul94].

Алгоритмы и программное обеспечение для обнаружения столкновений обсуждаются в работе [LM04].

Родственные задачи. Конфигурация прямых (см. *раздел 17.15*), планирование перемещений (см. *раздел 17.14*).

17.9. Разложение по контейнерам

Вход. Набор из n объектов размером d_1, ..., d_n. Набор из m контейнеров емкостью c_1, ..., c_n.

Задача. Уложить все объекты в контейнеры, используя как можно меньшее количество контейнеров (рис. 17.9).

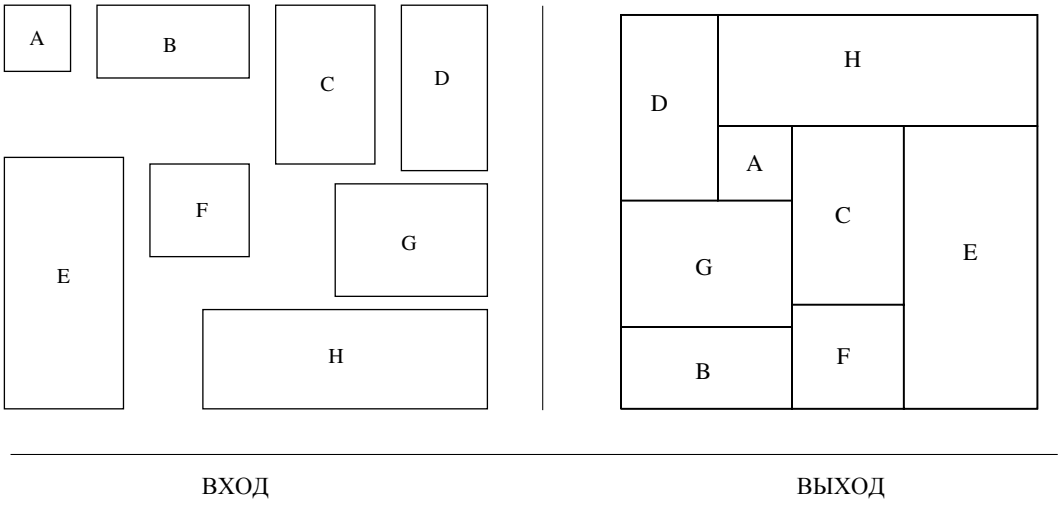


Рис. 17.9. Разложение по контейнерам

Обсуждение. Задача разложения по контейнерам возникает в разных приложениях упаковки и производства. Для примера рассмотрим раскройку листового железа под детали или раскройку ткани для шитья одежды. Для минимизации отходов мы хотим разместить детали или выкройки так, чтобы использовать как можно меньшее количество листов железа или рулонов материи. Задача выяснения, какую деталь разместить на каком листе и в каком месте, является вариантом задачи разложения по контейнерам, называющейся *задачей раскроя* (cutting stock problem). После изготовления деталей возникает другая задача упаковки — как погрузить ящики с деталями на грузовики, чтобы минимизировать необходимое количество грузовиков.

Даже самые простые на первый взгляд задачи разложения по контейнерам являются NP-полными (см. обсуждение задачи разбиения множества целых чисел в *разделе 13.10*). Поэтому нам приходится использовать эвристические подходы, а не алгоритмы, выдающие оптимальное решение в наихудшем случае. К счастью, для большинства задач разложения по контейнерам обычно хорошо подходят сравнительно простые эвристические методы. Более того, многие приложения имеют специфические ограничивающие условия, которые не позволяют использовать сложные алгоритмы решения задачи разложения по контейнерам. Выбор эвристического метода для решения конкретной задачи будет зависеть от следующих факторов:

- *формы и размера объектов.* Характер задачи разложения по контейнерам в большой степени зависит от формы пакуемых объектов. Собрать пазл (картинку-

головоломку) гораздо сложнее, чем уложить квадратики на прямоугольном поле. В задаче одномерной задаче разложения по контейнерам размер каждого объекта указывается в виде целого числа. Эта задача эквивалентна задаче упаковки ящиков одинаковой ширины в контейнер такой же ширины и является частным случаем задачи о рюкзаке (см. *раздел 13.10*).

Когда все объекты имеют одинаковый размер и форму, последовательное заполнение каждого ряда дает приемлемую, но не обязательно оптимальную упаковку. Попробуйте заполнить квадрат размером 3×3 прямоугольниками размером 2×1. Укладывая прямоугольники, ориентированные только в одном направлении, можно разместить только три из них, а если их ориентировать в двух направлениях, то поместится четыре прямоугольника;

- *ограничения на ориентацию и размещение объектов.* На практике у многих упаковочных коробок обозначен верх (что предписывает определенную ориентацию коробки) или присутствует надпись "не складывать в штабель" (означающая, что коробка может находиться только на самом верху штабеля). Такие условия ограничивают нашу свободу действий при упаковке и увеличивают количество грузовиков, необходимых для транспортировки товаров. Большинство транспортных компаний решают эту задачу, не обращая внимания на подобные маркировки. Несомненно, любая задача становится намного легче, если не беспокоиться о последствиях;
- *статичность или динамичность задачи.* Известен ли нам весь набор объектов для упаковки в начале работы (статическая задача) или же мы получаем их по одному и должны укладывать их по мере поступления (динамическая задача)? Упаковку можно выполнить гораздо лучше, если есть возможность планирования. Например, имеет смысл расположить объекты в таком порядке, который позволит осуществить более эффективную упаковку (т. е. отсортировать их по убыванию размера).

В стандартных эвристических методах статического разложения по контейнерам объекты упорядочиваются по размеру или форме, а потом укладываются в контейнеры. Типичные правила выбора контейнера:

- выбираем первый или самый левый контейнер, в который может поместиться объект;
- выбираем контейнер с наибольшим свободным объемом;
- выбираем контейнер с наименьшим свободным объемом, достаточным для размещения объекта;
- выбираем случайный контейнер.

Аналитические и экспериментальные результаты показывают, что самым лучшим эвристическим методом разложения по контейнерам является "первый подходящий контейнер в порядке убывания размера объектов". Сортируем объекты по убыванию размера. Вкладываем объекты по одному в контейнер, в котором имеется достаточно места для очередного объекта. Если в контейнере больше нет места, то переходим к следующему контейнеру. В случае одномерного разложения по контейнерам при таком подходе потраченное количество контейнеров никогда не превысит действительно необходимое больше чем на 22%, причем обычно этот показатель намного лучше. Такой метод интуитивно привлекателен, т. к. мы укладываем большие объекты в первую очередь и надеемся, что удастся уложить меньшие объекты в оставшееся место.

Этот алгоритм легко реализуется и имеет время исполнения $O(n\lg n + bn)$, где $b \le \min(n, m)$ — количество фактически использованных контейнеров. Просто для каждого объекта выполняем перебор контейнеров за линейное время. Возможен более быстрый алгоритм с временем исполнения $O(n\lg n)$, использующий двоичное дерево для отслеживания свободного места, оставшегося в каждом контейнере.

Чтобы удовлетворить специфическим ограничивающим условиям, можно разнообразить порядок размещения объектов. Например, ящики с маркировкой "не складывать в штабель" следует размещать в последнюю очередь (возможно, предварительно искусственно понизив высоту контейнеров, чтобы оставить достаточно свободного места), а ящики с маркировкой верха — в первую очередь (чтобы иметь больше свободы при укладывании ящиков сверху).

Укладывать ящики легче, чем объекты произвольной геометрической формы. Это настолько легче, что общее правило укладки произвольных объектов состоит в предварительной упаковке в индивидуальные ящики, после которой уже выполняется укладка ящиков в целевые контейнеры. Найти охватывающий прямоугольник для объекта-многоугольника не составляет труда: просто находим верхнюю, нижнюю, левую и правую касательные, идущие в заданном направлении. Выяснить ориентацию и минимизировать площадь (объем) такого прямоугольника (ящика) намного труднее, но все-таки возможно как на плоскости, так и в трехмерном пространстве (см. [O'R85]).

В случае невыпуклых объектов большой объем полезного пространства может оказаться неиспользованным из-за наличия пустот, получившихся после помещения детали в ящик. Одно из решений этой проблемы — найти *наибольший пустой прямоугольник* внутри каждого размещенного объекта и, если этот прямоугольник достаточно велик, поместить в него другие объекты.

Реализации. Коллекцию реализаций алгоритмов на языке FORTRAN для разных версий задачи о рюкзаке можно загрузить с веб-сайта **http://www.or.deis.unibo.it/kp.html**. Там же доступна электронная версия книги [MT90a].

Хорошо организованную коллекцию реализаций алгоритмов на языке C для решения разных видов задачи о рюкзаке и родственных ей задач, таких как разложение по контейнерам и загрузка контейнера, можно найти на веб-сайте **http://www.diku.dk/~pisinger/codes.html**.

Первым шагом упаковки объектов произвольной формы в контейнер является их размещение по индивидуальным прямоугольным контейнерам минимального объема. Этот алгоритм имеет почти линейное время исполнения (см. [BH01]).

Примечания

Обзоры литературы, посвященной задаче разложения по контейнерам и задаче раскроя, представлены в работах [CFC94], [CGJ96] и [LMM02]. Самым свежим справочником по различным вариантам задачи о рюкзаке является книга [KPP04]. Экспериментальные результаты эвристических алгоритмов решения задачи разложения по контейнерам изложены в работах [BJLM83] и]MT87].

Существуют эффективные алгоритмы поиска наибольшего пустого прямоугольника в многоугольнике (см. [DMR97]) и наборе точек (см. [CDL86]).

Упаковка шаров является важным и хорошо изученным частным случаем разложения по контейнерам. Общеизвестна "гипотеза Кеплера", касающаяся поиска наиболее плотной

упаковки единичных трехмерных шаров. Справедливость этой гипотезы была доказана в 1998 г. Хэйлзом (Hales) и Фергюсоном (Ferguson). Доказательство представлено в работе [Szp03]. Самым лучшим справочником по задаче упаковки шаров и родственным ей задачам является книга [CS93].

Миленкович (Milenkovic) много работал над двумерной задачей разложения по контейнерам для швейной промышленности, т. е. над минимизацией количества материала, необходимого для изготовления предметов одежды. Отчеты об этой работе содержатся в книгах [DM97] и [Mil97].

Родственные задачи. Задача о рюкзаке (см. *раздел 13.10*), задача упаковки множества (см. *раздел 18.2*).

17.10. Преобразование к срединной оси

Вход. Многоугольник или многогранник P.

Задача. Найти набор точек внутри объекта P, имеющих более чем одну ближайшую точку на границе P (рис. 17.10).

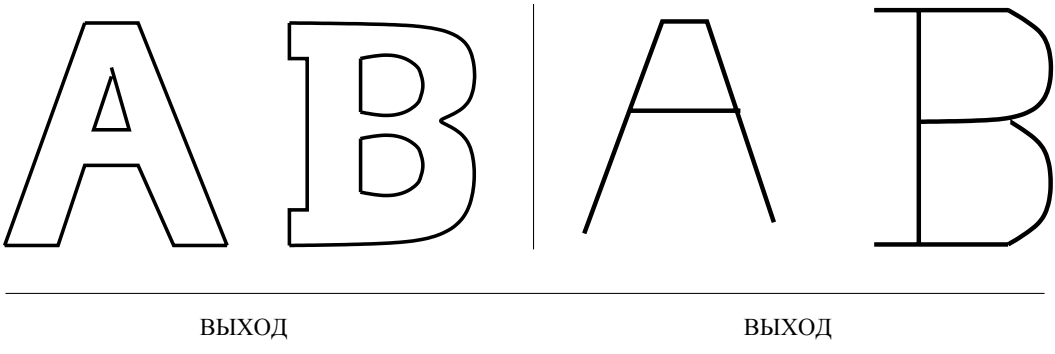


Рис. 17.10. Преобразование к срединной оси

Обсуждение. Преобразование к срединной оси применимо для "утончения" многоугольников или, другими словами, поиска *скелета*. Нашей целью является простое, устойчивое представление формы многоугольника. На рис. 17.10 "тонкие" версии букв A и B отражают их форму и будут устойчивы к изменению толщины линии или добавлению декоративных элементов, таких как засечки. Скелет также представляет собой центральное множество данной фигуры, и это свойство может быть использовано в других приложениях, таких как восстановление формы и планирование перемещений.

Результатом преобразования к срединной оси многоугольника является дерево, что позволяет использовать динамическое программирование для вычисления "расстояния редактирования" между скелетом известной модели и скелетом неизвестного объекта. Когда два скелета достаточно близки друг к другу, неизвестный объект классифицируется как экземпляр модели. Этот метод применяется в области компьютерного зрения и оптического распознавания текста. Скелетом многоугольника с отверстиями (как буква A или B) является не дерево, а вложенный планарный граф, но с ним тоже легко работать.

Существует два разных подхода к выполнению преобразований к срединной оси, в зависимости от того, имеем ли мы на входе произвольные геометрические точки или растровые изображения:

- *геометрические данные.* Вспомним, что диаграмма Вороного для множества точек S (см. *раздел 17.4*) разбивает плоскость на области вокруг каждой точки $s_i \in S$ таким образом, что все точки в области вокруг точки s_i находятся ближе к ней, чем к любой другой точке из множества S. Аналогичным образом, диаграмма Вороного для множества L отрезков разбивает плоскость на области вокруг каждого отрезка $l_i \in L$ таким образом, что все точки внутри области вокруг точки l_i находятся ближе к нему, чем к любому другому отрезку из множества L.

  Любой многоугольник определяется набором отрезков таких, что отрезок l_i имеет общую вершину с отрезком l_{i+1}. Преобразование к срединной оси многоугольника P сводится к получению той части диаграммы Вороного для отрезков, которая находится внутри этого многоугольника. Таким образом, для утончения многоугольника подойдет любая программа создания диаграмм Вороного для отрезков.

  Прямолинейным скелетом называется структура, родственная срединной оси многоугольника, за исключением того, что биссектрисы равноудалены не от его ребер, а от вспомогательных линий этих ребер. Для выпуклых многоугольников прямолинейный скелет, срединная ось и диаграмма Вороного идентичны, но в скелете общего вида биссектрисы могут не проходить по центру многоугольника. Прямолинейный скелет похож на результат преобразования к срединной оси, но его легче искать с помощью компьютера. В частности, все ребра прямолинейного скелета являются многоугольными;

- *изображения.* Оцифрованные изображения можно рассматривать как множества точек, расположенных на узлах целочисленной решетки. Таким образом, мы можем извлечь из изображения многоугольник, представляющий объект, и обработать его с помощью ранее описанных алгоритмов вычислительной геометрии. Но внутренние вершины скелета, скорее всего, не будут располагаться на узлах решетки. Применение алгоритмов вычислительной геометрии к задачам обработки изображений часто приводит к неудаче, потому что изображения состоят из отдельных пикселов и не являются непрерывными.

  В простейшем подходе к созданию скелета пиксельного объекта применяется метод "brush fire" (лесной пожар). Представьте себе огонь, охвативший все ребра многоугольника и продвигающийся с постоянной скоростью внутрь многоугольника. Скелет образуется точками, в которых сталкиваются две или более стены огня. Такой алгоритм обходит все граничные пикселы объекта, идентифицирует вершины, принадлежащие скелету, удаляет остальную часть границы и повторяет процесс. Алгоритм прекращает работу, когда все пикселы являются крайними, и возвращает объект толщиной всего в один или два пиксела. При правильной реализации время исполнения этого алгоритма будет линейным по отношению к количеству пикселов в изображении.

  Алгоритмы, которые манипулируют непосредственно пикселами, обычно легко поддаются реализации, по той причине, что в них не используются сложные структуры данных. Но при подходах, основанных на обработке пикселов, результаты по-

лучаются не вполне корректными. Например, скелет многоугольника не всегда будет деревом и не обязательно будет связным, а точки скелета могут оказаться "не совсем" равноудаленными от двух граничных ребер. Когда вы пытаетесь применять методы непрерывной геометрии в дискретном мире, у вас нет возможности решить задачу до конца, и с этим нужно смириться.

Реализации. Библиотека CGAL (**www.cgal.org**) содержит пакет процедур для вычисления прямолинейного скелета многоугольника P. Библиотека также содержит процедуры создания смещенных контуров, определяющих области внутри многоугольника P, точки которых находятся, по меньшей мере, на расстоянии d от границ многоугольника.

Программа VRONI (см. [Hel01]) является эффективным средством создания диаграмм Вороного на плоскости для отрезков, точек и дуг. Программа может с легкостью выполнять преобразования к срединной оси многоугольников, т. к. она поддерживает создание диаграмм Вороного для произвольных отрезков. Программа была протестирована на огромном количестве искусственно созданных и реальных наборов данных, причем некоторые из них содержали свыше миллиона вершин. Дополнительную информацию можно найти на веб-сайте разработчика **http://www.cosy.sbg.ac.at/~held/projects/vroni/vroni.html**. Другие программы создания диаграмм Вороного рассматриваются в *разделе 17.4*.

Программы для реконструкции или интерполирования облаков точек часто основаны на преобразованиях к срединной оси. Программное обеспечение Cocone выполняет приблизительные преобразования к срединной оси многогранной поверхности, интерполируя точки в E^3. Дополнительную информацию можно найти на веб-сайте разработчиков **http://www.cse.ohio-state.edu/~tamaldey/cocone.html**. Теоретические основы программы Cocone излагаются в работе [Dey06]. Программа Powercrust (см. [ACK01a] и [ACK01b]) выполняет дискретное приближенное преобразование к срединной оси, после чего реконструирует исходную поверхность на основе результатов этого преобразования. При достаточной плотности точек выборки алгоритм гарантированно выполняет геометрически и топологически правильную аппроксимацию поверхности.

Примечания

Всесторонние обзоры методов утончения в обработке изображений содержатся в работах [LLS92] и [Ogn93]. Преобразование к срединной оси было впервые использовано для изучения схожести фигур в биологии (см. [Blu67]). Применение преобразований к срединной оси в области распознавания образов рассматривается в работе [DHS00]. Преобразование к срединной оси является фундаментальной операцией для алгоритма Powercrust, который воссоздает поверхность по точкам выборки (см. [ACK01a] и [ACK01b]). Обсуждение преобразований к срединной оси можно найти в книгах [dBvKOS00], [O'R01 и [Pav82].

Срединную ось произвольного n-угольника можно вычислить за время $O(n\lg n)$ (см. [Lee82]), хотя для выпуклых многоугольников существуют алгоритмы с линейным временем исполнения (см. [AGSS89]). Алгоритм, выполняющий преобразование к срединной оси с временем исполнения $O(n\lg n)$, представлен в работе [Kir79].

Прямолинейные скелеты обсуждаются в работе [AAAG95], а алгоритм с временем исполнения меньше квадратичного — в работе [EE99]. Интересное применение прямолинейных скелетов для создания крыш виртуальных зданий описано в работе [LD03] .

Родственные задачи. Диаграммы Вороного (см. *раздел 17.4*), сумма Минковского (см. *раздел 17.16*).

17.11. Разбиение многоугольника на части

Вход. Многоугольник или многогранник P.

Задача. Разбить объект P на небольшое количество простых (обычно выпуклых) частей (рис. 17.11).

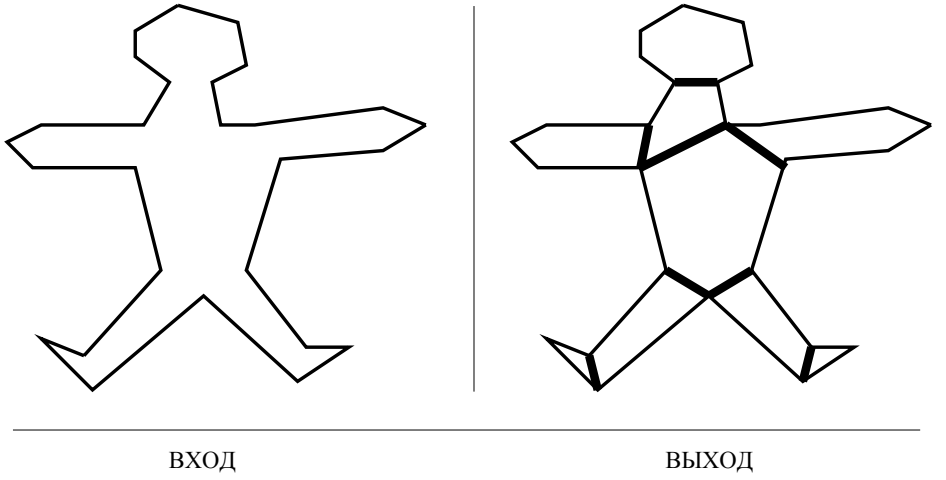


Рис. 17.11. Разбиение многоугольников на части

Обсуждение. Разбиение многоугольников на части является важным шагом предварительной обработки во многих алгоритмах вычислительной геометрии, т. к. на выпуклых объектах геометрические задачи решаются легче, чем на невыпуклых. Работать с небольшим количеством выпуклых фрагментов проще, чем с одним невыпуклым многоугольником.

В зависимости от конкретного приложения могут возникать разные варианты задачи разбиения многоугольника. Чтобы распознать их, ответьте на следующие вопросы.

- *Нужно ли, чтобы все фрагменты разбиения были треугольниками?* Триангуляция является самой главной из задач разбиения многоугольников, поскольку в ней областями разбиения являются треугольники. Треугольник имеет всего лишь три стороны и является выпуклым, что делает его простейшим возможным многоугольником.

  Любое триангуляционное разбиение n-вершинного многоугольника содержит ровно $n-2$ треугольника. Поэтому триангуляция не подходит для тех случаев, когда требуется разбиение на небольшое количество выпуклых частей. Критерием "хорошей" триангуляции является не количество треугольников, а их внешний вид. Триангуляция обсуждается в *разделе 17.3*.

- *Требуется покрытие или разбиение многоугольника?* Под *разбиением* многоугольника подразумевается разделение его внутренней области на части, не перекры-

вающие друг друга. А *покрытие* означает, что части многоугольника могут перекрываться. Оба способа могут оказаться полезными в разных ситуациях. Так, при декомпозиции сложного многоугольника во время подготовки к поиску в области (см. *раздел 17.6*), нам требуется разбиение, чтобы каждая искомая точка находилась ровно в одной области разбиения. А при декомпозиции многоугольника с целью его закрашивания будет достаточно покрытия, поскольку двойная заливка области цветом не влечет за собой никаких проблем. Мы будем рассматривать разбиение, т. к. его проще выполнить, и оно приемлемо для любого приложения, где требуется покрытие. Единственный недостаток этого подхода состоит в том, что разбиения могут потребовать больше памяти, чем покрытия.

- *Можно ли добавлять дополнительные вершины?* Можем ли мы добавлять в многоугольник вершины Штейнера (разделяя ребра или вставляя внутренние точки) или нам разрешено только соединять ребрами две существующие вершины? В первом случае мы получим разбиение с меньшим количеством областей, но будем вынуждены использовать более сложные алгоритмы обработки и, возможно, получим менее аккуратные результаты.

Существует простой и эффективный эвристический алгоритм Хертеля-Мельхорна для разбиения многоугольников на выпуклые области с помощью диагоналей. Алгоритм сначала выполняет произвольную триангуляцию многоугольника, а потом убирает все диагонали, после удаления которых остаются только выпуклые области. Удаление диагонали создает невыпуклую область только тогда, когда в результате получается внутренний угол, превышающий 180°. Выяснить, появится ли такой угол, можно за постоянное время, рассмотрев диагонали и стороны многоугольника, окружающие удаляемую диагональ. Полученное количество выпуклых областей никогда не превысит минимально возможное их количество более, чем в четыре раза.

Если минимизация количества областей разбиения не является вашей целью, то я рекомендую использовать этот эвристический подход. Экспериментируя с разными вариантами триангуляции и разным порядком удаления диагоналей, вы, возможно, сумеете получить более качественные разбиения.

С помощью динамического программирования можно выяснить, чему равно минимальное количество диагоналей, использованных в декомпозиции. Самая простая реализация, которая отслеживает количество областей для всех $O(n^2)$ частей многоугольника, разделенных ребром, имеет время исполнения $O(n^4)$. Более быстрые алгоритмы используют более сложные структуры данных и имеют время исполнения $O(n + r^2\min(r^2, n))$, где r — количество вершин с внутренним углом, превышающим 180°. Существует алгоритм с временем исполнения $O(n^3)$, который еще больше уменьшает количество областей разбиения, добавляя внутренние вершины. Но этот алгоритм сложен, и его трудно реализовать.

В другом варианте задачи многоугольник разбивается на *монотонные* области. Вершины y-монотонного многоугольника можно разделить на две цепочки таким образом, что любая горизонтальная линия будет пересекать любую из этих цепочек не более одного раза.

Реализации. Многие программы триангуляции начинают работу с трапецевидного или монотонного разбиения многоугольника. Кроме этого, триангуляция является про-

стейшим способом разбиения многоугольника на выпуклые области. В поисках отправной точки в выборе подходящего кода обратитесь к реализациям, упомянутым в *разделе 17.3*.

Библиотека CGAL (**www.cgal.org**) содержит набор процедур для разбиения многоугольников, включающий в себя реализации эвристического алгоритма Хертеля-Мельхорна для разбиения многоугольника на выпуклые области, алгоритма динамического программирования с временем исполнения $O(n^4)$ для поиска оптимального разбиения на выпуклые области и эвристического алгоритма на основе метода заметающей прямой с временем исполнения $O(n\log n)$ для разбиения на монотонные многоугольники.

Для решения задач триангуляции особенно хорошо подходит пакет GEOMPACK (**http://members.shaw.ca/bjoe/**), состоящий из набора процедур на языке FORTRAN 77. Пакет позволяет выполнять триангуляцию Делоне и разбиение многоугольников и многогранников на выпуклые области, а также триангуляцию Делоне в пространстве произвольной размерности.

Примечания

Сравнительно недавние обзоры содержатся в работах [Kei00] и [OS04]. В книге [KS85] дается отличный обзор существующего материала по разбиению и покрытию многоугольников. Описание эвристического алгоритма Хертеля-Мельхорна (см. [HM83]) можно найти в книге [O'R01]. Алгоритм динамического программирования для минимального разбиения на выпуклые области с временем исполнения $O(n + r^2\min(r^2, n))$ был предложен в работе [KS02]. Алгоритм с временем исполнения $O(r^3 + n)$ для минимизации количества выпуклых областей разбиения посредством вставки точек Штейнера представлен в книге [CD85]. В статье [LA06] содержится эффективный эвристический алгоритм с временем исполнения $O(nr)$ для разложения многоугольников с внутренними пустотами на "почти выпуклые" многоугольники, а впоследствии этот алгоритм был обобщен для работы с многогранниками.

С задачей покрытия многоугольника связана интересная задача о картинной галерее, в которой требуется разместить в данном многоугольнике минимальное количество охранников таким образом, чтобы каждая внутренняя точка этого многоугольника просматривалась, по крайней мере, одним охранником. Это соответствует покрытию данного многоугольника минимальным количеством звездообразных многоугольников. Прекрасной книгой, в которой представлена задача о картинной галерее и ее многие варианты, является [O'R87]. К сожалению, эта книга больше не издавалась.

Родственные задачи. Триангуляция (см. *раздел 17.3*), покрытие множества (см. *раздел 18.1*).

17.12. Упрощение многоугольников

Вход. Многоугольник или многогранник p, имеющий n вершин.

Задача. Найти многоугольник или многогранник p', имеющий только n' вершин, наиболее близкий по форме к исходному объекту p (рис. 17.12).

Обсуждение. Задача упрощения многоугольников имеет, в основном, два приложения. Первое касается удаления шума из фигуры, полученной, например, сканированием

изображения объекта. Упрощение ее позволит удалить шум и восстановить первоначальный объект. А второе относится к сжатию данных, когда требуется избавиться от незначительных деталей большого и сложного объекта, по возможности сохранив его первоначальный вид. Это может быть особенно полезным в области компьютерной графики, поскольку меньшая модель отображается на мониторе значительно быстрее.

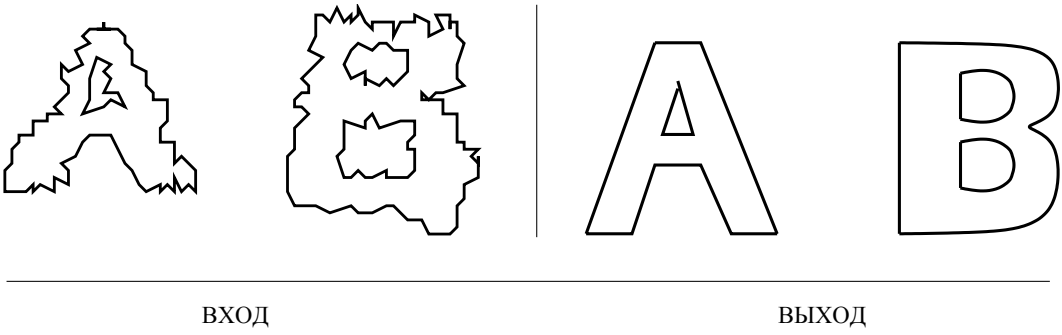


Рис. 17.12. Упрощение многоугольников

При решении задачи упрощения многоугольников возникает несколько вопросов.

- *Нужна ли выпуклая оболочка?* Самым простым решением будет выпуклая оболочка вершин объекта (см. *раздел 17.2*). Выпуклая оболочка многоугольника удаляет все его неровности и хорошо подходит для такой задачи, как упрощение перемещений робота. Однако использование выпуклой оболочки в системе оптического распознавания текста недопустимо, т. к. вогнутости символов являются их важнейшим свойством. Например, выпуклая оболочка буквы "X" идентична выпуклой оболочке буквы "H", т. к. обе оболочки — просто прямоугольники. Другая проблема заключается в том, что выпуклая оболочка выпуклого многоугольника никак его не упрощает.
- *Можно ли вставлять точки или разрешается только удалять их?* Типичной целью упрощения объекта является по возможности точное его представление с использованием заданного количества вершин. При самом простом подходе выполняются локальные модификации границы для уменьшения количества вершин. Например, если три последовательные вершины образуют небольшой треугольник, то центральную вершину можно удалить и заменить два ребра одним, не внося при этом значительных искажений в многоугольник.

  Однако подход, при котором вершины только удаляются, быстро изменит форму многоугольника до неузнаваемости. Более устойчивые эвристические методы перемещают вершины, чтобы закрыть промежутки, возникающие после удаления вершин. Такой подход типа "разделить и объединить" иногда позволяет добиться желаемого, хотя ничего не гарантирует. Получение хороших результатов гораздо вероятнее при использовании алгоритма Дугласа-Пекера, описанного далее.
- *Может ли получившийся многоугольник иметь самопересечения?* Серьезным недостатком инкрементальных процедур является то, что они не обеспечивают получение *простых* многоугольников, т. е. многоугольников, не содержащих самопересечений. Вследствие этого, "упрощенные" многоугольники могут иметь серьезные

дефекты, которые вызовут проблемы на последующих этапах обработки. Если важно получить простой многоугольник, то все его отрезки нужно проверить на попарное пересечение (см. *раздел 17.8*).

Подход к упрощению многоугольников, который гарантирует простую аппроксимацию, основан на поиске путей, состоящих из минимального количества отрезков. Реберной длиной пути между точками s и t называется количество отрезков в этом пути. Реберная длина прямого пути равна единице, а в общем случае она на единицу превышает количество поворотов в пути. В помещении с препятствиями реберная длина пути между точками s и t определяется минимальной реберной длиной по всем путям между этими точками.

Подход к упрощению многоугольника, основанный на вычислении реберной длины, заключается в "утолщении" границы многоугольника на некоторую приемлемую величину ε, в результате чего многоугольник оказывается в своеобразном канале. Замкнутый путь с минимальной реберной длиной в этом канале представляет простейший многоугольник, границы которого не отличаются от границ исходного больше, чем на ε. Легко вычисляемая аппроксимация реберной длины сводит задачу к поиску в ширину. В канале размещается дискретный набор возможных точек поворота, после чего пары точек, находящихся в прямой видимости, соединяются ребрами.

- *Требуется очистить изображение от шума (а не упростить многоугольник)?* При общепринятом подходе к очистке изображения от шума выполняется преобразование Фурье на этом изображении, из него отфильтровываются высокочастотные элементы, а потом выполняется обратное преобразование, восстанавливающее изображение. Подробную информацию о быстром преобразовании Фурье см. в *разделе 13.11*.

Вместо того, чтобы пытаться упростить сложный многоугольник, алгоритм Дугласа-Пекера для упрощения многоугольников находит примитивную аппроксимацию, а потом стремится улучшить ее. Для начала выбираем две вершины v_1 и v_2 многоугольника P, а вырожденный многоугольник v_1, v_2 и v_1 рассматриваем в качестве простой аппроксимации P'. Проходим через все вершины многоугольника P и выбираем самую дальнюю от соответствующего ребра многоугольника P'. Вставка этой вершины добавляет треугольник к многоугольнику P', минимизируя максимальное отклонение от многоугольника P. Таким образом точки можно вставлять до тех пор, пока не будет получен удовлетворительный результат. Процедура вставки k точек занимает время $O(kn)$, где $|P| = n$.

В трехмерном пространстве задача упрощения становится намного труднее. Более того, задача поиска минимальной поверхности, разделяющей два многогранника, является NP-полной. В качестве эвристического алгоритма упрощения многогранников можно использовать какой-либо многомерный аналог рассматриваемых здесь плоскостных алгоритмов. Дополнительную информацию см. в *подразделе "Примечания"*.

Реализации. Алгоритм Дугласа-Пекера достаточно прост. Реализация этого алгоритма на языке C, показывающая хорошую производительность в наихудшем случае, представлена в работе [HS94]. Программа доступна по адресу **http://www.cs.unc.edu/~snoeyink/papers/DPsimp.arch**.

Для автоматического генерирования иерархических структур, определяющих уровень детализации многоугольных моделей, применяется метод упрощающих оболочек. Пользователь указывает максимальное отклонение поверхности упрощенной модели от поверхности исходной модели, после чего генерируется новая, упрощенная модель. Реализацию такого подхода можно загрузить с веб-страницы **http://www.cs.unc.edu/~geom/envelope.html**. Данная процедура сохраняет отверстия и не допускает самопересечений.

Алгоритм QSlim на основе квадратичного упрощения может довольно быстро создавать высококачественные аппроксимации триангулированных поверхностей. Реализация алгоритма доступна по адресу **http://graphics.cs.uiuc.edu/~garland/software.html**.

Еще один подход к упрощению многоугольников основан на обработке результата преобразования к срединной оси многоугольника. Преобразование к срединной оси (см. *раздел 17.10*) создает скелет многоугольника, который можно упростить. После этого выполняется обратное преобразование, дающее в результате более простой многоугольник. Программное обеспечение Cocone может создавать аппроксимирующие преобразования к срединной оси многогранной поверхности, интерполируя точки в E^3. Дополнительную информацию можно найти на веб-сайте разработчиков **http://www.cse.ohio-state.edu/~tamaldey/cocone.html**. Теоретические основы программы Cocone излагаются в работе [Dey06]. Программа Powercrust (см. [ACK01a] и [ACK01b]) выполняет дискретное приближенное преобразование к срединной оси, после чего реконструирует исходную поверхность на основе результатов этого преобразования. При достаточной плотности точек выборки алгоритм гарантированно выполняет геометрически и топологически правильную аппроксимацию поверхности.

Библиотека CGAL (**www.cgal.org**) содержит процедуры для упрощения многоугольников и многогранников и поиска наименьшей охватывающей окружности или сферы.

Примечания

Алгоритм Дугласа-Пекера (см. [DP73]) составляет основу большинства схем упрощения очертаний. Быстрые реализации этого алгоритма представлены в работах [HS94] и [HS98]. Подход к упрощению многоугольников, основанный на вычислении реберной длины, представлен в работе [GHMS93]. Задача упрощения очертаний становится значительно сложнее в трех измерениях.

Даже задача построения выпуклого многоугольника с минимальным количеством вершин, находящегося между двумя вложенными выпуклыми многоугольниками, является NP-сложной (см. [DJ92]), хотя существуют аппроксимирующие алгоритмы для ее решения (см. [MS95b]).

Обзор алгоритмов для упрощения очертаний представлен в работе [HG97]. Использование преобразований к срединной оси (см. *раздел 17.10*) для упрощения очертаний рассматривается в работе [TH03].

Проверку многоугольника на простоту можно выполнить за линейное время, по крайней мере теоретически, благодаря существованию линейного алгоритма триангуляции Шазеля; см. [Cha91].

Родственные задачи. Преобразование Фурье (см. *раздел 13.11*), выпуклая оболочка (см. *раздел 17.2*).

17.13. Выявление сходства фигур

Вход. Две фигуры, P_1 и P_2.

Задача. Выявить степень сходства этих фигур (рис. 17.13).

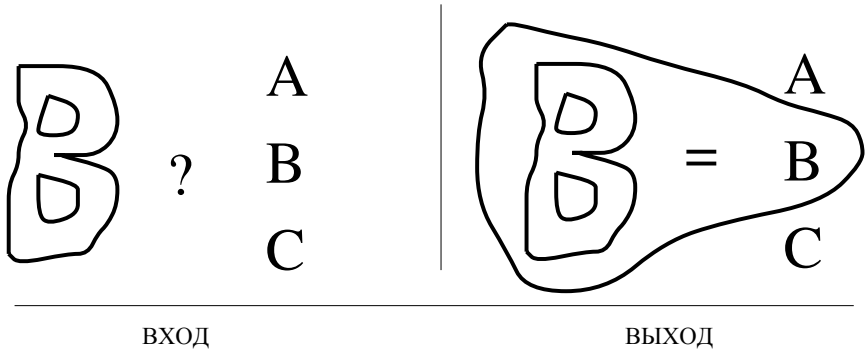


Рис. 17.13. Выявление сходства фигур

Обсуждение. Задача выявления сходства фигур относится к области распознавания образов. Рассмотрим, например, систему оптического распознавания текста. Дана библиотека моделей фигур, представляющих буквы, и неизвестные фигуры, полученные в результате сканирования текста. Нам нужно идентифицировать неизвестные фигуры, сопоставив их с наиболее похожими моделями.

Задача выявления сходства является плохо определенной по самой сути, т. к. смысл слова "сходство" зависит от конкретного приложения. Вследствие этого не существует единого алгоритмического подхода для решения всех задач выявления сходства фигур. Какой бы метод вы ни выбрали, вам придется потратить много времени на его настройку, чтобы добиться максимальной производительности.

Возможны следующие подходы к решению этой задачи:

- *использование расстояния Хемминга.* Допустим, что сравниваемые многоугольники наложены друг друга. Расстояние Хемминга определяется площадью симметрической разности между этими двумя многоугольниками, иными словами, площадью области, внутри одного из этих многоугольников, но не внутри обоих. Когда многоугольники идентичны и должным образом выровнены, расстояние Хемминга равно нулю. Если многоугольники отличаются только небольшим шумом вдоль границы, то расстояние Хемминга будет невелико.

  Вычисление площади области симметрической разности сводится к задачам поиска пересечения и объединения двух многоугольников (см. *раздел 17.8*), с последующим вычислением площадей (см. *раздел 17.1*). Трудным моментом при вычислении расстояния Хемминга является правильное выравнивание многоугольников. Задача выравнивания упрощается в таких приложениях, как оптическое распознавание текста, поскольку символы текста естественно выровнены в строках на странице. Существуют эффективные алгоритмы оптимизации наложения выпуклых многоугольников без использования вращения. Простые, но эффективные эвристические методы для решения этой задачи основаны на определении контрольных ориентиров для

каждого многоугольника (таких как центр тяжести, ограничивающий прямоугольник или экстремальные вершины) с последующим сопоставлением подмножеств этих ориентиров при выравнивании многоугольников.

Расстояние Хемминга легко вычисляется на растровых изображениях, поскольку после выравнивания изображений остается лишь сложить расстояния между соответствующими пикселами. Хотя расстояние Хемминга имеет концептуальный смысл и легко поддается реализации, оно отражает форму фигур весьма приблизительно и, скорее всего, окажется неэффективным в большинстве приложений;

- *использование хаусдорфова расстояния.* Альтернативной метрикой расстояний является хаусдорфово расстояние, равное расстоянию между точкой на многоугольнике P_1, максимально удаленной от многоугольника P_2, и многоугольником P_2. Эта метрика несимметрична. Например, длинный и тонкий выступ многоугольника P_1 может значительно увеличить хаусдорфово расстояние от P_1 до P_2, даже если каждая точка P_2 находится поблизости от какой-либо точки P_1. Небольшое утолщение всей границы одной из моделей (что может случиться при наличии шума на границе) может значительно увеличить расстояние Хемминга, но при этом мало повлияет на хаусдорфово расстояние.

  Какая же из этих двух метрик лучше? Все зависит от характера вашего приложения. Кроме прочего, правильное выравнивание многоугольников может быть сопряжено с трудностями и отнимает много времени;

- *сравнение скелетов.* При более эффективном подходе к выявлению сходства фигур применяется утончение (см. *раздел 17.10*), позволяющее получить древоподобный скелет каждого объекта, отражающий многие характеристики исходной фигуры. После этого задача сводится к сравнению двух скелетов с использованием таких их свойств, как топология дерева и длина и наклон ребер. Это сравнение можно смоделировать в виде выяснения изоморфизма подграфов (см. *раздел 16.9*), при котором ребра считаются совпадающими при достаточной схожести их длины и наклона;

- *метод опорных векторов.* Наконец, можно воспользоваться каким-либо обучающим методом, например, нейронной сетью или более мощным методом опорных векторов. Применение этих методов является разумным подходом к решению задач распознавания, когда имеется большой объем данных, но нет ясной идеи, что делать с этими данными. В первую очередь нам нужно определить набор таких характеристик фигуры, которые было бы легко найти с помощью компьютера. Например, это может быть площадь, количество сторон или количество отверстий. Исходя из этих характеристик программа-"черный ящик" (т. е. алгоритм обучения методом опорных векторов) обрабатывает обучающие данные и создает классифицирующую функцию. Эта функция принимает в качестве входа значения заданных характеристик и возвращает меру фигуры, т. е. степень ее близости к определенной фигуре.

  Каково качество получаемых классифицирующих функций? Все зависит от особенностей вашего приложения. Подобно любому специализированному методу, метод опорных векторов требует серьезной настройки, если вам требуется полностью использовать его потенциал.

  Кроме того, нужно иметь в виду следующее. Если вы не знаете, как классифицирующие функции "черного ящика" принимают решения, то вы не сумеете распо-

знать неверное решение, если они его выдадут. В связи с этим будет интересным пример системы, созданной для армии и предназначенной для различения танков и автомобилей. Система прекрасно работала на тестовых изображениях, но не выдержала полевых испытаний. Наконец, кто-то сообразил, что фотографии автомобилей снимались в солнечный день, а танков — в облачный, и система различала два типа объектов исключительно по присутствию облаков на заднем плане!

Реализации. Реализация на языке C алгоритма для сравнения изображений с использованием хаусдорфова расстояния доступна по адресу **http://www.cs.cornell.edu/vision/hausdorff/hausmatch.html**. Альтернативная метрика сходства многоугольников основана на угле проворота (см. [ACH+91]). Программа на языке C, использующая эту метрику, доступна по адресу **http://www.cs.sunysb.edu/~algorith**.

Существует также несколько отличных реализаций алгоритмов, основанных на методе опорных векторов, такие как библиотека Kernel-Machine (**http://www.terborg.net/research/kml/**), программа SVM^{light} (**http://svmlight.joachims.org/**) и широко используемая и хорошо поддерживаемая библиотека LIBSVM (**http://www.csie.ntu.edu.tw/~cjlin/libsvm/**).

Примечания

Среди книг общей тематики по алгоритмам классификации образов можно назвать [DHS00] и [JD88]. Было предложено большое количество разнообразных подходов к решению задачи выявления сходства фигур. См., например, [AMWW88], [ACH+91], [Ata84], [AE83], [BM89] и [OW85]. Хороший обзор содержится в работе [AG00].

Оптимальное выравнивание при сдвиге (но не при вращении) сравниваемых многоугольников из n и m вершин можно вычислить за время $O((n + m)\log(n + m))$ (см. [dBDK^{+}98]). Аппроксимация оптимального наложения при сдвиге и вращении была представлена в работе [ACP+07].

Алгоритм с линейным временем исполнения для вычисления хаусдорфова расстояния между двумя выпуклыми многоугольниками был представлен в работе [Ata83], а алгоритмы для общего случая — в работе [HK90].

Родственные задачи. Изоморфизм графов (см. *раздел 16.9*), преобразование к срединной оси (см. *раздел 17.10*).

17.14. Планирование перемещений

Вход. Робот многоугольной формы, начинающий движение в точке s в комнате, содержащей многоугольные препятствия, и конечная точка t.

Задача. Найти самый короткий маршрут от точки s до точки t, не пересекающий никаких препятствий (рис. 17.14).

Обсуждение. Сложность задачи планирования перемещений знакома каждому, кто пытался внести мебель в небольшую квартиру. Эта задача также возникает в области молекулярного докинга. Многие лекарства являются небольшими молекулами, действие которых основано на связывании с некоторой целевой молекулой. Поиск доступных участков связывания при разработке новых лекарств является экземпляром задачи планирования перемещений. Классическим приложением задачи планирования перемещений является разработка маршрутов роботов.

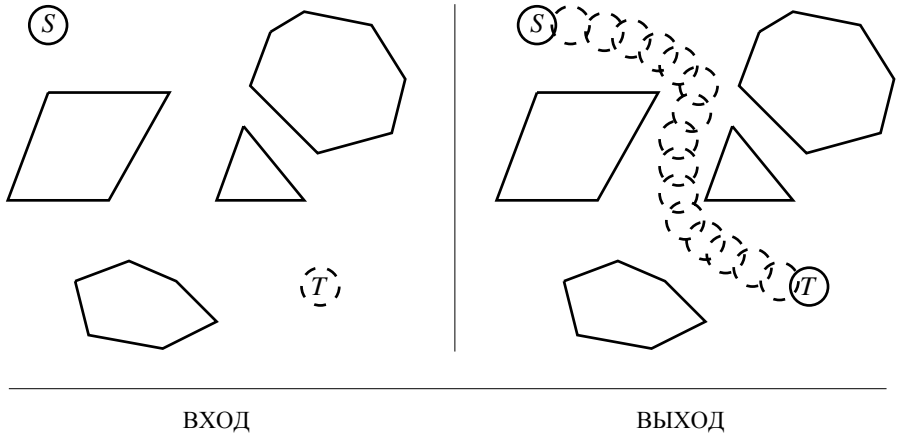


Рис. 17.14. Планирование перемещений

Наконец, планирование перемещений применяется в компьютерной анимации. Для данного набора объектов и их местоположения в сценах s_1 и s_2 алгоритм планирования перемещений может создать короткую последовательность промежуточных перемещений, чтобы преобразовать сцену s_1 в сцену s_2. Эти перемещения можно использовать для того, чтобы заполнить промежуточные сцены между сценами s_1 и s_2, что значительно облегчит работу аниматора.

Сложность задач планирования перемещений зависит от многих факторов.

- *Является ли робот точкой?* Задача планирования перемещения точечного робота сводится к поиску кратчайшего пути от точки s к точке t без столкновений с препятствиями. Проще всего реализуется подход, который заключается в создании *графа видимости* многоугольных препятствий и точек s и t. У этого графа видимости имеется вершина для каждой вершины препятствия, а две вершины препятствия соединяются ребром тогда и только тогда, когда между ними нет ребра препятствия, мешающего им "видеть" друг друга.

  Граф видимости можно создать, перебирая все кандидаты на ребро между $\binom{n}{2}$ пар вершин и проверяя, пересекаются ли они с каждым из n ребер препятствий. Впрочем, существуют и более быстрые алгоритмы. Каждому ребру графа видимости присваивается вес, равный его длине. Тогда кратчайший путь от точки s к точке t можно найти, используя алгоритм Дейкстры для поиска кратчайшего пути (см. *раздел 15.4*). Время исполнения этого алгоритма будет ограничено временем, требуемым для создания графа видимости.

- *Какие действия может выполнять робот?* Задача планирования перемещений становится значительно труднее, когда робот — не точка, а многоугольник. Тогда ширина всех используемых для его перемещения коридоров должна быть достаточной, чтоб робот мог пройти по ним.

  Алгоритмическая сложность зависит от числа *степеней подвижности* (degrees of freedom) робота. В частности, может ли робот, кроме перемещения, совершать повороты? Есть ли у робота конечности, способные сгибаться или вращаться независимо от него, как, например, рука? Каждая степень подвижности соответствует из-

мерению пространства поиска возможных конфигураций. Дополнительная степень подвижности означает большую вероятность существования короткого маршрута, но при этом задача поиска этого маршрута также становится труднее.

- *Можно ли упростить форму робота?* Алгоритмы планирования перемещений обычно сложны и трудоемки. Вам будет полезно все, что позволит упростить задачу. В частности, рассмотрите возможность помещения робота в охватывающую окружность. Если существует маршрут для этой окружности, то он будет также и маршрутом для находящегося внутри него робота. Кроме этого, любая ориентация окружности эквивалентна любой другой ее ориентации, поэтому повороты не будут играть никакой роли при поиске пути. Тогда все движения будут ограничены простым перемещением.
- *Ограничены ли движения одним лишь перемещением?* Когда повороты робота не разрешаются, можно воспользоваться методом *расширения препятствий*, чтобы свести задачу планирования перемещения робота-многоугольника к ранее решенной задаче планирования перемещения робота-точки. Для этого выбираем на роботе базисную точку и заменяем каждое препятствие его суммой Минковского с многоугольником робота. В результате получаем препятствие большего размера, включающее в себя след, оставляемый роботом, когда он движется вокруг препятствия, прикасаясь к нему. Поиск маршрута среди таких увеличенных препятствий от исходной базисной точки до цели определяет допустимый маршрут робота-многоугольника в исходном окружении.
- *Известны ли препятствия заранее?* До сих пор мы предполагали, что для планирования перемещений робота мы имеем карту с расположением всех препятствий. Но это невозможно в приложениях с движущимися препятствиями. Для решения задач планирования перемещений без карты существует два подхода. При первом подходе мы исследуем окружение, создаем карту и на ее основе разрабатываем маршрут от начальной точки к целевой. Другой, более простой подход напоминает перемещение в тумане с помощью компаса. Идем в направлении цели до тех пор, пока наш путь не будет перекрыт препятствием. Потом движемся вокруг препятствия, пока опять не появится возможность продолжить движение в прежнем направлении. К сожалению, этот подход неприменим в достаточно сложной обстановке.

Наиболее практичным подходом к решению общей задачи планирования перемещения будет произвольная выборка данных *конфигурационного пространства* робота. Конфигурационное пространство определяет набор допустимых положений робота с использованием одного измерения для каждой степени подвижности. Плоский робот, обладающий возможностью перемещения и поворота, имеет три степени подвижности, а именно x- и y-координату базисной точки на роботе и угол θ поворота относительно этой точки. Некоторые точки в этом пространстве соответствуют разрешенным позициям, а другие — препятствиям.

Набор разрешенных точек конфигурационного пространства создается случайной выборкой. Для каждой пары точек p_1 и p_2 выясняем, существует ли между ними прямой путь, не пересекающий препятствий. Таким образом строится граф, вершины которого представляют допустимые точки конфигурационного пространства, а ребра — свободные от препятствий пути между этими точками. Задача планирования перемещения

теперь сводится к поиску прямого пути между начальной/конечной точкой маршрута и какой-либо вершиной графа и последующему поиску кратчайшего пути между этими двумя вершинами.

Существует много способов улучшения этого базового метода, например, путем добавления дополнительных вершин в области, представляющих особый интерес. Создание такой дорожной карты позволяет точно решать задачи, которые в противном случае выглядят очень запутанными.

Реализации. Пакет Motion Planning Kit содержит библиотеку процедур на языке C++ и набор средств для разработки планировщиков перемещений для одного и нескольких роботов. В состав набора входит SBL — быстрый вероятностный планировщик маршрутов на дорожной карте. Набор можно найти по адресу **http://robotics.stanford.edu/~mitul/mpk/**.

Группа GAMMA из университета Северной Каролины разработала несколько эффективных библиотек для обнаружения столкновений (строго говоря, это не то же самое, что планирование перемещений), из которых самой последней является библиотека SWIFT++ (см. [EL01]). Эта библиотека поддерживает выявление пересечений, вычисление приблизительных и точных расстояний между объектами, а также обнаружение контакта между парой объектов в сценах, составленных из негибких многогранных моделей. Подробную информацию об этих библиотеках, включая условия загрузки и использования, можно найти на веб-сайте **http://www.cs.unc.edu/~geom/collide/**.

Библиотека вычислительной геометрии CGAL (**www.cgal.org**) содержит большое количество реализаций алгоритмов, касающихся задачи планирования перемещений, включая процедуры создания графов видимости и вычисления сумм Минковского. В книге [O'R01] предоставлена реализация алгоритма для планирования перемещений на плоскости двухшарнирного робота-манипулятора. Подробности см. в *разделе 19.1.10*.

Примечания

В книге [Lat91] обсуждаются практические подходы к планированию перемещений, включая описанный выше метод случайной выборки. Две другие заслуживающие внимания книги по предмету планирования перемещений можно загрузить бесплатно, [LaV06] — по адресу **http://planning.cs.uiuc.edu**, а [Lau98] — по адресу **http://homepages.laas.fr/jpl/book.html**.

Задача планирования перемещений изначально изучалась Шварцем (Schwartz) и Шариром (Sharir). Разработанное ими решение создает полное пространство позиций робота, не пересекающихся с препятствиями, после чего находит кратчайший путь в соответствующей компоненте связности. Эти описания свободного от препятствий пространства очень сложны. Доклады, посвященные задаче планирования перемещений, приводятся в книге [HSS87], а в книге [Sha04] дается обзор последних результатов.

Самый лучший результат для подхода, основанного на использовании свободного от препятствий пространства, был приведен в [Can87], где показано, что любую задачу с d степенями подвижности можно решить за время $O(n^d \lg n)$, хотя для частных случаев общей задачи планирования перемещений существуют более быстрые алгоритмы. Подход к задаче планирования перемещений, основанный на методе расширения препятствий, был представлен в работе [LPW79]. Обсуждение эвристического метода "с компасом в тумане" содержится в работе [LS87].

Временная сложность алгоритмов, основанных на методе свободного от препятствий пространства, зависит от комбинаторной сложности расположения поверхностей, определяющих свободное пространство. Алгоритмы для поддержания таких компоновок представлены в *разделе 17.15*. В анализе таких компоновок часто возникают последовательности Дэвенпорта-Шинцеля. Всестороннее рассмотрение последовательностей Дэвенпорта-Шинцеля и их значимость для задачи планирования перемещений приводится в книге [SA95].

Граф видимости n отрезков с E парами видимых вершин можно создать за время $O(n\lg n + E)$ (см. [GM91] и [PV96]), что является оптимальным результатом. Алгоритм с временем исполнения $O(n\lg n)$ для поиска кратчайшего пути для робота-точки среди препятствий-многоугольников приводится в работе [HS99]. А в работе [Che85] приводится алгоритм с временем исполнения $O(n^2\lg n)$ для поиска кратчайшего пути среди препятствий-многоугольников для робота-окружности.

Родственные задачи. Задача поиска кратчайшего пути (см. *раздел 15.4*), сумма Минковского (см. *раздел 17.16*).

17.15. Конфигурации прямых

Вход. Набор прямых l_1, ..., l_n.

Задача. Найти разбиение плоскости, определяемое набором прямых l_1, ..., l_n (рис. 17.15).

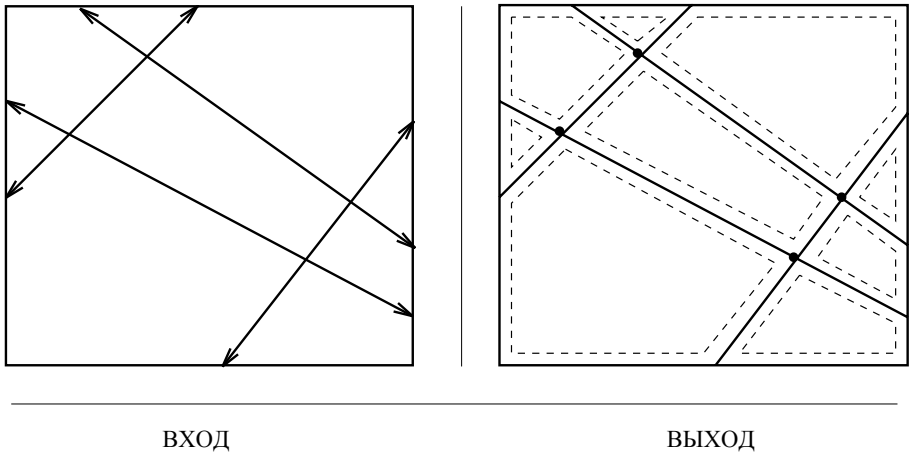


Рис. 17.15. Разбиение плоскости

Обсуждение. Явное создание областей, формируемых пересечениями набора n прямых, является одной из фундаментальных задач вычислительной геометрии. Многие задачи сводятся к созданию и анализу конфигурации некоторого набора прямых. Можно привести два примера подобных задач:

- *проверка на вырожденность.* Для данного набора из n прямых на плоскости выяснить, проходят ли какие-либо три из них через одну и ту же точку. Проверка всех таких триад методом исчерпывающего перебора займет время $O(n^3)$. В качестве альтернативы мы можем создать конфигурацию прямых, а потом рассмотреть каж-

дую вершину и явно подсчитать ее степень, причем все это делается за квадратичное время;

- *удовлетворение максимальному количеству линейных ограничений.* Допустим, что нам дан набор линейных ограничений, каждое в виде $y \le a_i x + b_i$. Нам нужно найти точку на плоскости, которая удовлетворяет самому большому количеству таких ограничений. Для решения этой задачи сначала создаем конфигурацию прямых. Все точки в любой области (или ячейке), образуемой пересекающимися линиями, удовлетворяют одному и тому же набору ограничений, поэтому, чтобы найти глобальный максимум, нам нужно проверить только одну точку в каждой ячейке.

При разработке алгоритмов бывает полезно формулировать геометрические задачи в терминах свойств конфигурации. К сожалению, следует признать, что на практике конфигурации не пользуются такой популярностью, как можно было бы предполагать. Основной причиной является то обстоятельство, что для правильного применения конфигураций требуется определенный уровень знаний. Библиотека вычислительной геометрии CGAL предоставляет общую и достаточно устойчивую реализацию, оправдывающую усилия по использованию конфигураций. Столкнувшись с задачей конфигурации прямых, постарайтесь ответить на следующие вопросы.

- *Какой метод лучше всего подходит для создания конфигураций прямых?* Для этой цели используются инкрементальные алгоритмы. Начинаем с конфигурации из одной или двух линий, в которую добавляем по одной новые прямые, получая конфигурации все большего размера. Чтобы добавить в конфигурацию новую прямую, начинаем с самой левой ячейки, содержащей эту прямую, и идем по конфигурации вправо, перемещаясь от текущей ячейки к смежной ячейке и разбивая на две части те ячейки, которые содержат новую прямую.
- *Каким будет размер создаваемой конфигурации?* Согласно теореме о зоне, k-я добавленная прямая пересекает k ячеек конфигурации, причем $O(k)$ ребер образуют границы этих ячеек. Это означает, что мы можем перебрать все ребра каждой ячейки, обнаруживаемой в процессе добавления прямых, и быть уверенным, что общий объем работы, выполненный при добавлении прямой в конфигурацию, будет линейным. Поэтому общее время добавления всех n прямых для создания полной конфигурации будет $O(n^2)$.
- *Какова цель создания конфигурации?* Часто требуется найти ячейку данной конфигурации, содержащую заданную точку. Это задача выяснения местоположения точки (см. *раздел 17.7*). А для данной конфигурации прямых или отрезков часто требуется вычислить все точки пересечения этих прямых. Задача выявления пересечений обсуждается в *разделе 17.8*.
- *Входные данные — это набор точек, а не прямых?* Хотя прямые и точки являются разными геометрическими объектами, они могут заменить друг друга. Используя *преобразования двойственности*, можно преобразовать прямую L в точку p и наоборот:

$$L\colon y = 2ax - b \leftrightarrow p\colon (a, b)$$

Важность двойственности состоит в том, что мы теперь можем применять конфигурации прямых в задачах на точках, нередко получая неожиданные результаты.

Рассмотрим пример. Имеется множество n точек и требуется узнать, не располагаются ли какие-либо три из этих точек на одной и той же прямой. Эта задача похожа на задачу проверки на вырожденность, рассматриваемую ранее. В действительности, это *та же самая задача*, но точки и прямые в ней поменялись ролями. Мы можем выполнить преобразование двойственности точек в прямые, как описано выше, а потом выполнить поиск вершины, через которую проходят три прямых. Преобразование двойственности этой вершины определяет прямую, на которой располагаются три исходные вершины.

Часто требуется перебрать все многоугольники существующей конфигурации ровно один раз. Такие алгоритмы называются *алгоритмами заметающей прямой* и рассматриваются в *разделе 17.8*. Базовая процедура таких алгоритмов состоит в упорядочивании точек пересечения по x-координате и выполнении обхода слева направо с сохранением информации, обнаруженной в процессе этого обхода.

Реализации. Библиотека CGAL (**www.cgal.org**) содержит пакет общих процедур для работы с конфигурациями кривых (не только прямых) на плоскости. Эта библиотека должна быть отправной точкой для любого проекта, в котором используются конфигурации.

Устойчивый код на языке C++ для создания и топологического заметания конфигураций можно найти на веб-сайте **http://www.cs.tufts.edu/research/geometry/other/sweep**. В библиотеке CGAL предоставлено расширение топологического заметания для работы с комплексом видимости набора попарно непересекающихся выпуклых плоских множеств.

Пакет Arrange на языке C предназначен для размещения многоугольников на плоскости или на сфере. Многоугольники могут быть вырожденными, и в таком случае работа сводится к размещению отрезков. В пакете используется рандомизированный инкрементальный алгоритм и поддерживается эффективное выяснение местоположения точки в разбиении. Пакет Arrange был разработан Майклом Голдвассером (Michael Goldwasser) и доступен на веб-сайте **http://euler.slu.edu/~goldwasser/publications**.

Примечания

Подробное изложение комбинаторной теории конфигураций и соответствующие алгоритмы представлены в книге [Ede87]. Эта книга — основной справочник для каждого, кто серьезно интересуется данной темой. Свежие обзоры комбинаторных и алгоритмических результатов приводятся в работах [AS00] и [Hal04]. Обсуждение принципов создания конфигураций можно найти в книгах [dBvKOS00] и [O'R01]. Вопросы реализации процедур библиотеки CGAL рассматриваются в работах [HH00] и [FWH04].

Конфигурации естественным образом обобщаются для случая многомерных пространств. В трех измерениях разбиение пространства определяется плоскостями, а в многомерных пространствах — гиперплоскостями. Теорема о зоне утверждает, что сложность любой конфигурации из n d-мерных гиперплоскостей равна $O(n^d)$, а любая одиночная гиперплоскость пересекает ячейки со сложностью $O(n^{d-1})$. Это дает основание для алгоритма инкрементального создания конфигураций. Обход вдоль границы ячейки с целью поиска следующей ячейки, пересекаемой гиперплоскостью, занимает время, пропорциональное количеству ячеек, создаваемых добавлением гиперплоскости.

Теорема о зоне имеет несколько запутанную историю. Первоначальные доказательства оказались ошибочными для случая многомерных пространств. Обсуждение теоремы и верное доказательство приведены в работе [ESS93]. Теория последовательностей Дэвенпорта-Шинцеля тесно связана с изучением конфигураций (см. [SA95]).

Простой алгоритм заметания конфигурации прямых сортирует n^2 точек пересечения по x-координате и поэтому занимает время $O(n^2 \lg n)$. Использование топологического заметания (см. [EG89] и [EG91]) избавляет от необходимости в сортировке, вследствие чего обход конфигурации выполняется за линейное время. Этот алгоритм легко поддается реализации и его можно использовать, чтобы ускорить работу многих алгоритмов, в основе которых лежит метод заметающей прямой. Устойчивая реализация и экспериментальные результаты представлены в работе [RSS02].

Родственные задачи. Выявление пересечений (см. *раздел 17.8*), выяснение местоположения точки (см. *раздел 17.7*).

17.16. Сумма Минковского

Вход. Наборы точек или многоугольники A и B, содержащие n и m вершин соответственно.

Задачи. Найти сумму Минковского: $A + B = \{x + y | x \in A, y \in B\}$ (рис. 17.16).

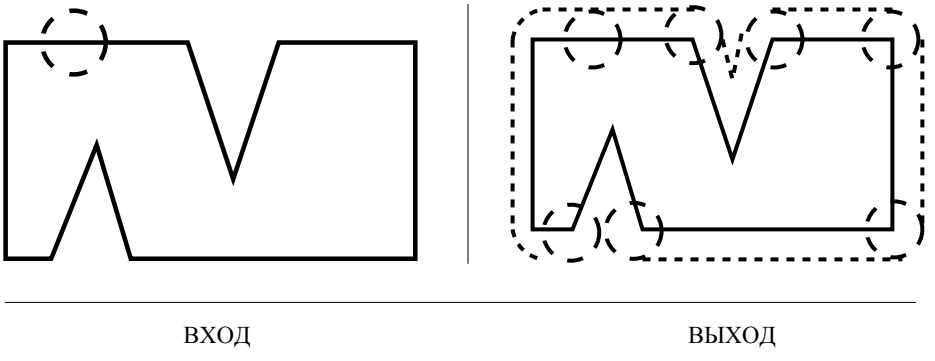


Рис. 17.16. Сумма Минковского

Обсуждение. Вычисление суммы Минковского — полезная геометрическая операция, с помощью которой можно увеличивать объекты, чтобы они удовлетворяли определенным требованиям. Например, в известном подходе к планированию перемещений роботов многоугольной формы в комнате с препятствиями многоугольной формы (см. *раздел 17.14*) размер каждого из препятствий увеличивается на величину суммы Минковского препятствия и робота. Таким образом, задача сводится к более простому случаю планирования перемещения робота-точки. Другим применением суммы Минковского является задача упрощения формы многоугольников (см. *раздел 17.12*). Здесь мы делаем границу объекта толще, чтобы создать вокруг него канал, а потом выясняем упрощенную форму объекта, для чего находим путь внутри этого канала, состоящий из минимального количества отрезков. Наконец, сумма Минковского для объекта, имеющего очень неровную границу, и небольшого круга поможет сгладить его границу, устранив небольшие впадины и выступы.

Далее приводится определение суммы Минковского (при этом предполагается, что многоугольники A и B находятся в некоторой системе координат):

$$A + B = \{x + y | x \in A, y \in B\},$$

где $x + y$ — сумма соответствующих векторов. В терминах операции переноса сумма Минковского — это объединение всех переносов многоугольника A на расстояние, определяемое точкой внутри многоугольника B. При вычислении суммы Минковского возникают следующие вопросы.

- *Входные объекты представляют собой растровые изображения или многоугольники?* Если A и B являются растровыми изображениями, то определение суммы Минковского подсказывает простой алгоритм для ее вычисления. Инициализируем достаточно большую матрицу пикселов, вычислив сумму Минковского ограничивающих прямоугольников для многоугольников A и B. Для каждой пары точек из A и B складываем их координаты и уменьшаем яркость соответствующего пиксела. Эти алгоритмы становятся более сложными, если требуется явное представление суммы Минковского в виде многоугольника.
- *Требуется ли увеличить размер объекта на заданное значение?* При типичной операции увеличения объекта размер модели M увеличивается на определенное значение допуска t, называемое *смещением* (offsetting). Как показано на рис. 17.16, результат достигается путем вычисления суммы Минковского для модели M и круга радиусом t. Базовые алгоритмы продолжают работать, хотя полученный объект не является многоугольником. Теперь его граница состоит из дуг и отрезков.
- *Являются ли входные объекты выпуклыми?* Сложность вычисления суммы Минковского во многом зависит от формы многоугольников. Если оба многоугольника выпуклые, то сумму Минковского можно вычислить за время $O(n + m)$, проведя один многоугольник по контурам другого. Если один из многоугольников не является выпуклым, то размер результирующего объекта может достичь значения $\Theta(nm)$. Если же невыпуклыми являются оба многоугольника, то размер результата может достичь значения $\Theta(n^2m^2)$. Суммы Минковского для невыпуклых многоугольников часто имеют эстетически непривлекательный вид, поскольку отверстия в них возникают или исчезают самым неожиданным образом.

Простейший подход к вычислению сумм Минковского основан на триангуляции и объединении. Сначала выполняем триангуляцию каждого многоугольника, а затем вычисляем сумму Минковского для каждого треугольника из A с каждым треугольником из B. Сумма двух треугольников легко вычисляется и является частным случаем суммы выпуклых многоугольников, рассматриваемым далее. Объединением этих $O(nm)$ выпуклых многоугольников будет сумма $A + B$. Алгоритмы поиска объединения многоугольников основаны на методе заметающей прямой (см. *раздел 17.8*).

Вычислить сумму Минковского для двух выпуклых многоугольников легче, чем для общего случая, т. к. эта сумма всегда будет выпуклой. Когда многоугольники выпуклые, проще перемещать многоугольник A вдоль границы многоугольника B и вычислять сумму для каждого ребра. Подход, в котором каждый многоугольник разбивается на небольшое количество выпуклых фрагментов (см. *раздел 17.11*), а потом вычисляется сумма Минковского для каждой пары фрагментов, обычно намного эффективнее, чем обработка двух многоугольников, полностью подвергнутых триангуляции.

Реализации. Пакет библиотеки CGAL предоставляет эффективные процедуры вычисления суммы Минковского для двух произвольных многоугольников, а также вычисления точных и приблизительных смещений.

Реализация алгоритма вычисления суммы Минковского для двух выпуклых многогранников в трех измерениях описана в работе [FH06] и доступна по адресу **http://www.cs.tau.ac.il/~efif/CD/**.

Примечания

Обсуждение алгоритмов вычисления суммы Минковского можно найти в книгах [dBvKOS00] и [O'R01]. Самые быстрые алгоритмы вычисления суммы Минковского представлены в работах [KOS91] и [Sha87].

Практическая эффективность вычисления суммы Минковского в общем случае зависит от того, каким образом многоугольники разбиты на выпуклые фрагменты. Разбиение многоугольников на минимальное количество выпуклых фрагментов не обязательно будет оптимальным решением. Всестороннее обсуждение методов разбиения многоугольников для вычисления суммы Минковского представлено в работе [AFH02].

Комбинаторная сложность суммы Минковского для двух выпуклых многогранников в трех измерениях полностью определена в работе [FHW07]. Реализация алгоритма вычисления суммы Минковского для таких многогранников описана в работе [FH06].

В работе [KS90] представлен эффективный алгоритм, основанный на вычислении сумм Минковского для планирования перемещений роботов многоугольной формы.

Родственные задачи. Преобразования к срединной оси (см. *раздел 17.10*), планирование перемещений (см. *раздел 17.14*), упрощение многоугольников (см. *раздел 17.12*).

ГЛАВА 18

Множества и строки

Как множества, так и строки являются коллекциями объектов, и разница между ними состоит в том, имеет ли значение порядок элементов коллекции. Множества — это коллекции символов, порядок которых не имеет значения, в то время как строки определяются как последовательность символов.

Тот факт, что элементы в строках упорядочены, позволяет решать задачи со строками намного эффективнее, чем задачи с множествами, благодаря возможности использовать такие методы, как динамическое программирование, и такие развитые структуры данных, как суффиксные деревья. Причиной повышения интереса, проявляемого к алгоритмам обработки строк, и важности этих алгоритмов являются такие приложения, как биоинформатика, поиск в Интернете и другие приложения обработки текста. Среди последних книг по алгоритмам для обработки строк можно назвать следующие:

- [Gus97]) — пожалуй, самое лучшее введение в обработку строк. Эта книга содержит подробное обсуждение суффиксных деревьев, а также современные формулировки классических алгоритмом для точного сравнения строк;
- [CHL07] — подробное описание алгоритмов обработки строк, автор которого является признанным лидером в данной области. Перевод с французского на английский;
- [NR07] — краткое, но имеющее практическую ценность обсуждение алгоритмов поиска по образцу, ориентированное на конкретные реализации. Особое внимание уделяется подходам, в которых применяется параллелизм на уровне битов;
- [CR03] — обзор некоторых специальных тем, имеющих отношение к алгоритмам обработки строк, с акцентом на теорию.

Ежегодная конференция CPM (Combinatorial Pattern Matching, комбинаторное сравнение строк) является основным форумом, посвященным практическим и теоретическим аспектам алгоритмов обработки строк.

18.1. Поиск покрытия множества

Вход. Коллекция подмножеств $S = \{S_1, ..., S_m\}$ универсального множества $U = \{1, ..., n\}$.

Задача. Найти наименьшую коллекцию T подмножеств множества S, объединение которых равно универсальному множеству, т. е. $\cup_{i=1}^{|T|} T_i = U$ (рис. 18.1).

Обсуждение. Задача о покрытии множества возникает, когда мы стремимся экономно приобрести товары, разложенные по наборам. Мы хотим получить, как минимум, по одному товару каждого вида, покупая при этом как можно меньшее количество набо-

ров. Задача поиска *хоть какого-нибудь* покрытия множества не представляет сложности, т. к. можно купить все предлагаемые наборы товаров. Но поиск наименьшего покрытия множества позволяет нам минимизировать наши расходы. Задача о покрытии множества позволила упростить формулировку задачи оптимизации выбора лотерейных билетов, обсуждавшейся в *разделе 1.6*. В той задаче нам требовалось купить наименьшее количество лотерейных билетов, покрывающее все комбинации данного набора.

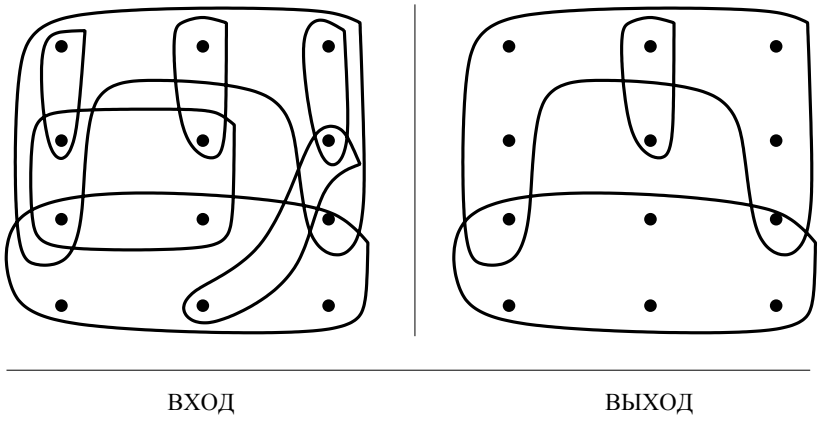


Рис. 18.1. Покрытие множества

Другим интересным применением задачи о покрытии множества является задача оптимизации булевой логики. Рассмотрим, например, булеву функцию с k переменными, возвращающую 0 или 1 для каждого из возможных 2^k входных векторов. Нам нужно найти простейшую логическую схему, которая реализует эту функцию. Один из подходов — определить на этих переменных и их дополнениях булеву формулу в дизъюнктивной нормальной форме (ДНФ), например: $x_1\overline{x}_2 + \overline{x}_1\overline{x}_2$. Для каждого входного вектора мы могли бы построить один член И, а потом выполнить операцию ИЛИ над всеми этими членами И, однако мы существенно сэкономим, если вынесем за скобки общие подмножества переменных. Имея набор выполнимых членов И, каждый из которых покрывает какое-либо подмножество нужных нам векторов, мы хотим объединить операцией ИЛИ наименьшее количество членов, которые реализует данную функцию. Это как раз и есть задача о покрытии множества. Существует несколько разновидностей задачи о покрытии множества, и для их распознавания нужно ответить на следующие вопросы.

- *Разрешено ли повторное включение элементов в покрытие?* Если любой элемент может входить только в одно подмножество, то задача о покрытии множества превращается в задачу *укладки множества*, которая рассматривается в *разделе 18.2*. Если есть возможность включать один элемент в несколько подмножеств, то следует воспользоваться ею, т. к. в результате обычно удается получить меньшее покрытие.
- *В задаче рассматривается множество ребер или множество вершин графа?* Задача о покрытии множества имеет общий характер и включает в себя несколько полезных задач на графах в качестве частных случаев. Допустим, что нужно найти

наименьшее множество ребер графа, которое затронет каждую вершину, по меньшей мере, один раз. Решением данной задачи будет максимальное *паросочетание* в графе (см. *раздел 15.6*), к которому добавлены любые ребра, позволяющие охватить все вершины, не вошедшие в паросочетание. А теперь допустим, что нужно найти наименьшее множество вершин графа, которое задействует каждое ребро, по крайней мере, один раз. Эта задача является задачей о вершинном покрытии, эвристические методы для решения которой рассматриваются в *разделе 16.3*.

Здесь полезно продемонстрировать, как смоделировать вершинное покрытие в виде экземпляра покрытия множества. Пусть универсальное множество U соответствует множеству ребер $\{e_1, ..., e_m\}$. Создаем n подмножеств, где подмножество S_i состоит из ребер, инцидентных вершине v_i. Хотя задача о вершинном покрытии является лишь замаскированным экземпляром задачи о покрытии множества, имеет смысл воспользоваться более эффективными эвристическими методами решения частной задачи о вершинном покрытии.

- *Каждое подмножество содержит только два элемента?* Если в любом подмножестве содержится, самое большее, два элемента, считайте, что вам повезло. Для этого частного случая можно получить оптимальное решение, т. к. он сводится к поиску максимального паросочетания графа. К сожалению, как только количество элементов во всех подмножествах возрастает до трех, задача становится NP-полной.
- *Нужно найти множества, содержащие элементы, или элементы, содержащиеся во множествах?* В *задаче о минимальном множестве представителей* (hitting set) нужно найти множество элементов, которые совместно представляют каждое подмножество из данной коллекции. Пример минимального множества представителей показан на рис. 18.2. Оптимальное решение экземпляра задачи о минимальном множестве представителей получается при выборе элементов 1 и 3 или элементов 2 и 3 (*а*). Эту задачу можно преобразовать в экземпляр двойственной задачи о покрытии множества, оптимальным решением которой будет выбор подмножеств 1 и 3 или подмножеств 2 и 4 (*б*).

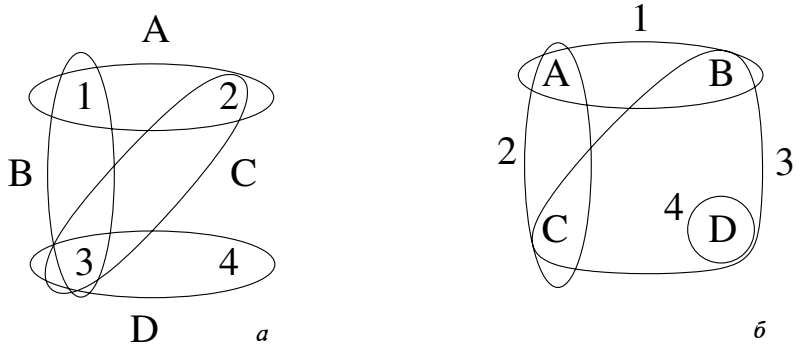


Рис. 18.2. Пример минимального множества представителей

Входной экземпляр задачи о минимальном множестве представителей идентичен входному экземпляру задачи о покрытии множества, но здесь требуется найти такое минимальное подмножество элементов $T \subset U$, чтобы каждое подмножество S_i содержало, по крайней мере, один элемент подмножества T. Таким образом, $S_i \cap T \neq 0$

для всех $1 \le i \le m$. Допустим, что мы хотим создать небольшой парламент, в который входил бы, по меньшей мере, один представитель от каждой этнической группы. Если этническая группа определяется как подмножество всего населения, то решение задачи о минимальном множестве представителей будет решением задачи создания политкорректного парламента.

Задача о минимальном множестве представителей двойственна задаче о покрытии множества. Заменим каждый элемент множества U множеством имен подмножеств, содержащих его. Теперь множества S и U поменялись ролями, т. к. мы ищем множество подмножеств множества U, чтобы покрыть все элементы в множестве S. Мы получили задачу о покрытии множества, поэтому можем использовать любую программу для ее решения, чтобы решить задачу о минимальном множестве представителей. Пример изображен на рис. 18.2.

Задача о покрытии множества должна быть, по меньшей мере, так же сложна, как и задача о вершинном покрытии, поэтому она тоже является NP-полной. На самом деле, она еще сложнее. Решение, выдаваемое аппроксимирующими алгоритмами для задачи о вершинном покрытии, хуже оптимального не более чем в два раза, а для задачи о покрытии множества самое лучшее решение хуже оптимального в $\Theta(\lg n)$ раз.

Самым естественным и эффективным подходом к решению задачи о покрытии множества будет использование "жадного"" эвристического алгоритма. Для начала выбираем самое мощное подмножество для покрытия, после чего удаляем все его элементы из универсального множества. Добавляем подмножество, содержащее наибольшее количество неохваченных элементов, и повторяем это действие, пока все элементы не будут покрыты. Количество подмножеств покрытия множества, выдаваемых таким эвристическим алгоритмом, никогда не превысит оптимальное более, чем в $\ln n$ раз, причем на практике этот коэффициент намного меньше.

Простейшая реализация "жадного" эвристического алгоритма просматривает на каждом шаге весь входной экземпляр из m подмножеств. Впрочем, используя такие структуры данных, как связные списки и очереди с приоритетами ограниченной высоты (см. *раздел 12.2*), можно реализовать "жадный" эвристический алгоритм с временем исполнения $O(S)$, где $S = \cup_{i=1}^{n'} |S_i|$ — размер входного экземпляра.

Полезно проверить существование элементов, содержащихся лишь в небольшом количестве подмножеств, в идеале, только в одном. При наличии таких элементов следует выбрать наибольшие содержащие их подмножества в самом начале работы алгоритма. В конечном счете нам все равно придется их выбрать, но они содержат другие элементы, покрытие которых потребует дополнительных затрат, если выбрать эти подмножества не с самого начала.

Метод имитации отжига, скорее всего, даст лучшие результаты, чем эти простые эвристические подходы. Чтобы гарантировать оптимальное решение, можно использовать поиск с возвратом, но выгода от этого не оправдает дополнительные расходы.

Еще один, более мощный подход, основан на переформулировке задачи о покрытии множества в терминах целочисленного программирования. Пусть целая переменная s_i, принимающая два значения, 0 и 1, обозначает, выбрано ли подмножество S_i для данного покрытия. Для каждого элемента x из универсального множества добавляется огра-

ничение $\sum_{x \in S_i} s_i \geq 1$, гарантирующее, что он будет покрыт хотя бы одним выбранным подмножеством. Минимальное покрытие множества удовлетворяет всем ограничивающим условиям и одновременно минимизирует $\sum_i s_i$. Эту задачу целочисленного программирования можно с легкостью обобщить до задачи о взвешенном покрытии множества (если допустить разную стоимость разных подмножеств). Ослабив эту задачу до задачи линейного программирования (т. е., позволив каждой переменной s_i находиться в диапазоне $0 \leq s_i \leq 1$, не ограничивая ее только двумя значениями 0 или 1), можно получить эффективный эвристический алгоритм, основанный на методах округления.

Реализации. Как "жадный" эвристический подход, так и подход с использованием целочисленного линейного программирования является достаточно простым в своей области, что его нужно реализовать с чистого листа.

Реализация на языке Pascal алгоритма исчерпывающего перебора для решения задачи укладки множества, а также для задачи о покрытии множества, дается в книге [SDK83]. Подробности см. в *разделе 19.1.10*.

Пакет SYMPHONY содержит процедуру для решения задачи разбиения множества методами смешанного линейного программирования. Загрузить ее можно с веб-сайта **http://branchandcut.org/SPP**.

Примечания

В работе [BP76] представлен классический обзор методов решения задачи о покрытии множества, а более свежий обзор аппроксимирующих методов с анализом их сложности дан в работе [Pas97]. Результаты исследований эвристических методов целочисленного программирования и точные алгоритмы решения задачи о покрытии множества изложены в работах [CFT99] и [CFT00]. Отличное обсуждение алгоритмов и правил сведения для задачи о покрытии множества представлено в книге [SDK83].

Среди хороших работ, посвященных "жадным" эвристическим алгоритмам решения задачи о покрытии множества, можно назвать книги [CLRS01] и [Hoc96]. Пример, демонстрирующий, что решение задачи о покрытии множества, выдаваемое "жадным" эвристическим алгоритмом, может быть в $\lg n$ хуже оптимального, представлен в работах [Joh74] и [PS98]. Но такой результат не является следствием дефекта в алгоритме. Доказана сложность получения приблизительного решения задачи о покрытии множества с коэффициентом, лучшим, чем $(1 - o(1))\ln n$, (см. [Fei98]).

Родственные задачи. Паросочетание (см. *раздел 15.6*), вершинное покрытие (см. *раздел 16.3*), укладка множества (см. *раздел 18.2*).

18.2. Задача укладки множества

Вход. Множество подмножеств $S = \{S_1, ..., S_m\}$ универсального множества $U = \{1, ..., n\}$.

Задача. Выбрать коллекцию (в идеале, небольшую) взаимно непересекающихся подмножеств из множества S, объединением которых будет универсальное множество (рис. 18.3).

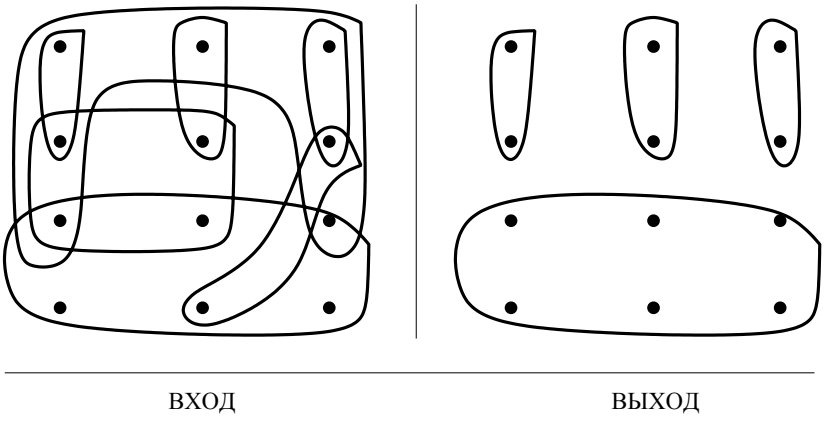


Рис. 18.3. Пример укладки множества

Обсуждение. Задачи укладки множества возникают в приложениях, имеющих строгие ограничивающие условия на разрешенное разбиение. Главной особенностью задач укладки множества является условие, согласно которому ни один элемент не может быть покрыт больше, чем одним выбранным подмножеством.

Эта задача в определенной степени аналогична задаче поиска независимого множества в графах (см. *раздел 16.2*), где требуется найти наибольшее подмножество вершин графа G, такое что каждое ребро инцидентно не более чем одной из выбранных вершин. Смоделируем задачу поиска подмножества вершин в виде задачи укладки множества. Пусть универсальное множество состоит изо всех ребер графа G, а подмножество S_i состоит изо всех ребер, инцидентных вершине v_i. Дополнительно определим одноэлементное множество для каждого ребра. Любая укладка множества определяет множество вершин, не имеющих общих ребер, другими словами, независимое множество. Одноэлементные множества используются для покрытия ребер, не охваченных выбранными вершинами.

Еще одним применением укладки множества будет комплектование экипажей самолетов. На каждый самолет авиакомпании нужно назначить экипаж, состоящий из двух пилотов и штурмана. К составу экипажа предъявляются определенные требования, такие как умение управлять самолетом данного типа, психологическая совместимость, а также рабочее расписание. Для всех возможных комбинаций экипажей и самолетов, каждая из которых представлена подмножеством элементов, нужно найти такой вариант, при котором каждый самолет и каждый член экипажа присутствуют только в одной комбинации. При данных ограничивающих условиях на подмножества нам нужно найти безупречную укладку.

Задача укладки множества включает в себя несколько типов NP-полных задач на множествах. Для их распознавания нужно ответить на следующие вопросы.

- *Обязательно ли, чтобы каждый элемент входил только в одно выбранное подмножество?* В задаче о точном покрытии требуется найти коллекцию подмножеств, в которой каждый элемент покрыт ровно один раз. Рассмотренная выше задача комплектования экипажей самолетов близка к задаче точного покрытия, т. к. в ней должны быть задействованы каждый самолет и каждый экипаж.

К сожалению, задача поиска точного покрытия ставит нас в ситуацию, аналогичную поиску гамильтонова цикла в графах. Если *действительно* нужно покрыть все элементы только по одному разу, а эта задача является NP-полной, то единственным способом ее решения является поиск с экспоненциальным временем исполнения. Стоимость такого подхода окажется очень высокой, если только вам не повезет, и вы не натолкнетесь на решение в начале поиска.

- *Есть ли у каждого элемента свое одноэлементное множество?* Ситуация намного улучшается, если нам достаточно частичного решения, например, такого, при котором каждый элемент универсального множества U определяется как одноэлементное подмножество множества S. Тогда мы можем расширить любую укладку множества до точного покрытия, покрыв не попавшие в укладку элементы U одноэлементными множествами. После этого наша задача сводится к поиску укладки множества минимальной мощности, решение которой можно получить с помощью эвристических методов.
- *Какова плата за двойное покрытие элементов?* В задаче о покрытии множества (см. *раздел 18.1*) включение одного элемента в несколько выбранных подмножеств не влечет за собой никаких отрицательных последствий. Однако в задаче точного покрытия такие многократные вхождения запрещены. Во многих приложениях запреты не такие строгие. В одном из подходов к решению таких задач можно увеличить стоимость выбора подмножеств, содержащих элементы, входящие в уже выбранные подмножества.

Самым лучшим подходом к решению задачи укладки множества является использование "жадных" эвристических алгоритмов, подобных тем, что применяются для решения задачи о покрытии множества (см. *раздел 18.1*). Если нам требуется найти укладку, содержащую наибольшее (наименьшее) количество подмножеств, тогда мы выбираем наименьшее (наибольшее) подмножество, удаляем все конфликтующие с ним подмножества из множества S и повторяем процесс. Как обычно, если мы дополним этот подход каким-либо методом исчерпывающего поиска или рандомизации (например, методом имитации отжига), мы, вероятно, получим более удачные укладки за счет увеличения времени работы.

Еще один, более мощный подход, основан на переформулировке задачи о покрытии множества в терминах целочисленного программирования, подобно тому, как мы делали это в задаче о покрытии множества. Пусть целая переменная s_i, принимающая два значения, 0 и 1, обозначает, выбрано ли подмножество S_i для данного покрытия. Для каждого элемента x из универсального множества добавляется ограничение $\sum_{x \in S_i} s_i = 1$, гарантирующее, что он будет покрыт *только* одним выбранным подмножеством. Минимизация или максимизация $\sum_i s_i$ при соблюдении этих ограничивающих условий позволяет нам регулировать количество подмножеств в покрытии.

Реализации. Поскольку задача о покрытии множества более популярна и легче решается, чем задача укладки множества, для ее решения проще найти подходящую реализацию. Рассматриваемые в *разделе 18.1* реализации без труда поддаются модификации, что позволяет соблюдать конкретные ограничивающие условия задачи упаковки множества.

Реализация на языке Pascal алгоритма исчерпывающего перебора для решения задачи упаковки множества, а также для задачи о покрытии множества дается в книге [SDK83]. Информацию по загрузке этих программ можно найти в *разделе 19.1.10*.

Пакет SYMPHONY содержит процедуру для решения задачи разбиения множества методами смешанного линейного программирования. Загрузить ее можно с веб-сайта **http://branchandcut.org/SPP**.

Примечания

Среди обзорных статей, посвященных задаче укладки множества, можно назвать такие, как [BP76] и [Pas97]. Стратегии предложения цены в аукционах, в которых товары предлагаются в наборах партий, обычно сводятся к решению задач укладки множества, как описывается в работе [dVV03].

Ослабленные ограничивающие условия для задач целочисленного программирования, соответствующих задаче укладки множества, представлены в работе [BW00]. Отличное обсуждение алгоритмов и правил сведения для задачи о покрытии множества представлено в книге [SDK83]. В ней же вы найдете описанное ранее приложение комплектования экипажей самолетов.

Родственные задачи. Независимое множество (см. *раздел 16.2*), вершинное покрытие (см. *раздел 16.3*).

18.3. Сравнение строк

Вход. Текстовая строка t из n символов, строка-образец p длиной в m символов.

Задача. Найти в текстовой строке t первое (или все) вхождение строки-образца p (рис. 18.4).

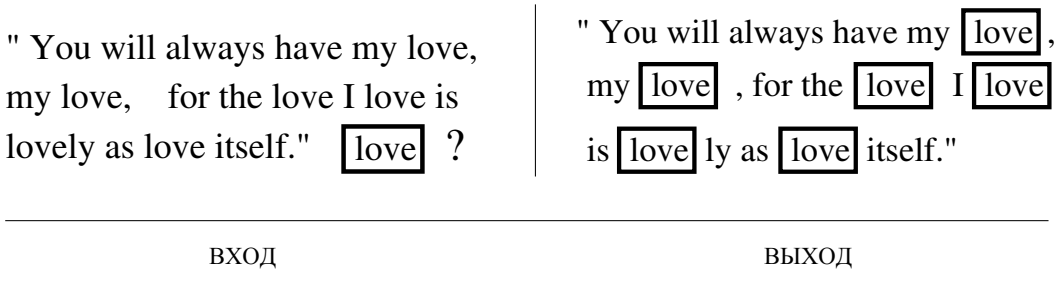


Рис. 18.4. Поиск в строке по образцу

Обсуждение. Задача сравнения строк возникает почти во всех приложениях обработки текста. Любой текстовый редактор имеет механизм поиска произвольной строки в текущем документе. Языки программирования Perl и Python обладают широкими возможностями поиска подстрок благодаря наличию встроенных примитивов сравнения строк, и это позволяет писать на них программы, способные фильтровать и модифицировать текст. Наконец, программы проверки правописания ищут в своем словаре каждое слово проверяемого текста и помечают слова, отсутствующие в словаре.

При выборе правильного алгоритма сравнения строк для конкретного приложения возникает несколько вопросов.

- *Какова длина образцов для поиска?* При недлинных образцах и нечастых запросах на поиск достаточно использовать простой алгоритм поиска со временем исполнения $O(mn)$. Для каждой возможной начальной позиции $1 \le i \le n - m + 1$ в строке поиска выполняется проверка подстроки длиной m символов на идентичность со строкой-образцом. Реализация этого алгоритма на языке C приводится в *разделе 2.5.3*.

  При очень коротких образцах ($m \le 5$) нет смысла пытаться улучшить этот простой алгоритм в надежде на повышение производительности. Кроме этого, для типичных строк ожидаемое время исполнения будет намного лучше, чем $O(mn)$, т. к. мы продвигаем образец дальше, как только обнаруживаем его несовпадение с подстрокой текста. Более того, этот простой алгоритм обычно исполняется за линейное время. Впрочем, наихудший случай вполне вероятен, — возьмем, например, образец $p = a^m$ и текст $t = (a^{m-1}b)^{n/m}$.

- *Как поступать с длинными текстами и образцами?* В действительности, поиск строк может осуществляться за линейное время в наихудшем случае. Обратите внимание, что при обнаружении несовпадающих символов нет необходимости возобновлять поиск с начала, т. к. префикс образца и текст одинаковы вплоть до точки несовпадения. Имея длинное частичное совпадение, заканчивающееся в позиции i, мы переходим на первый символ в образце или тексте, который может предоставить новую информацию о тексте в позиции $i + 1$. Алгоритм Кнута-Морриса-Пратта выполняет предварительную обработку образца для эффективного создания такой таблицы переходов. Подробности довольно сложны, но имеющийся алгоритм позволяет создавать короткие, простые программы.

- *Велика ли вероятность найти совпадение с образцом?* Алгоритм Бойера-Мура сравнивает образец с текстом справа налево, что позволяет избежать просмотра больших фрагментов текста при отсутствии совпадения. Допустим, что для образца поиска используется строка *abracadabra*, а в одиннадцатой позиции в тексте находится буква *x*. Этот образец не может совпасть с первыми одиннадцатью символами текста, поэтому следующей точкой проверки в тексте является двадцать второй символ. Если нам повезет, то проверить придется всего лишь n/m символов. В случае несовпадения алгоритм Бойера-Мура использует два набора таблиц переходов: один построен на основе текущих совпадений, а второй — на символе, вызвавшем несовпадение.

  Хотя этот алгоритм сложнее, чем алгоритм Кнута-Морриса-Пратта, он оправдывает себя на практике для строк-образцов длиной свыше пяти символов, при условии отсутствия частых вхождений образца в текст поиска. Время исполнения алгоритма в наихудшем случае равно $O(n + rm)$, где r — количество вхождений образца p в текст t.

- *Будет ли выполняться повторный поиск в одном и том же тексте?* Допустим, что вы создаете программу для многократных запросов на поиск в некоторой текстовой базе. Так как текст поиска не изменяется, имеет смысл создать структуру данных, позволяющую ускорить выполнение запросов. Подходящими структурами данных

для этой цели будут суффиксные деревья и суффиксные массивы, рассматриваемые в *разделе 12.3*.

- *Планируется ли поиск одного и того же образца в разных текстах?* Допустим, что вы создаете программу, чтобы отфильтровывать нецензурные выражения из текста. В этом случае набор образцов остается постоянным, а текст поиска может меняться. В таких приложениях может потребоваться найти все вхождения в текст любого из k разных образцов, при этом значение k может быть довольно большим.

  Линейное время для поиска каждого образца означает алгоритм с общим временем исполнения $O(k(m + n))$. Для больших значений k существует лучшее решение, создающее один конечный автомат, который распознает эти образцы и возвращается в соответствующее начальное состояние при любом несовпадении символов. Алгоритм Ахо-Корасика создает такой автомат за линейное время. Оптимизируя автоматы распознавания образов (см. *раздел 18.7*), можно достичь экономии памяти. Этот подход был использован в первоначальной версии программы fgrep.

  Иногда несколько образцов можно указать не в виде списка строк, а сжато, в форме регулярного выражения. Например, регулярное выражение $a(a + b + c)^*a$ соответствует любой строке алфавита (a, b, c), которая начинается и заканчивается на букву a. Самый лучший способ проверки, описывается ли входная строка данным регулярным выражением R, заключается в создании конечного автомата, эквивалентного R, с последующей эмуляцией этого автомата на данной строке. Подробности создания автомата на основе регулярных выражений приводятся в *разделе 18.7*.

  Когда вместо регулярных выражений для указания образцов используются контекстно-свободные грамматики, задача превращается в задачу синтаксического разбора (см. *раздел 8.6*).

- *Как поступить, если текст или образец содержит орфографические ошибки?* Рассматриваемые здесь алгоритмы пригодны только для точного сравнения строк. Если необходим допуск на орфографические ошибки, то задача превращается в задачу *нечеткого сравнения строк* (см. *раздел 18.4*).

Реализации. Программы на языке C коллекции strmat реализуют алгоритмы для точного сравнения образцов, включая реализацию нескольких вариантов алгоритмов Кнута-Морриса-Пратта и Бойера-Мура. Самым лучшим справочным материалом по этим алгоритмам является книга [Gus97]. Загрузить пакет можно с веб-сайта **http://www.cs.ucdavis.edu/~gusfield/strmat.html**.

Пакет для распознавания образов SPARE Parts, написанный на языке C++ (см. [WC04a]), предоставляет пригодные для коммерческого использования реализации всех основных вариантов классических алгоритмов сравнения строк (алгоритмы Кнута-Морриса-Пратта и Бойера-Мура), как для одного образца, так и для нескольких (алгоритмы Ахо-Корасика и Комменца-Вальтера). Загрузить пакет можно с веб-сайта **http://www.fastar.org/**.

Свободно доступны несколько версий программы сравнения регулярных выражений grep. Вариант GNU программы grep можно найти по адресу **http://directory.fsf.org/project/grep/**; эта версия заменяет такие предыдущие версии программы, как egrep и fgrep. Версия GNU grep использует гибрид быстрого детерминистического алгоритма

отложенных состояний с алгоритмом Бойера-Мура для поиска строк фиксированной длины.

Библиотека Boost содержит реализации на языке C++ алгоритмов обработки строк, включая поиск.

Примечания

Все книги по алгоритмам обработки строк, включая такие как [CHL07], [NR07] и [Gus97] , содержат исчерпывающее обсуждение точного сравнения строк. Хорошие описания алгоритмов Бойера-Мура (см. [BM77]) и Кнута-Морриса-Пратта (см. [KMP77]) представлены в книгах [BvG99], [CLRS01] и [Man89]. В истории создания алгоритмов сравнения строк были и неудачи, — некоторые из опубликованных работ содержал и ошибки. Подробности см. в книге [Gus97].

В книге [Aho90] дается хороший обзор алгоритмов поиска образцов в строках, в частности, алгоритмов поиска с помощью регулярных выражений. Алгоритм Ахо-Корасика описывается в работе [AC75].

Эксперименты по сравнению алгоритмов обработки строк представлены в книгах [DB86], [Hor80], [Lec95] и [dVS82]. Качество работы каждого конкретного алгоритм а зависит от свойств строк и размера алфавита. Для работы с дли н н ы м и образцами и большими текстами я рекомендую использовать самые лучшие реализации алгоритма Бойера-Мура, которые вы сможете найти.

В алгоритме Карпа-Рабина (см. [KR87]) для сравнения строк используется хэш-функция, что позволяет получить линейное ожидаемое время исполнения. Но время исполнения этого алгоритма в наихудшем случае остается квадратичным, а его производительность на практике оказывается несколько хуже, чем у описанных ранее методов сравнения символов. Этот алгоритм обсуждается в *разделе 3.7.2*.

Родственные задачи. Суффиксные деревья (см. *раздел 12.3*), нечеткое сравнение строк (см. *раздел 18.4*).

18.4. Нечеткое сравнение строк

Вход. Текстовая строка t и строка-образец p.

Задача. Найти самый дешевый способ преобразования строки t в строку p, в котором используются вставки, удаления и замены символов (рис. 18.5).

Обсуждение. Задача нечеткого сравнения строк является одной из фундаментальных задач, т. к. мы живем в мире, где нет четких границ. Программы проверки правописания должны находить наиболее подходящее слово для любой строки символов, отсутствующей в словаре (т. е. базе данных образцов). Предоставляя поддержку эффективного поиска сходных последовательностей в больших базах данных последовательностей ДНК, компьютерная программа BLAST коренным образом изменила направление исследований в молекулярной биологии. Допустим, что вы исследовали некоторый ген человека, и оказалось, что он аналогичен гену, вырабатывающему гемоглобин в организме крысы. Скорее всего, исследуемый ген также отвечает за выработку гемоглобина, а разница между этими двумя генами является результатом эволюционных мутаций.

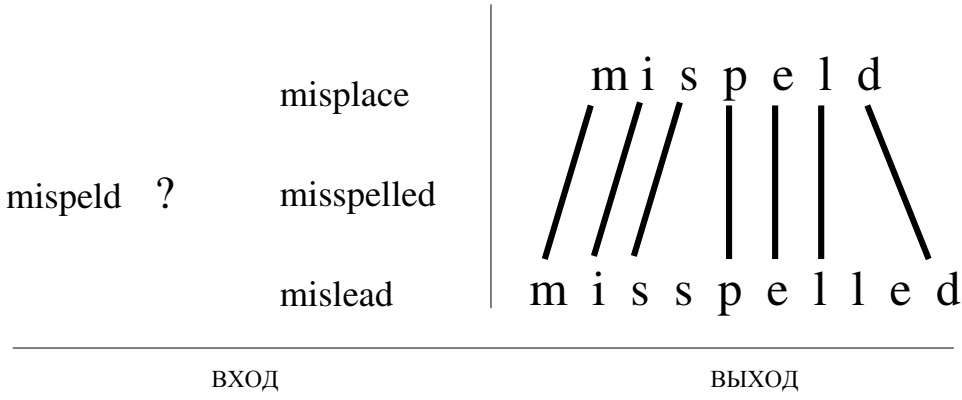


Рис. 18.5. Нечеткое сравнение строк

Мне однажды пришлось столкнуться с задачей нечеткого сравнения строк при оценке качества работы системы оптического распознавания текста. В ней нужно было сравнить ответы, выдаваемые системой для тестового документа, с правильными результатами. Чтобы улучшить систему, нам требовалось идентифицировать ошибочно распознаваемые буквы и "мусор", т. е. несуществующие буквы. Было принято решение выполнить нечеткое сравнение ответов и тестовых документов. В таком случае вставки и удаления соответствовали бы "мусору", а замены указывали бы на ошибки в системе распознавания. Аналогичный принцип используется в программах сравнения файлов, которые выявляют строки, отличающиеся в двух разных версиях файла.

Когда ошибки недопустимы, задача сводится к точному сравнению строк, которая рассматривается в *разделе 18.3*. Здесь же мы будем рассматривать только строки с ошибками.

Динамическое программирование предоставляет нам базовый подход к нечеткому сравнению строк. Пусть $D[i, j]$ обозначает стоимость редактирования первых i символов строки образца p в первые j символов текста t. Отсюда следует существование рекуррентного соотношения, т. к. мы должны были что-то сделать с концевыми символами p_i и t_j. У нас имелись следующие возможности: зафиксировать совпадение или заменить один символ другим, удалить p_i или, наконец, вставить символ, совпадающий с t_j. Таким образом, значение $D[i, j]$ является минимальным из следующих трех значений стоимости:

- если $p_i = t_j$, то $D[i-1, j-1]$, иначе $D[i-1, j-1]$ + стоимость замены;
- $D[i-1, j]$ + стоимость удаления p_i;
- $D[i, j-1]$ + стоимость удаления t_j.

Общая реализация этого алгоритма на языке C и более подробное обсуждение приводятся в *разделе 8.2*. Прежде чем использовать это рекуррентное соотношение, ответьте на несколько вопросов.

- *Образец должен совпасть со всем текстом или только с подстрокой?* Разница между алгоритмами сравнения строк и алгоритмами сравнения подстрок определяется граничными условиями этого рекуррентного соотношения. Допустим, что мы хотим сравнить весь образец со всем текстом. Тогда стоимость $D[i, 0]$ должна рав-

няться стоимости удаления первых i символов образца, поэтому $D[i, 0] = i$. Подобным образом, $D[0, j] = j$.

Теперь допустим, что образец может встретиться в любом месте текста. Корректным значением стоимости $D[0, j]$ будет 0, т. к. в этом случае не должно быть штрафа за то, что выравнивание начинается в j-й позиции текста. Стоимость $D[i, 0]$ попрежнему остается равной i, т. к. единственным способом получения совпадения первых i символов образца с пустой строкой является удаление всех этих символов. Стоимость самого лучшего совпадения образца с подстрокой будет $\min_{k=1}^{n} D[m, k]$.

- *Как определить стоимость операций замены, вставки и удаления?* Базовый алгоритм можно с легкостью модифицировать, установив разную стоимость вставки, удаления и замены символов. Конкретное значение стоимости каждой операции зависит от того, что вы намереваетесь делать с выровненными текстами.

  Самым распространенным вариантом является назначение одинаковой стоимости для операций вставки, удаления и замены. Более высокая стоимость замены, чем суммарная стоимость вставки и удаления гарантирует, что замена никогда не будет выполнена, т. к. редактирование вне строки обойдется дешевле. Если выполняются только операции вставки и удаления, то задача сводится к задаче поиска максимальной общей подстроки, рассматриваемой в *разделе 18.8*. Часто бывает полезно слегка изменить стоимость расстояний редактирования и исследовать полученные результаты, повторяя процесс до тех пор, пока не будут определены наилучшие параметры для данной задачи.

- *Как узнать, какие операции фактически привели к выравниванию строк?* Ранее было сказано, что рекуррентное соотношение дает только стоимость оптимального выравнивания образца с текстом, но не последовательность операций редактирования, приводящую к этому выравниванию. Получить список операций можно, двигаясь в обратном направлении от матрицы полной стоимости D. Чтобы добраться до ячейки $D[m, n]$, мы должны были выйти из ячейки $D[m-1, n]$ (удаление образца/вставка текста), $D[m, n-1]$ (удаление текста/вставка образца) или $D[m-1, n-1]$ (замена/совпадение). Опцию, которая была выбрана фактически, можно выяснить по этим значениям стоимости и по символам p_m и t_n. Продолжая двигаться в обратном направлении к предыдущей ячейке, можно восстановить все операции редактирования. Реализация этого алгоритма на языке C приводится в *разделе 8.2*.

- *Как поступить, если обе строки очень похожи друг на друга?* Алгоритм динамического программирования позволяет найти кратчайший путь в решетке размером $m \times n$, где стоимость каждого ребра зависит от представляемой им операции. Чтобы найти выравнивание, включающее в себя комбинацию из не более чем d вставок, удалений и замен, нужно только обойти полосу из $O(dn)$ ячеек на расстоянии d в каждую сторону от главной диагонали. Если в этой полосе нет выравнивания низкой стоимости, его нет и во всей матрице стоимости.

  Можно также использовать фильтрацию — быстрое удаление тех участков строки, в которых образец наверняка отсутствует. Разбиваем образец длиной m символов на $d+1$ частей. Если существует соответствие, имеющее максимум d отличий, то хотя бы одна из этих частей имеет точное совпадение в оптимальном выравнивании. Та-

ким образом, мы можем идентифицировать все возможные точки приблизительных совпадений, выполняя точный поиск по нескольким фрагментам образца, а затем более внимательно рассмотреть только потенциальные кандидаты.

- *Насколько длинной является строка-образец?* Недавно появился новый подход к сравнению строк, в котором используется то обстоятельство, что современные компьютеры могут выполнять операции на словах длиной в 64 бита. В таком машинном слове можно разместить восемь 8-битовых символов ASCII, что стимулирует разработку алгоритмов с параллелизмом на уровне битов, в которых за одну операцию выполняется несколько сравнений.

  Основная идея далеко не тривиальна. Для каждой буквы α алфавита создаем битовую маску B_α, в которой i-й бит $B_\alpha[i]$ равен 1 тогда и только тогда, когда i-м символом образца является α. Допустим теперь, что имеется такой битовый вектор совпадения M_j для j-й позиции в текстовой строке, что $M_j[i] = 1$ тогда и только тогда, когда первые i битов образца точно совпадают с символами текста, начиная с $(j - i + 1)$-го и заканчивая j-м. Мы можем найти все биты M_{i+1} посредством лишь двух операций, а именно, сдвинув вектор M_j на один бит вправо, а потом выполнив для него побитовую операцию И с маской B_α, где α — символ в $(j + 1)$-й позиции текста.

  Такой алгоритм с параллелизмом на уровне битов, обобщенный до нечеткого сравнения, используется в рассматриваемой далее программе agrep. Подобные алгоритмы легко поддаются реализации, а работают они во много раз быстрее алгоритмов динамического программирования.

- *Как минимизировать требования к памяти?* Квадратичный объем памяти, требуемый для хранения таблицы динамического программирования, представляет проблему, более серьезную, чем время исполнения соответствующих алгоритмов. К счастью, для вычисления $D[m, n]$ требуется только $O(\min(m, n))$ памяти. Для вычисления окончательного значения нужно сопровождать только две активные строки (или столбца) матрицы. Вся матрица понадобится только в том случае, когда потребуется воссоздать последовательность операций, в результате которых было получено данное выравнивание.

  Для эффективного вычисления за линейное время оптимального выравнивания можно использовать рекурсивный алгоритм Хиршберга. За один проход описанного выше алгоритма с линейной сложностью по памяти для вычисления $D[m, n]$ мы выясняем, какая ячейка среднего элемента $D[m/2, x]$ была использована для оптимизации $D[m, n]$. Это сводит нашу задачу к задаче поиска наилучших путей от $D[1, 1]$ до $D[x/2, x]$ и от $D[m/2, x]$ до $D[m/2, n]$, причем оба пути могут быть найдены рекурсивно. Каждый раз мы исключаем из рассмотрения половину элементов матрицы, поэтому общее время остается равным $O(mn)$. Этот алгоритм с линейной сложностью по памяти обладает хорошей производительностью на длинных строках, но программировать его затруднительно.

- *Следует ли по-разному оценивать многократные повторы операций вставки и удаления?* Во многих приложениях сравнения строк оказывается предпочтение выравниваниям, в которых операции вставки и/или удаления сгруппированы в небольшое количество последовательностей. Удаление слова из текста, по идее, должно стоить

меньше, чем удаление соответствующего количества одиночных символов, т. к. при этом выполняется одна (хотя и сложная) операция редактирования.

Применение штрафов за появление разрывов при сравнении строк позволяет корректно учитывать такие операции. Обычно для каждой операции вставки/удаления t последовательных символов устанавливается стоимость $A + Bt$, где A — стоимость создания разрыва, а B — стоимость удаления одного символа. Если значение A велико по сравнению со значением B, то при выравнивании появится стимул выполнять относительно небольшое количество последовательных операций удаления.

Сравнение строк при таких *аффинных* штрафах за разрывы можно выполнять за квадратичное время, как и обычное вычисление расстояния редактирования. Мы будем использовать различные рекуррентные соотношения E и F для операций вставки и удаления, чтобы вычислить стоимость пребывания в режиме разрыва, предполагая, что цена за создание разрыва уже заплачена:

$V(i,j) = \max(E(i,j), F(i,j), G(i,j))$

$G(i,j) = V(i-1, j-1) + \text{match}(i,j)$

$E(i,j) = \max(E(i,j-1), V(i,j-1) - A) - B$

$F(i,j) = \max(F(i-1,j), V(i-1,j) - A) - B$

При постоянном времени обращения к каждой ячейке время исполнения этого алгоритма равно $O(mn)$.

- *Считаются ли похожими строки с одинаковым произношением?* Для некоторых приложений лучше подходят другие модели нечеткого сравнения строк. Особый интерес представляет схема хэширования Soundex, в которой одинаково звучащим словам присваивается одинаковый индекс. Эта схема может быть полезной при проверке, не являются ли два по-разному написанных слова в действительности одним и тем же словом. Например, мою фамилию часто пишут как "Skina", "Skinnia", Schiena", а иногда и "Skiena". Всем этим по-разному написанным словам присваивается одинаковый индекс Soundex — S25.

  Алгоритм Soundex отбрасывает гласные, непроизносимые и повторяющиеся буквы, а потом назначает оставшимся буквам числа в соответствии со следующими классами: BFPV — 1, CGJKQSXZ — 2, DT — 3, L — 4, MN — 5, R — 6. Код слова состоит из первой буквы, за которой следуют максимум три цифры. Хотя такой подход кажется не очень естественным, на практике он работает довольно хорошо. А практика уже довольно продолжительная — Soundex используется с 1920-х годов.

Реализации. Для нечеткого сравнения строк существует несколько отличных программных средств. Программа agrep (см. [WM92a, WM92b]) поддерживает поиск в тексте с орфографическими ошибками. Последнюю версию этой программы можно загрузить с веб-страницы **http://www.tgries.de/agrep**. Программа nrgrep (см. [Nav01b]) сочетает параллелизм на уровне битов с фильтрацией и имеет постоянное время исполнения, хотя не всегда работает быстрее, чем agrep. Загрузить программу можно с веб-сайта **http://www.dcc.uchile.cl/~gnavarro/software/**.

Библиотека TRE поиска совпадений с регулярными выражениями предназначена для поиска точных и приблизительных совпадений и обладает более широкими возможно-

стями, чем программа agrep. Временная сложность программы в худшем случае равна $O(nm^2)$, где m — длина списка используемых регулярных выражений. Загрузить программу можно с веб-сайта **http://www.dcc.uchile.cl/~gnavarro/software/**.

В Wikipedia можно найти программы для вычисления расстояния редактирования, в частности, расстояния Левенштейна, написанные на разных языках (включая Ada, C++, Emacs Lisp, Io, JavaScript, Java, PHP, Python, Ruby, VB и C#). Дополнительную информацию можно найти на веб-сайте **http://en.wikibooks.org/wiki/Algorithm_Implementation/Strings/Levenshtein_distance**.

Примечания

В последнее время наблюдаются определенные успехи в области нечеткого сравнения строк, особенно в алгоритмах, применяющих параллелизм на уровне битов. Алгоритмам обработки строк посвящены книги [CHL07] и [Gus97], а самым лучшим справочником по методам сравнения строк является [NR07].

Считается, что базовый алгоритм динамического программирования для получения выравниваний впервые был описан в работе [WF74], хотя, по-видимому, он был известен и раньше. Широта диапазона применения нечеткого сравнения строк была продемонстрирована в книге [SK99], которая по сей день является полезным историческим справочником в данной области. Обзоры решений задачи нечеткого сравнения строк представлены в работах [HD80] и [Nav01a]. Расстояние редактирования между двумя строками иногда называют *расстоянием Левенштейна*. Алгоритм Хиршберга с линейной сложностью по памяти (см. [Hir75]) рассматривается в книгах [CR03] и [Gus97].

В работе [MP80] представлен алгоритм для вычисления расстояния редактирования между строками длиной в m и n символов за время $O(mn/\log(\min\{m, n\}))$ для алфавитов постоянного размера. В алгоритме использованы идеи из алгоритма четырех русских для умножения булевых матриц (см. [ADKF70]).

Формулировка задачи нечеткого сравнения строк в терминах поиска кратчайшего пути позволяет создать множество алгоритмов, дающих хорошие результаты для небольших расстояний редактирования, включая алгоритмы с временем исполнения $O(n\lg n + d^2)$ (см. [Mye86]) и $O(dn)$ (см. [LV88])

Максимальную возрастающую последовательность можно вычислить за время $O(n\lg n)$ (см. [HS77]), как описано в [Man89].

В числе алгоритмов нечеткого сравнения строк, применяющих параллелизм на уровне битов, можно назвать алгоритм Майерса (Myers) (см. [Mye99b]) с временем исполнения $O(mn/w)$, где w — количество бит компьютерного слова. Результаты экспериментальных исследований алгоритмов с параллелизмом на уровне битов представлены в работах [FN04], [HFN05] и [NR,00].

Система Soundex была изобретена и запатентована Оделл (M. K. Odell) и Расселлом (R. C. Russell). Описание этой системы можно найти в работах [BR95] и [Knu98]. Недавняя разработка Metaphone (см. [BR95] и [Par90]) представляет собой попытку создания системы, превосходящей Soundex. Применение таких систем фонетического хэширования для унификации названий задач рассматривается в работе [LMS06].

Родственные задачи. Поиск строк (см. *раздел 18.3*), поиск максимальной общей подстроки (см. *раздел 18.8*).

18.5. Сжатие текста

Вход. Текстовая строка S.

Задача. Преобразовать строку S в более короткую строку S', из которой можно правильно воссоздать исходную строку S (рис. 18.6).

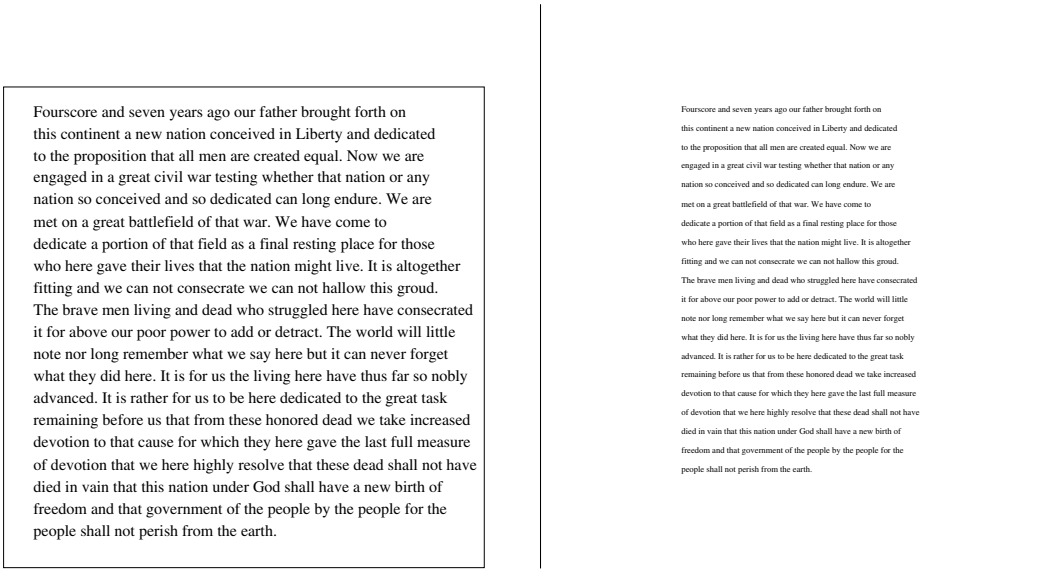


Рис. 18.6. Сжатие текста

Обсуждение. Хотя емкость внешних устройств хранения данных удваивается каждый год, ее все время не хватает. Снижение цен на устройства хранения ничуть не уменьшило интерес к сжатию данных, вероятно, потому что данных, подлежащих сжатию, становится все больше. Сжатие данных является алгоритмической задачей, в которой требуется найти экономичную кодировку для файла указанного типа. Развитие компьютерных сетей поставило новую цель перед задачей сжатия данных — повышение эффективной пропускной способности сети посредством уменьшения количества передаваемых битов.

Создается впечатление, что разработчикам *нравится* изобретать специальные методы сжатия данных для каждого конкретного приложения. Иногда производительность этих специализированных методов даже превосходит производительность методов общего назначения. При выборе правильного алгоритма сжатия возникает несколько вопросов.

- *Требуется ли точное восстановление исходных данных после сжатия?* Основной вопрос, возникающий при сжатии данных, заключается в том, выполнять сжатие с потерями или без потерь. Приложения для хранения документов обычно требуют обратимого сжатия (без потерь), т. к. пользователи не хотят, чтобы их данные под-

вергались изменениям. При сжатии изображений или видеофайлов точность данных не настолько важна, т. к. небольшие погрешности незаметны для зрителя. Применение сжатия с потерями позволяет получить значительно более высокую степень сжатия, вследствие чего этот тип сжатия применяется в большинстве приложений для сжатия аудио- и видеоданных и изображений.

- *Можно ли упростить данные перед сжатием?* Самым эффективным способом увеличения свободного дискового пространства является удаление ненужных файлов. Аналогичным образом любая предварительная обработка, уменьшающая объем информации в файле, приводит к более эффективному сжатию. Постарайтесь выяснить, можно ли удалить из файла избыточные пробельные символы, преобразовать все буквы текста в заглавные или снять форматирование.

  Особый интерес представляет упрощение данных с помощью *преобразования Барроуза-Уилера*. По ходу этого преобразования выполняется сортировка всех n циклических сдвигов n-символьной входной строки, а затем возвращается последний символ каждого сдвига. Например, строка *ABAB* имеет циклические сдвиги *ABAB*, *BABA*, *ABAB* и *BABA*. После сортировки получаем строки *ABAB*, *ABAB*, *BABA* и *BABA*. Считывая последний символ каждой из этих строк, получаем результат преобразования — *BBAA*.

  При условии, что последний символ входной строки уникален (например, представляет собой символ конца строки), это преобразование является полностью обратимым. Строка, предварительно обработанная с помощью преобразования Барроуза-Уилера, сжимается на 10–15% лучше, чем первоначальный текст, т. к. повторяющиеся слова превращаются в блоки повторяющихся символов. Кроме этого, это преобразование можно осуществить за линейное время.

- *Что делать, если алгоритм сжатия запатентован?* Некоторые алгоритмы сжатия данных были запатентованы. Одним из самых известных примеров является рассматриваемая далее версия LZW алгоритма Лемпеля-Зива. К счастью, в настоящее время срок действия этого патента истек, чего нельзя сказать об алгоритме сжатия JPEG, который все еще является предметом судебных разбирательств. Обычно у любого алгоритма сжатия имеются версии, которыми можно пользоваться без ограничений, причем работают они так же хорошо, как и запатентованный вариант.

- *Как сжимать изображения?* Самым простым обратимым алгоритмом сжатия изображений является *кодирование длин серий*. При таком способе сжатия последовательности одинаковых пикселов (серии) заменяются одним экземпляром пиксела и числом, указывающим длину последовательности. Этот метод хорошо работает с двоичными представлениями изображений, имеющими большие области одинаковых пикселов, например, с отсканированным текстом. Но на изображениях, имеющих много градаций цвета и/или содержащих шум, производительность падает. Сильное влияние на качество сжатия оказывают два фактора: выбор размера поля, содержащего длину последовательности, и выбор порядка обхода при преобразовании двумерного изображения в поток пикселов.

  Для профессиональных приложений сжатия аудиоданных, видеоданных и изображений я рекомендую использовать один из популярных методов сжатия с потерями и не пытаться реализовать свой собственный. Стандартным высокопроизводитель-

ным методом сжатия изображений является JPEG, а MPEG предназначен для сжатия видеоданных.

- *Должно ли сжатие выполняться в реальном времени?* Быстрое восстановление данных нередко оказывается важнее, чем их быстрое сжатие. Например, видеоданные на YouTube сжимаются только один раз, но восстанавливаются при каждом его воспроизведении. Впрочем, существует и противоположный пример. Операционной системе для увеличения эффективной емкости диска посредством сжатия файлов требуется симметричный алгоритм, обладающий также быстрым временем сжатия.

Существуют десятки алгоритмов для сжатия текста, но их можно разбить на две группы в соответствии с применяемым подходом. *Статические алгоритмы*, такие как метод Хаффмана, создают одну кодовую таблицу, исследуя полностью весь документ. *Адаптивные алгоритмы*, такие как алгоритм Лемпеля-Зива, динамически создают кодовую таблицу, которая адаптируется под локальное распределение символов документа. В большинстве случаев следует предпочитать адаптивный алгоритм.

- *Код Хаффмана.* Алгоритм Хаффмана заменяет каждый символ алфавита кодовой строкой битов переменной длины. Использование восьми бит для кодирования каждой буквы неэкономично, т. к. некоторые символы (например, буква "e") встречаются намного чаще, чем другие (например, буква "q"). При кодировании методом Хаффмана букве "e" присваивается короткий код, а букве "q" — более длинный, таким образом сжимая текст.

  Код Хаффмана можно создать, используя "жадный" алгоритм. Сначала сортируем символы в порядке возрастания частоты их вхождения в текст. Два самых редких символа x и y объединяются в один новый символ xy с частотой вхождения, равной сумме частот вхождения двух его родительских символов. Таким образом мы получаем меньший набор символов. Эта операция повторяется $n - 1$ раз, пока все символы не будут слиты попарно. Посредством таких операций объединения строится корневое двоичное дерево, в котором исходные символы алфавита представлены листьями. Биты слова двоичного кода для каждого символа определяются выбором правого или левого пути в проходе от корня к листу. При написании программы упорядочивания символов по частоте использования можно применять очереди с приоритетами, что позволит выполнять эту операцию за время $O(n\lg n)$.

  Несмотря на свою популярность, алгоритм Хаффмана имеет три недостатка. Во-первых, для того чтобы закодировать документ, по нему необходимо выполнить два прохода — первый, чтобы создать таблицу кодирования, а второй, чтобы выполнить само кодирование документа. Во-вторых, таблицу кодирования нужно явно сохранять вместе с закодированным документом, чтобы можно было декодировать его, и это приводит к неэкономному расходу памяти при кодировании коротких документов. Наконец, алгоритм Хаффмана эффективен при неравномерном распределении символов, в то время как адаптивные алгоритмы способны распознавать повторяющиеся последовательности, например, *0101010101*.

- *Алгоритмы Лемпеля-Зива.* Алгоритмы Лемпеля-Зива (включая популярный вариант LZW) сжимают текст, создавая таблицу кодирования по ходу чтения документа. Эта таблица корректируется на каждой позиции в тексте. Благодаря использованию хо-

рошо продуманного протокола, программы кодирования и декодирования работают с одной и той же таблицей, что позволяет избежать потери информации.

Алгоритмы Лемпеля-Зива создают кодовые таблицы часто встречающихся подстрок, размер которых может быть очень большим. Таким образом алгоритмы этого типа могут использовать часто встречающиеся слоги, слова и фразы, чтобы выполнить кодирование наилучшим образом. Алгоритм приспосабливается к локальным изменениям в распределении элементов текста, что является важным, т. к. многие документы обладают значительной пространственной локальностью.

Замечательным свойством алгоритма Лемпеля-Зива является его устойчивость при обработке данных разных типов. Разработать для конкретного приложения специализированный алгоритм, превосходящий алгоритм Лемпеля-Зива, довольно трудно. Я рекомендую не предпринимать таких попыток. Если имеется возможность устранить специфическую для приложения избыточность на стадии предварительной обработки, воспользуйтесь ею. Но не тратьте понапрасну время, пытаясь создать собственный алгоритм сжатия. Вряд ли вам удастся получить значительно лучшие результаты, чем выдает программа gzip или другая распространенная утилита.

Реализации. Самой популярной программой сжатия текста, вероятно, является программа gzip, которая реализует публично доступный вариант алгоритма Лемпеля-Зива. Эта программа распространяется с лицензией GNU; загрузить ее можно с веб-сайта **http://www.gzip.org**.

Всегда приходится искать разумный компромисс между коэффициентом сжатия и временем работы алгоритма. Альтернативой программе gzip является программа сжатия bzip2, в которой используется преобразование Барроуз-Уилера. Она обеспечивает лучший коэффициент сжатия, чем программа gzip, но за счет увеличения времени исполнения. Создатели некоторых алгоритмов впадают в крайность, жертвуя скоростью ради повышенной компактности. Типичные представители таких алгоритмов собраны на веб-сайте **http://www.cs.fit.edu/~mmahoney/compression**.

Авторитетное сравнение программ сжатия, включающее в себя ссылки на все доступное программное обеспечение, представлено на веб-сайте **http://www.maximumcompression.com/**.

Примечания

Существует большое количество книг, посвященных сжатию данных. Из недавно вышедших можно назвать книги [Say05] и [Sal06]. Также можно порекомендовать уже не новую книгу [BCW90]. Обзор алгоритмов сжатия текстов приведен в работе [CL98].

Хорошее описание алгоритма Хаффмана (см. [Huf52]) можно найти в книгах [AHU83], [CLRS] и [Man89]. Алгоритм Лемпеля-Зива и его варианты описаны в работах [Wel84] и [ZL78]. Преобразование Барроуз-Уилера было представлено в работе [BW94].

Главным форумом по обмену информацией в области сжатия данных является ежегодная тематическая конференция института IEEE. Подробности см. на веб-сайте **http://www.cs.brandeis.edu/~dcc**. Сжатие данных является хорошо развитой технической областью, и в последнее время усилия разработчиков сосредоточены на весьма незначительных улучшениях алгоритмов, особенно в случае сжатия текстовых данных. Однако радует то обстоятельство, что ежегодная конференция IEEE проводится на горнолыжном курорте мирового класса в штате Юта.

Родственные задачи. Поиск минимальной общей надстроки (см. *раздел 18.9*), криптография (см. *раздел 18.6*).

18.6. Криптография

Вход. Открытое сообщение T или зашифрованный текст E и ключ k.

Задача. С помощью ключа k зашифровать сообщение T (расшифровать текст E), получив в результате зашифрованный текст E (открытое сообщение T) (рис. 18.7).

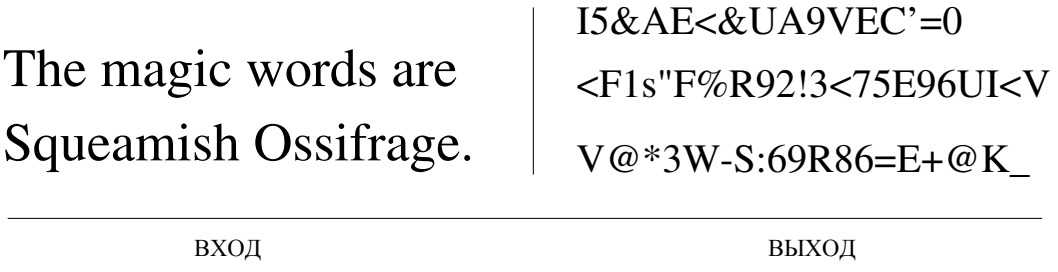


Рис. 18.7. Пример шифрования текста

Обсуждение. Поскольку широкое распространение компьютерных сетей предоставляет все больше возможностей несанкционированного доступа к конфиденциальной информации, криптография играет важную роль в сохранении целостности такой информации. Криптография повышает безопасность обмена сообщениями, позволяя обрабатывать их таким образом, что их будет невозможно прочесть, если они попадут в чужие руки. Хотя этой дисциплине, по крайней мере, две тысячи лет, ее алгоритмические и математические основы лишь недавно сформировались настолько, что появилась возможность рассуждать о доказуемо безопасных системах шифрования.

Идеи и приложения шифрования выходят за рамки общеизвестных понятий "шифрования" и "аутентификации". Сейчас эта область включает такие важные математические конструкции, как криптографические хэши, цифровые подписи, а также базовые протоколы, предоставляющие гарантии безопасности.

Существует три класса систем шифрования:

- *шифр Цезаря.* В самых старых шифрах каждый символ алфавита заменялся другим символом этого же алфавита. В наиболее слабых шифрах выполняется циклический сдвиг алфавита на заданное количество символов (часто 13), вследствие чего они имеют только 26 возможных ключей (речь идет об английском алфавите — *прим. перев*). Лучше использовать произвольную перестановку букв, что дает 26! возможных ключей. Но даже и в этом случае такие системы шифрования можно с легкостью взломать, подсчитав частоту появления каждого символа в тексте и применив знание частотного распределения букв в текстах данного алфавита. Хотя существуют варианты этого шифра, более устойчивые к взлому, ни один из них не будет таким надежным, как шифр AES или RSA;
- *блочные шифры.* В шифрах этого типа биты текста многократно перемешиваются в соответствии с заданным ключом. Классическим примером такого шифра являет-

ся шифр Data Encryption Standard (DES). Хотя он был одобрен в 1976 г. в качестве Федерального стандарта США для обработки информации, в настоящее время 56-битовый ключ этого шифра считается слишком коротким для приложений, требующих значительного уровня безопасности. В частности, компьютер, созданный специально для взлома этого шифра, продемонстрировал, что этот шифр можно взломать быстрее, чем за один день. С 19 мая 2005 г. шифр DES официально перестал быть Федеральным стандартом США и был заменен более сильным шифром *Advanced Encryption Standard* (AES).

Но простой вариант шифра DES под названием "тройной DES" позволяет осуществлять шифрование с помощью ключа длиной в 112 битов за счет троекратного применения шифра DES с двумя 56-битовыми ключами. Для этого текст сначала зашифровывается ключом *key1*, потом расшифровывается ключом *key2*, а затем опять зашифровывается ключом *key1*. Использование трех раундов шифрования вместо двух имеет целью обратную совместимость со старым шифром DES, у которого *key1* = *key2*. Национальный институт стандартов и технологий США (NIST) недавно одобрил применение тройного DES для шифрования важной правительственной информации вплоть до 2030 г.;

- *шифрование с открытым ключом.* Если вы боитесь, что ваши недоброжелатели читают ваши сообщения, вы также должны держать в секрете ключ, необходимый для их расшифровки. Однако в системах с открытым ключом для шифрования и дешифрования сообщений используются разные ключи. Так как ключ шифрования бесполезен для расшифровки, он может быть открытым, без риска для безопасности. Решение о распределении ключей лежит в основе упеха шифрования с открытым ключом.

  Классическим примером системы шифрования с открытым ключом является система RSA, названная так в честь ее изобретателей — Ривеста (Rivest), Шамира (Shamir) и Адлемана (Adleman). Безопасность системы RSA основана на вычислительной сложности задач разложения на множители и проверки чисел на простоту (см. *раздел 13.8*). Процесс шифрования протекает быстро, т. к. для создания ключа используется проверка на простоту, а сложность расшифровки вытекает из сложности разложения на множители. Однако система шифрования RSA работает медленно по сравнению с другими системами. В частности, она примерно в 100–1 000 раз медленнее, чем шифр DES.

Важнейшим фактором, влияющим на выбор системы шифрования, является требуемый уровень безопасности. От кого вы пытаетесь защитить ваши сообщения — от своей бабушки, квартирных воров, организованной преступности или спецслужб? Если вы можете использовать современную реализацию шифра AES или RSA, вы будете чувствовать себя в безопасности, по крайней мере, в ближайшее время. Однако постоянно растущая мощность компьютеров способна на удивление быстро расшатать криптосистему. Вспомните, что шифр DES продержался в качестве надежной системы менее тридцати лет. Обязательно используйте ключи максимальной длины и следите за новостями в области шифрования, если вы планируете долгосрочное хранение конфиденциальной информации.

Что касается меня лично, признаюсь, что я использую DES для шифрования задач выпускного экзамена в конце каждого семестра. Этого шифра оказалось более чем доста-

точно, когда один решительно настроенный студент проник в мой офис в поисках задач. Результат был бы иным, если бы взломщиком оказался агент спецслужб. При этом важно понимать, что самые серьезные бреши в безопасности обусловлены человеческим фактором, а не качеством алгоритма. Использовать достаточно длинный, трудный для отгадывания пароль и не записывать его на бумажке гораздо важнее, чем излишне усердствовать в выборе алгоритма шифрования.

При одинаковой длине ключа большинство симметричных систем шифрования труднее поддается взлому, чем системы с открытым ключом. Это означает, что для симметричного шифрования можно использовать намного более короткие ключи, чем для шифрования с открытым ключом. Институт NIST и лаборатория RSA предоставляют списки рекомендуемых размеров ключей для безопасного шифрования, и на момент написания этой книги они рекомендуют 80-битовые симметричные ключи, как эквивалентные 1024-битовым асимметричным. Эта разница объясняет, почему алгоритмы шифрования симметричным ключом работают на несколько порядков быстрее алгоритмов шифрования с открытым ключом.

Простые шифры, вроде шифра Цезаря, легко программируются. По этой причине их можно использовать в приложениях, требующих минимального уровня безопасности, например, для того, чтобы скрыть ответы на загадки. Поскольку эти шифры легко поддаются взлому, их никогда не следует использовать в приложениях, требующих серьезного уровня безопасности.

А еще *никогда* не следует пытаться разработать собственную систему шифрования. Безопасность шифров DES и RSA общепризнанна, т. к. они выдержали многолетнее испытание массовым использованием. За это время было предложено много других систем шифрования, которые оказались уязвимыми к взлому. Разработкой алгоритмов шифрования должны заниматься только профессионалы. Если же вам поручили создать систему безопасности, начните с изучения какой-либо признанной программы, например PGP, чтобы разобраться, как в ней решаются вопросы выбора и распространения ключей. Стойкость любой системы безопасности определяется ее самым слабым звеном.

На практике часто возникают следующие вопросы, связанные с шифрованием.

- *Как проверить, что данные не были повреждены случайно?* Часто требуется удостовериться в том, что при передаче данных они не подверглись случайным искажениям, и полученные данные идентичны отправленным. Одно из решений этой задачи — передать полученные данные обратно отправителю для подтверждения идентичности двух текстов. Но этот метод не срабатывает, если при обратной передаче происходит в точности противоположная ошибка. К тому же, при такой схеме пропускная способность канала уменьшается вдвое, что представляет серьезное неудобство.

  Более эффективным методом будет использование контрольной суммы, т. е. хэширование длинного текста с помощью простой математической функции в небольшое число и передача этой контрольной суммы вместе с текстом. Принимающая сторона вычисляет контрольную сумму принятого текста и сравнивает ее с первоначальной. В самой простой схеме с контрольной суммой вычисляется сумма байтов текста по модулю некоторой константы, например, $2^8 = 256$. К сожалению, такая схема не вы-

является ошибку, при которой несколько символов меняются местами, поскольку сложение коммутативно.

Более надежный способ вычисления контрольной суммы включает в себя использование *циклического избыточного кода*. Он применяется в большинстве систем связи и в компьютерах для проверки целостности обмена данных с диском. Для получения этой контрольной суммы вычисляется остаток от деления двух многочленов, причем делитель является функцией входного текста. Разработка этих многочленов является сложной математической задачей, но такая контрольная сумма обеспечивает обнаружение всех достаточно вероятных ошибок. Подробности эффективной реализации весьма сложны, поэтому рекомендуется начинать с уже существующих реализаций.

- *Как проверить, что данные не были повреждены преднамеренно?* Метод контроля с помощью циклического избыточного кода хорошо распознает случайные искажения, но не годится для выявления преднамеренных. Для таких целей лучше подходят криптографические функции хэширования, такие как MD5 или SHA-256, которые легко вычисляются для любого документа, но являются труднообратимыми. Это означает, что по данному хэш-значению x трудно создать такой документ d, для которого $H(d) = x$. Это свойство хэш-функций делает их полезными для использования в качестве цифровых подписей и в других приложениях.
- *Как доказать, что файл не был изменен?* Если я вышлю вам контракт в электронной форме, ничто не помешает вам отредактировать файл и утверждать, что ваша версия отражает нашу договоренность. Мне нужен способ доказать, что модифицированная версия документа является подделкой. *Цифровая подпись* представляет собой криптографический способ подтверждения подлинности документа.

  Я могу вычислить контрольную сумму для данного файла и зашифровать ее с помощью закрытого ключа. Вместе с файлом контракта я высылаю его зашифрованную контрольную сумму. Вы можете отредактировать файл, но, чтобы обмануть суд, вам надо будет отредактировать и зашифрованную контрольную сумму так, чтобы после ее расшифровки получилась исходная. При использовании достаточно хорошей функции вычисления контрольной суммы создание другого файла с такой же контрольной суммой будет невыполнимой задачей. Для полной безопасности мне понадобятся услуги пользующейся всеобщим доверием третьей стороны, которая удостоверит подлинность цифровой подписи и свяжет закрытый ключ со мной.
- *Как ограничить доступ к материалам, защищенным авторским правом?* Важным новым применением шифрования является управление авторскими правами на цифровые аудио- и видеоматериалы. Принципиальным вопросом здесь является скорость декодирования, т. к. она должна соответствовать скорости пересылки данных в реальном времени. Такие *потоковые шифры* обычно включают в себя генерирование потока псевдослучайных битов с использованием, например, генератора кода на регистре сдвига. Операция исключающего ИЛИ над этими битами и потоком данных дает зашифрованную последовательность. Для восстановления исходных данных опять выполняется операция исключающего ИЛИ над зашифрованным потоком данных с тем же самым потоком псевдослучайных битов, который использовался для шифрования.

Было доказано, что высокоскоростные системы потокового шифрования сравнительно легко поддаются взлому. Современное решение этой проблемы включает принятие специальных законов и уголовное преследование за их нарушение.

Реализации. Криптографическая библиотека Nettle содержит, среди прочего, функции хэширования MD5 и SHA-256, а также блочные шифры DES, AES и некоторые другие. В библиотеку включена и реализация системы RSA. Загрузить библиотеку Nettle можно по адресу **http://www.lysator.liu.se/~nisse/nettle/**.

Подробный обзор алгоритмов шифрования, включающий оценку их устойчивости к взлому, доступен на веб-странице **http://www.cryptolounge.Org/wiki/Category: Algorithm**. На странице **http://csrc.nist.gov/groups/ST/toolkit** институт NIST представляет коллекцию стандартов и руководств по разработке алгоритмов шифрования.

Большая библиотека Crypto++ классов языка C++ схем шифрования содержит все рассмотренные в этом разделе системы шифрования. Загрузить библиотеку можно с веб-сайта **http://www.cryptopp.com**.

Многие популярные утилиты с открытым кодом используют профессиональные схемы шифрования и служат хорошими моделями текущей практики в этой области. Программу GnuPG, версию с открытым кодом программы PGP, можно загрузить по адресу **http://www.gnupg.org/**. Программу OpenSSL для аутентификации доступа к компьютерным системам можно загрузить по адресу **http://www.openssl.org/**.

Библиотека Boost CRC Library содержит несколько реализаций алгоритмов вычисления контрольной суммы с помощью циклического избыточного кода. Загрузить библиотеку можно с веб-сайта **http://www.boost.org/libs/crc/**.

Примечания

В книге [MOV96] приведено описание технических деталей всех аспектов криптографии. Электронную версию книги можно найти на веб-сайте **http://www.cacr.math.uwaterloo.ca/hac/**. В книге [Sch96] содержится всестороннее обсуждение разных алгоритмов шифрования, но, пожалуй, книга [FS03] является лучшим введением в эту область. В книге [Kah67] увлекательно изложена история криптографии, начиная с древних времен до 1967 г.

Обсуждение алгоритма RSA (см. [RSA78]) приводится, в частности, в книге [CLRS01]. Обширная информация представлена на домашней странице RSA по адресу **http://www.rsa.com/rsalabs/**.

Главным источником информации о современном состоянии криптографии является Агентство национальной безопасности США (National Security Agency, NSA). История шифра DES хорошо представлена в книге [Sch96]. Особенно спорным было решение NSA ограничить длину ключа 56-ю битами.

Хэш-функция MD5 (см. [Riv92]) используется в программе PGP для вычисления цифровых подписей. Ее описание можно найти, например, в [Sch96] и [Sta06]. Недавно были выявлены серьезные проблемы с безопасностью этой функции (см. [WY05]). Семейство хэш-функций SHA (в частности, функции SHA-256 и SHA-512) производит впечатление более надежного.

Родственные задачи. Разложение на множители и проверка чисел на простоту (см. *раздел 13.8*), сжатие текста (см. *раздел 18.5*).

18.7. Минимизация конечного автомата

Вход. Детерминированный конечный автомат M.

Задача. Создать наименьший детерминированный конечный автомат M', поведение которого идентично поведению автомата M (рис. 18.8).

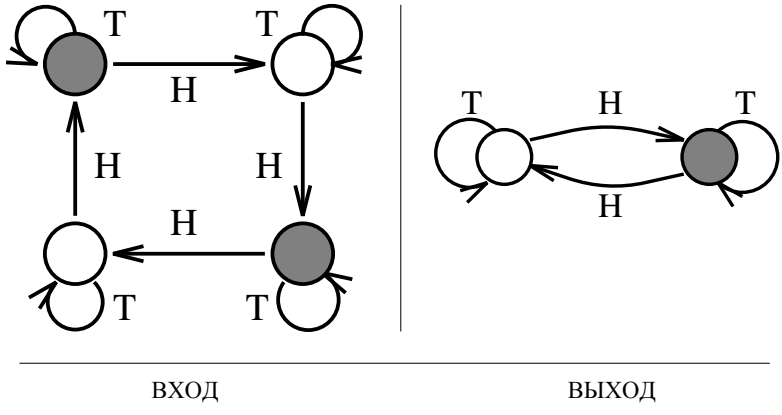


Рис. 18.8. Пример минимизации конечного автомата

Обсуждение. Задача создания и минимизации конечных автоматов часто возникает при разработке программного и аппаратного обеспечения. Конечные автоматы широко используются в приложениях, связанных с распознаванием образов. Современные языки программирования, такие как Java и Python, содержат встроенные средства поддержки регулярных выражений, являющихся самым естественным способом определения автоматов. Конечные автоматы часто применяются в системах управления и компиляторах для кодирования текущего состояния и связанных с ним действий или переходов. Минимизация размера автоматов понижает как затраты по хранению данных, так и стоимость исполнения при использовании таких автоматов.

Конечные автоматы определяются посредством ориентированных графов. Каждая вершина представляет состояние, а каждое помеченное символом алфавита ребро определяет переход из одного состояния в другое при получении данного символа. Показанные на рис. 18.8 автоматы анализируют последовательность бросков монеты. Закрашенные вершины обозначают состояние, при котором "орел" выпал четное количество раз. Такие автоматы можно представить, используя граф (см. *раздел 12.4*) или матрицу переходов размером $n \times |\Sigma|$, где Σ — размер алфавита.

Конечные автоматы часто используются для поиска образцов, представленных в виде регулярных выражений, которые являются результатом выполнения операций И, ИЛИ или повтора над регулярными выражениями меньшего размера. Например, регулярное выражение $a(a + b + c)^*a$ соответствует любой строке алфавита (a, b, c), которая начинается и заканчивается на букву a. Самый лучший способ проверки, описывается ли входная строка данным регулярным выражением R, заключается в создании конечного автомата, эквивалентного R, с последующей эмуляцией этого автомата на данной строке. Альтернативные подходы к решению задачи поиска строк рассматриваются в *разделе 18.3*.

Здесь мы рассмотрим три задачи, связанные с конечными автоматами:

- *минимизация детерминированных конечных автоматов.* У сложных конечных автоматов матрицы переходов имеют недопустимо большой размер, что вызывает необходимость в более "плотных" структурах. Самым простым подходом к решению данной проблемы является устранение повторяющихся состояний автомата. Как показано на рис. 18.8, автоматы самого разного размера могут выполнять одну и ту же функцию.

  Алгоритмы минимизации количества состояний детерминированных конечных автоматов можно найти в любом учебнике по теории автоматов. Основной подход к решению этой задачи — приблизительное разбиение состояний на классы эквивалентности с последующим уточнением этого разбиения. Сначала состояния разделяются на три класса: допускающие, отвергающие и все остальные. Теперь переходы из каждого узла ведут к данному классу по данному символу. Каждый раз, когда два состояния s и t из одного и того же класса C переходят в состояния из разных классов, класс C нужно разбить на два подкласса, один из которых содержит состояние s, а другой — состояние t.

  Этот алгоритм перебирает все классы, проверяя необходимость нового разбиения, и, если такая необходимость имеется, повторяет процесс с начала. Время исполнения этого алгоритма равно $O(n^2)$, т. к. нужно будет выполнить, самое большее, $n - 1$ проходов. Полученные после последнего прохода классы эквивалентности соответствуют состояниям в минимальном конечном автомате. В действительности, существует более эффективный алгоритм с временем исполнения $O(n\log n)$;

- *преобразование недетерминированных автоматов в детерминированные.* С детерминированными конечными автоматами очень легко работать, т. к. в любой данный момент автомат находится в одном-единственном состоянии. *Недетерминированные конечные автоматы* могут находиться в нескольких состояниях одновременно, поэтому их текущее "состояние" представляет подмножество всех возможных состояний автомата.

  Однако любой недетерминированный конечный автомат можно механически преобразовать в эквивалентный детерминированный конечный автомат, который потом удастся минимизировать, как описано выше. Это преобразование может вызвать экспоненциальный рост количества состояний, но не исключено, что оно уменьшится во время минимизации детерминированного конечного автомата. Такой экспоненциальный "взрыв" делает большинство задач минимизации недетерминированных конечных автоматов PSPACE-полными, а это даже хуже, чем NP-полнота.

  Доказательства эквивалентности недетерминированных конечных автоматов, детерминированных конечных автоматов и регулярных выражений достаточно элементарны и рассматриваются в университетских курсах теории автоматов. Впрочем, при кодировании возникают неожиданные проблемы;

- *создание автоматов на основе регулярных выражений.* Для преобразования регулярного выражения в эквивалентный конечный автомат существуют два подхода, разница между которыми состоит в том, создается детерминированный или недетерминированный автомат. Недетерминированные конечные автоматы легче создавать, но их сложнее эмулировать.

Для создания недетерминированных конечных автоматов используются ε-переходы, которые являются дополнительными переходами, не требующими входного символа. Попав в некоторое состояние с помощью ε-перехода, мы должны предполагать, что автомат может быть в любом состоянии. Использование ε-переходов позволяет с легкостью создать автомат на основе обхода в глубину дерева синтаксического разбора регулярного выражения. Данный автомат будет иметь $O(m)$ состояний, где m — длина регулярного выражения. Кроме этого, эмуляция автомата на строке длиной n символов занимает время $O(mn)$, т. к. каждую пару "состояние/префикс" нужно рассмотреть только один раз.

Создание детерминированного конечного автомата начинается с построения дерева синтаксического разбора регулярного выражения с соблюдением того условия, что каждый лист должен представлять символ алфавита в образце. Распознав префикс текста, мы оказываемся в некотором подмножестве возможных позиций, которое будет соответствовать состоянию конечного автомата. Алгоритм Бржозовского (Brzozowski) создает этот автомат по одному состоянию за раз по мере надобности. Но даже в этом случае для некоторых регулярных выражений длиной в m символов потребуется $O(2^m)$ состояний в любом реализующем их детерминированном конечном автомате. В качестве примера можно привести выражение $(a + b)^*a(a + b)(a + b)...(a + b)$. Нет никакой возможности избежать этого экспоненциального "взрыва" сложности по памяти. К счастью, эмуляция входной строки занимает линейное время на любом детерминированном конечном автомате, независимо от размера автомата.

Реализации. Пакет Grail+ на языке C++ предназначен для символьных вычислений над конечными автоматами и регулярными выражениями. Пакет позволяет выполнять преобразования между разными представлениями автомата и минимизировать автоматы. Пакет может обрабатывать автоматы большого размера, определенные на больших алфавитах. Код и документацию можно найти на веб-сайте **http://www.csd.uwo.ca/Research/grail/**, где также предоставлены ссылки на многие другие пакеты. Использование пакета Grail в коммерческих целях требует предварительного разрешения, но для образовательных целей пакет предоставляется бесплатно, как для студентов, так и для преподавателей.

Библиотека FSM компании AT&T представляет собой программный пакет под UNIX, предназначенный для создания, комбинирования, оптимизирования и исследования взвешенных конечных автоматов, как распознавателей, так и преобразователей. Библиотека поддерживает автоматы с более чем десятью миллионами состояний и переходов. Подробности см. на веб-сайте **http://www.research.att.com/~fsmtools/fsm**.

Пакет JFKAP содержит графические инструменты для изучения основных понятий теории автоматов. В пакет включены функции для выполнения преобразований между детерминированными конечными автоматами, недетерминированными конечными автоматами и регулярными выражениями, а также функции для минимизации автоматов. Поддерживаются автоматы более высокого уровня, в том числе контекстно-свободные языки и машины Тьюринга. Пакет JFLAP можно загрузить с веб-сайта **http://www.jflap.org**. Там же доступна книга [RF06].

Пакет FIRE Engine предоставляет коммерческие реализации алгоритмов работы с конечными автоматами и регулярными выражениями. Пакет также содержит реализации

нескольких алгоритмов минимизации конечных автоматов, включая алгоритм Хопкрофта с временем исполнения $O(n\lg n)$. Поддерживаются как детерминированные, так и недетерминированные автоматы. Пакет можно загрузить с веб-сайта **http://www.fastar.org/**, а улучшенную версию — с веб-сайта **http://www.eti.pg.gda.pl/~jandac/minim.html**.

Примечания

В работе [Aho90] предоставлен хороший обзор алгоритмов поиска по образцу, в том числе и для случая, когда образцы представлены регулярными выражениями. Метод поиска образцов в виде регулярных выражений, использующий ε-переходы, был представлен в работе [Tho68]. Описание методов поиска по образцу с помощью конечных автоматов можно найти в книге [AHU74]. Обсуждение конечных автоматов и теории вычислений представлено в книгах [HMU06] и [Sip05].

Основным форумом специалистов в этой области является конференция CIAA (Conference on Implementation and Application of Automata, конференция по реализации и применению автоматов). На веб-сайте **http://tln.li.univ-tours.fr/ciaa/** доступны ссылки на материалы предыдущих конференций и соответствующее программное обеспечение.

Оптимальный алгоритм с временем исполнения $O(n\lg n)$ для минимизации количества состояний детерминированного конечного автомата был представлен в работе [Hop71]. Метод производных для создания конечного автомата из регулярного выражения был представлен в работе [Brz64] и изложен более подробно в работе [BS86]. Среди описаний алгоритма Бржозовского можно назвать книгу [Con71]. Среди последних работ по инкрементальному созданию и оптимизации автоматов можно назвать статью [Wat03]. Задача преобразования детерминированного конечного автомата в минимальный недетерминированный конечный автомат (см. [JR93]), а также задача проверки на эквивалентность двух недетерминированных конечных автоматов (см. [SM73]) являются PSPACE-полными.

Родственные задачи. Задача выполнимости (см. *раздел 14.10*), поиск строк (см. *раздел 18.3*).

18.8. Максимальная общая подстрока

Вход. Множество S строк $S_1, ..., S_n$.

Задача. Найти такую максимальную строку S', все символы которой входят в виде подстроки в каждую строку S_i, где $1 \le i \le n$ (рис. 18.9).

Обсуждение. Задача поиска максимальной общей подстроки или подпоследовательности возникает при сравнении текстов. Особенно важным приложением является выяснение степени сходства биологических последовательностей. Гены белков эволюционируют со временем, но их важнейшие участки изменяться не должны. Максимальная общая подпоследовательность вариаций одного гена в разных биологических видах позволяет получить представление о генном материале, не подвергшемся изменениям.

Задача поиска максимальной общей подстроки является частным случаем задачи вычисления расстояния редактирования (см. *раздел 18.4*), в котором замены символов недопустимы, а разрешаются только вставка и удаление. При этих условиях расстояние редактирования между строками P и T равно $n + m - 2|lcs(P, T)|$, т. к. мы можем удалять

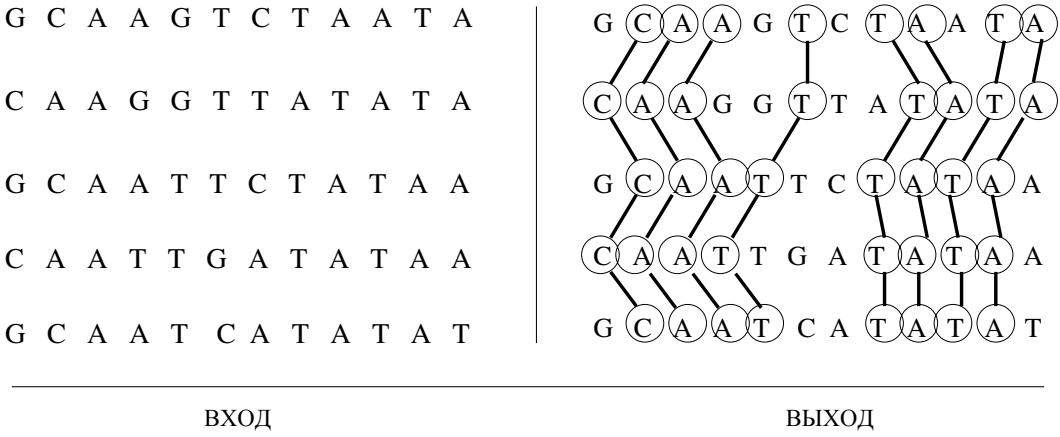


Рис. 18.9. Поиск максимальной общей подстроки

из P символы, отсутствующие в $lcs(P, T)$, и вставлять символы строки T, чтобы преобразовать строку P в строку T.

При решении данной задачи возникают следующие вопросы.

- *Требуется ли найти общую подстроку?* При проверке текстов на уникальность нам нужно найти самую длинную общую фразу в двух и более документах. Так как фразы являются строками символов, ищется максимальная подстрока, общая для всех текстов.

  Максимальную общую подстроку для множества строк можно найти за линейное время, используя суффиксные деревья (см. *раздел 12.3*). Решение заключается в том, чтобы создать суффиксное дерево, содержащее все строки, пометить каждый лист входной строкой, которую он представляет, а потом выполнить обход в глубину и найти самый дальний узел с потомками из каждой входной строки.

- *Требуется ли найти общую подпоследовательность разбросанных символов?* Далее в этом разделе мы будем рассматривать только задачу поиска общих подпоследовательностей разбросанных символов. Алгоритм поиска таких подпоследовательностей является частным случаем алгоритма динамического программирования для вычисления расстояния редактирования. Реализация этого алгоритма на языке C приводится в листинге 8.14.

  Пусть $M[i, j]$ — количество символов самой длинной общей подстроки строк $S[1]$, ..., $S[i]$ и $T[1]$, ..., $T[j]$. Когда $S[i] \neq T[j]$, совпадение последней пары символов невозможно ни при каких обстоятельствах, поэтому $M[i, j] = \max(M[i, j-1], M[i-1, j])$. Но если $S[i] = T[j]$, мы можем выбрать этот символ для нашей подстроки, поэтому $M[i, j] = \max(M[i-1, j-1] + 1, M[i-1, j], M[i, j-1])$.

  Данное рекуррентное соотношение вычисляет длину максимальной общей подпоследовательности за время $O(nm)$. Саму общую подстроку можно воссоздать, двигаясь в обратном направлении от $M[n, m]$ и выясняя, какие символы совпали.

- *Как поступать, если количество наборов совпадающих символов сравнительно невелико?* Для строк, содержащих не слишком много экземпляров одного и того же

символа, существует более быстрый алгоритм. Пусть r — количество пар таких позиций (i, j), для которых $S_i = T_j$. Таким образом, r может достичь значения mn, если обе строки состоят полностью из одного и того же символа, но $r = n$, если обе строки являются перестановками множества $\{1, ..., n\}$. Этот метод рассматривает пары r, как определяющие точки на плоскости.

Полный набор из r таких точек можно вычислить за время $O(n + m + r)$, используя метод раскладывания по корзинам. Для каждого символа алфавита c и каждой строки (S или T) создается корзина, после чего номера позиций каждого символа строки распределяются по соответствующим корзинам. Потом из каждой пары $s \in S_c$ и $t \in T_c$ в корзинах S_c и T_c создается точка (s, t).

Общая подпоследовательность описывает монотонно неубывающий путь по этим точкам, т. е. путь, идущий только вверх и вправо. Самый длинный такой путь можно найти за время $O((n + r)\lg n)$. Точки сортируются в порядке возрастания x-координаты; конфликты с одинаковым значением этой координаты разрешаются в пользу точки с большим значением y-координаты. В этом порядке мы вставляем точки по одной и отслеживаем минимальную конечную y-координату любого пути, проходящего ровно через k точек, для каждого k, где $1 \le k \le n$. Новая точка (p_x, p_y) изменяет только один из этих путей, либо определяя новую максимальную подпоследовательность, либо уменьшая значение y-координаты кратчайшего пути, конечная точка которого находится выше точки p_y.

- *Как поступать, если строки являются перестановками?* Перестановки — это строки, в которых символы не повторяются. Две перестановки задают n пар совпадающих символов, и в этом случае приведенный ранее алгоритм исполняется за время $O(n\lg n)$. Важным частным случаем является поиск максимальной *возрастающей* подпоследовательности числовой последовательности. Отсортировав последовательность, а потом заменив каждое число его номером, мы получим перестановку p. Максимальная общая подпоследовательность перестановки p и последовательности $\{1, 2, 3, ..., n\}$ дает нам максимальную возрастающую подпоследовательность.
- *Как поступать, если задано несколько строк?* Базовый алгоритм динамического программирования можно обобщить для работы с k строками. Время исполнения в этом случае будет равно $O(2^k n^k)$, где n — длина максимальной строки. Алгоритм имеет сложность, экспоненциально зависящую от количества строк k, и поэтому он годится только для небольшого количества строк. Кроме того, данная задача является NP-полной, поэтому не следует надеяться на появление более быстрого точного алгоритма в ближайшее время.

  Для решения задачи выравнивания нескольких последовательностей было предложено большое количество эвристических методов. Часто эти алгоритмы начинают работу с вычисления попарного выравнивания для каждой из $\binom{k}{2}$ пар строк. В одном из подходов две наиболее похожие последовательности объединяются в одну и заменяются ею, и этот процесс повторяется до тех пор, пока все последовательности не будут слиты в одну. Проблема в том, что для двух строк часто существует много разных выравниваний оптимальной стоимости. Выбор самого подходящего из них

зависит от оставшихся последовательностей, подлежащих слиянию, а они алгоритму неизвестны.

Реализации. Для выравнивания нескольких последовательностей ДНК существует много программ. В частности, популярным инструментом выравнивания протеиновых последовательностей является программа ClustalW (см. [THG94]). Программу можно загрузить с веб-сайта **http://www.ebi.ac.uk/Tools/clustalw/**. Еще одним заслуживающим внимания программным средством выравнивания нескольких биологических последовательностей является пакет MSA, который можно загрузить с веб-страницы **http://www.ncbi.nlm.nih.gov/CBBresearch/Schaffer/msa.html**.

Любую из программ динамического программирования для нечеткого сравнения строк, рассмотренных в *разделе 18.4*, можно использовать для поиска максимальной общей подстроки. Специализированные реализации на языках Perl, Java и C доступны на веб-странице **http://bix.ucsd.edu/bioalgorithms/downloads/code/**.

Библиотека Combinatorica (см. [PS03]) содержит реализацию (на языке пакета Mathematica) алгоритма создания максимальной возрастающей подпоследовательности перестановки, что является частным случаем максимальной общей подпоследовательности. Этот алгоритм основан не на динамическом программировании, а на использовании таблиц Янга. Подробности см. в *разделе 19.1.9*.

Примечания

Обзоры алгоритмов поиска максимальных общих подпоследовательностей можно найти в [BHR00] и [GBY91]. Алгоритм для последовательностей, имеющих мало или вовсе не имеющих совпадений, был представлен в работе [HS77]. Его описание приводится в книгах [Aho90] и [Man89]. В последнее время наблюдается неожиданное повышение интереса к этой задаче, в частности к разработке эффективных алгоритмов поиска максимальной общей подпоследовательности, применяющих параллелизм на уровне битов (см. [CIPR01]). В работе [MP80] представлен быстрый алгоритм поиска максимальной общей последовательности для алфавитов фиксированного размера. Алгоритм основан на методе четырех русских и имеет время исполнения, равное $O(mn/\log(\min\{m, n\}))$.

Создадим две случайные строки длиной n на алфавите из a символов. Какова ожидаемая длина максимальной общей подстроки? Эта задача является предметом активных исследований, прекрасный обзор которых представлен в работе [Dan94].

Выравнивание нескольких биологических последовательностей представляет собой отдельную область, хорошим введением в которую послужат книги [Gus97] и [DEKM98]. Свежий обзор состояния дел вы найдете в работе [Not02]. Сложность задачи выравнивания нескольких последовательностей следует из сложности задачи поиска минимальной общей подстроки для больших наборов строк (см. [Mai78]).

Среди приложений задачи поиска максимальной общей подстроки мы упомянули возможность ее применения для выявления плагиата. Интересные рассуждения о том, как реализовать детектор плагиата для компьютерных программ, представлены в работе [SWA03].

Родственные задачи. Нечеткое сравнение строк (см. *раздел 18.4*), поиск минимальной общей надстроки (см. *раздел 18.9*).

18.9. Поиск минимальной общей надстроки

Вход. Множество строк $S = \{S_1, ..., S_m\}$.

Задача. Найти минимальную строку S, содержащую в виде подстроки каждую строку S_i (рис. 18.10).

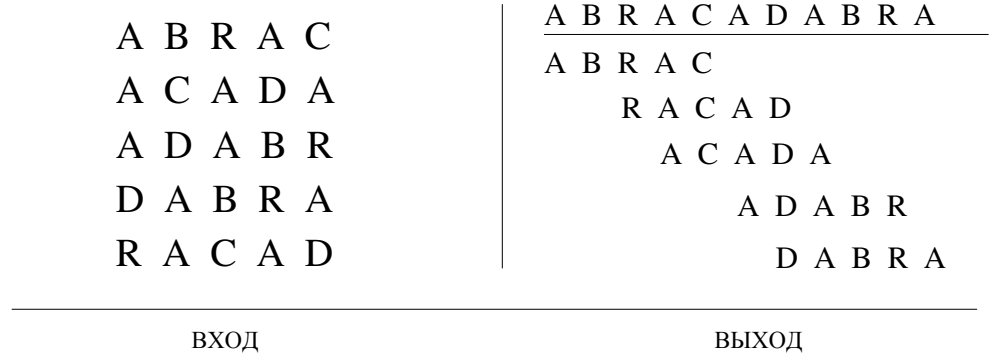


Рис. 18.10. Поиск минимальной общей надстроки

Обсуждение. Задача поиска минимальной общей надстроки возникает в разных приложениях. Однажды заядлый игрок казино спросил у меня, как восстановить последовательность символов на диске игрового автомата. При очередном раунде игры каждый диск игрового автомата после вращения останавливается в случайной позиции, показывая выпавший символ, а также непосредственно предшествующий и следующий ему символы. При наблюдении за достаточно большим количеством раундов игры можно выяснить порядок символов каждого диска игрового автомата, решив задачу поиска минимальной общей (циклической) надстроки для трехсимвольных подстрок, полученных в результате наблюдения.

Другое применение задачи поиска минимальной общей надстроки — сжатие матрицы. Предположим, имеется разреженная матрица M размером $n \times m$, т. е. значение большинства ее элементов равно нулю. Каждую строку матрицы можно разбить на m/k подстроки из k элементов, а потом создать минимальную общую надстроку S' из этих подстрок. Теперь матрицу можно представить этой надстрокой совместно с массивом указателей размером $n \times m/k$, обозначающих начало каждой из этих последовательностей в надстроке. Доступ к любому элементу по-прежнему занимает постоянное время, но если $|S| \ll mn$, то экономия памяти будет значительной.

Возможно, самым популярным приложением задачи поиска минимальной общей надстроки является сборка ДНК. Роботы без труда собирают последовательности из 500 базовых пар ДНК, но наибольший интерес вызывает секвенирование длинных молекул. В процессе сборки последовательностей методом "дробовика" создается множество копий целевой молекулы, которые разбиваются на случайные фрагменты. Затем выполняется секвенирование этих фрагментов, и их минимальная надстрока считается корректной последовательностью.

Найти надстроку набора строк нетрудно, т. к. мы можем просто конкатенировать исходные строки. Проблема в том, чтобы найти *кратчайшую* надстроку. Задача поиска минимальной общей надстроки является NP-полной для строк всех классов.

Задачу поиска минимальной общей надстроки можно с легкостью свести к задаче коммивояжера (см. *раздел 16.4*). Для этого создаем граф перекрытий G, в котором вершина v_i представляет строку S_i. Присваиваем ребру (v_i, v_j) вес, равный длине строки S_i, уменьшенной на длину перекрытия строк S_j и S_i. Например, $w(v_i, v_j) = 1$ для $S_i = abc$ и $S_j = bcd$. Путь с минимальным весом, проходящий через все вершины, определяет минимальную общую надстроку. Веса ребер несимметричны, — обратите внимание, что $w(v_j, v_i) = 3$ на рис. 18.10. К сожалению, несимметричные задачи коммивояжера намного сложнее симметричных.

Стандартным решением является аппроксимирование минимальной общей надстроки с помощью "жадного" алгоритма. Находим пару строк с максимальным перекрытием. Заменяем эти строки их объединением и повторяем процесс до тех пор, пока не останется только одна строка. Для этого эвристического алгоритма можно создать реализацию с линейным временем исполнения. По-видимому, самым дорогим этапом является создание графа перекрытий. Поиск максимального перекрытия двух строк длиной l методом исчерпывающего перебора занимает время $O(l^2)$ для каждой из $O(n^2)$ пар строк. Но для решения этой задачи существуют более быстрые методы, основанные на использовании суффиксных деревьев (см. *раздел 12.3*). Создаем дерево, содержащее все суффиксы всех строк множества S. Строка S_i перекрывает строку S_j тогда и только тогда, когда суффикс строки S_i совпадает с префиксом строки S_j, а это событие отображается вершиной суффиксного дерева. Обход таких вершин в порядке увеличения расстояния от корня определяет соответствующий порядок слияния строк.

Какова эффективность этого "жадного" эвристического алгоритма? Очевидно, существуют случаи, заставляющие его создать надстроку, вдвое превышающую оптимальную. Например, оптимальный порядок слияния строк $c(ab)^k$, $(ba)^k$ и $(ab)^k c$ — слева направо. Но наш "жадный" алгоритм начинает со слияния первой и третьей строки, не оставляя для средней строки возможности перекрытия. Надстрока, возвращаемая "жадным" алгоритмом, никогда не будет превышать оптимальную более, чем в 3,5 раза, а на практике результат обычно даже лучше.

Задача создания надстрок значительно усложняется, когда некоторые исходные строки, согласно условию, не могут быть подстроками конечной надстроки. Задача выявления таких подстрок является NP-полной, если только не разрешено добавить в алфавит дополнительный символ, чтобы использовать его как разделитель.

Реализации. Для сборки последовательностей ДНК существует несколько высокопроизводительных программ. Такие программы исправляют ошибки секвенирования, так что конечный результат не обязательно будет надстрокой входных строк. Как минимум, эти программы могут послужить в качестве моделей, когда вам требуется найти недлинную корректную надстроку.

Самыми последними разработками в этой серии являются программы CAP3 (см. [HM99]) и PCAP (см. [HWA+03]), которые можно загрузить с веб-сайта **http://seq.cs.iastate.edu/**. Эти программы применялись в проектах сборки генов млекопитающих, когда одновременно были задействованы сотни миллионов участков ДНК.

Программа Celera, которая использовалась для первоначального секвенирования генома человека, теперь доступна для общего пользования. Ее можно загрузить с веб-страницы **http://sourceforge.net/projects/wgs-assembler/**.

Примечания

Описание задачи поиска минимальной общей подстроки и ее применения в сборке последовательностей ДНК методом "дробовика" представлено в книге [MKT07] и в работе [Mye99a]. В статье [KM95] приводится алгоритм решения более общего случая задачи поиска минимальной общей надстроки, когда предполагается, что исходные строки содержат ошибки. Эта статья рекомендуется для чтения всем, интересующимся сборкой фрагментов ДНК.

В работе [BJL+94] были представлены первые аппроксимирующие алгоритмы поиска минимальной общей надстроки, в которых применяется разновидность "жадного" эвристического метода и которые выдают решения, имеющие постоянный коэффициент. В результате более поздних исследований (см. [Swe99]) удалось уменьшить этот коэффициент до 2,5, что является шагом вперед на пути к ожидаемому коэффициенту 2. В настоящее время самое лучшее приблизительное решение, выдаваемое стандартным "жадным" эвристическим алгоритмом, имеет коэффициент 3,5 (см. [KS05a]). Эффективные реализации такого алгоритма описаны в работе [Gus94].

Отчет об экспериментах с эвристическими методами поиска минимальных общих надстрок представлен в статье [RBT04]. На основе этих экспериментов можно предположить, что для входных экземпляров разумного размера "жадные" эвристические методы обычно выдают решения, имеющие коэффициент лучший, чем 1,4%. Результаты экспериментов с использованием подходов, основанных на генетических алгоритмах, представлены в статье [ZS04]. В работе [YZ99] содержатся аналитические результаты, демонстрирующие очень небольшое сокращение минимальной общей надстроки случайных последовательностей, обусловленное тем, что ожидаемая длина перекрытия двух произвольных строк невелика.

Родственные задачи. Суффиксные деревья (см. *раздел 12.3*), сжатие текста (см. *раздел 18.5*).

ГЛАВА 19

Ресурсы

Эта глава содержит краткое описание ресурсов, с которыми должен быть знаком каждый разработчик алгоритмов. Хотя часть этой информации была представлена в других местах каталога задач, здесь собраны наиболее важные ссылки.

19.1. Программные системы

В этом разделе дается описание нескольких наиболее полных пакетов с реализациями комбинаторных алгоритмов, любой из которых можно загрузить из Интернета. Хотя эти программы упоминаются в соответствующих разделах каталога задач, они заслуживают особого внимания.

Хороший разработчик алгоритмов не изобретает колесо, а хороший программист не создает код, который уже был создан другими. Лучше всего на эту тему выразился Пикассо: "Хорошие художники заимствуют. Великие художники крадут".

Здесь необходимо сказать несколько слов по поводу воровства. Многие из программных продуктов, описываемых в этой книге, доступны только для исследовательских или образовательных целей. Для использования их в коммерческих целях необходимо заключить лицензионное соглашение с авторами. Я настоятельно рекомендую вам соблюдать это правило. Лицензионные условия большинства академических организаций обычно вполне скромны. Факт использования кода в коммерческих разработках часто бывает важнее для автора, чем финансовая сторона. Автор получает стимул для сопровождения кода и выпуска следующих версий. Поступайте честно и приобретайте необходимые лицензии. Условия лицензии и контактные данные обычно указаны в документации на программное обеспечение или доступны на веб-сайте разработчика.

Хотя многие из описанных здесь программ можно получить из нашего хранилища алгоритмов (**http://www.cs.sunysb.edu/~algorith**), мы настоятельно рекомендуем загружать эти программы с сайтов их разработчиков. Во-первых, существует *очень* большая вероятность, что версия на сайте разработчика является самой свежей. Во-вторых, на сайте разработчика часто можно найти вспомогательные файлы и документацию, которые не были скопированы в наше хранилище, но могут оказаться полезными. Наконец, многие разработчики следят за количеством загрузок своих программ, поэтому вы лишите их морального поощрения, если будете загружать их программы с других сайтов.

19.1.1. Библиотека LEDA

Библиотека LEDA (Library of Efficient Data Types and Algorithms, библиотека эффективных типов данных и алгоритмов) — это, пожалуй, самый лучший из доступных ре-

сурсов с реализациями комбинаторных методов. Библиотека была создана группой разработчиков из института Макса Планка в г. Саарбрюкен, Германия, в составе Курта Мельхорна (Kurt Mehlhorn), Штефана Наера (Stefan Naher), Штефана Ширры (Stefan Schirra), Кристиана Урига (Christian Uhrig) и Кристофа Бурникеля (Christoph Burnikel). Библиотека LEDA уникальна благодаря высочайшей квалификации ее создателей и большому количеству средств, вложенных в этот проект.

Эта библиотека предлагает обширную коллекцию хорошо реализованных на языке C++ структур данных и типов. Особенно полезным является тип `graph`, который поддерживает все основные операции, хотя платой за такую универсальность являются несколько больший объем кода и меньшая скорость работы, чем у специализированных реализаций. В библиотеку входит набор программ для работы с графами, наглядно демонстрирующих, как можно аккуратно и кратко реализовать алгоритмы, используя типы из библиотеки LEDA. В библиотеке имеются хорошие реализации самых важных структур данных, в частности словарей и очередей с приоритетами. Библиотека также содержит алгоритмы и структуры данных для приложений вычислительной геометрии, включая поддержку визуализации. Дополнительную информацию о библиотеке см. в книге [MN99].

С 2001 г. библиотеку можно получить исключительно через компанию Algorithmic Solutions Software GmbH (**http://www.algorithmic-solutions.com/**). Таким образом обеспечивается профессиональная поддержка библиотеки и гарантируется регулярный выпуск новых версий. В феврале 2008 г. была выпущена бесплатная версия библиотеки, содержащая все основные структуры данных (включая словари, очереди с приоритетами, графы и числовые типы). В бесплатной версии отсутствуют исходный код и некоторые особо сложные алгоритмы. Но плата за лицензию на использование полной версии библиотеки невелика, к тому же у пользователя имеется возможность бесплатно загрузить пробную версию полной библиотеки.

19.1.2. Библиотека CGAL

Библиотека CGAL (Computational Geometry Algorithms Library, библиотека алгоритмов вычислительной геометрии) включает в себя эффективные и надежные реализации алгоритмов вычислительной геометрии на языке C++. Библиотека весьма обширна и содержит широкий спектр реализаций алгоритмов создания триангуляционных разбиений, диаграмм Вороного, конфигураций прямых, альфа-очертаний, выпуклых оболочек, а также операций с многоугольниками и многогранниками. Предлагаемые реализации предназначены для работы в двух- и трехмерном пространстве, а некоторые — и в пространствах с бóльшим количеством измерений.

Сайт библиотеки CGAL (**www.cgal.org**) — это первое место, где вы должны искать профессиональные программы для геометрических вычислений, хотя поначалу вам придется потратить какое-то время на то, чтобы разобраться в структуре этой библиотеки. Библиотека CGAL распространяется по схеме двойного лицензирования. Ее можно бесплатно использовать вместе с программным обеспечением с открытым исходным кодом, но для использования в других ситуациях необходимо приобрести коммерческую лицензию.

19.1.3. Библиотека Boost

Библиотека Boost (**www.boost.org**) содержит популярную коллекцию прошедших экспертную оценку бесплатных переносимых исходных кодов библиотек на языке C++. Условия лицензии библиотеки предусматривают как коммерческое, так и некоммерческое использование.

Библиотека Boost Graph Library, пожалуй, лучше всего подходит для читателей этой книги. Руководство пользователя можно найти в [SLL02] и на странице **http://www.boost.org/libs/graph/doc**. Библиотека включает в себя реализации списков смежности, матриц смежности и списков ребер, а также неплохую коллекцию базовых алгоритмов для работы с графами. Интерфейс и компоненты библиотеки являются обобщенными в том смысле, в каком этот термин употребляется в стандартной библиотеке шаблонов STL языка C++. Другие интересные коллекции из библиотеки Boost содержат программы обработки строк и пакеты для математических вычислений.

19.1.4. Библиотека GOBLIN

Библиотека GOBLIN (Graph Object Library for Network Programming Problems, библиотека объектов-графов для задач программирования сетей) содержит классы, написанные на языке C++, и посвящена, в основном, задачам оптимизации графов. Она включает в себя реализации нескольких разновидностей алгоритмов поиска кратчайшего пути, минимальных остовных деревьев и компонент связности, а также большой набор средств для решения задач о потоках в сети и паросочетаниях графов. Для решения таких сложных задач, как поиск независимого множества и вершинная раскраска, предоставляется модуль обобщенный, использующий метод ветвей и границ.

Библиотека GOBLIN была разработана и поддерживается Кристианом Фремут-Пагером (Christian Fremuth-Paeger) из университета г. Аусберга. Библиотека доступна на условиях общественной лицензии GNU ограниченного применения и находится на веб-странице **http://www.math.uni-augsburg.de/~fremuth/goblin.html**. Библиотека имеет интерфейс на языке программирования сценариев Tcl/Tk. Предположительно, библиотека GOBLIN не является такой устойчивой, как библиотека Boost или LEDA, но она содержит ряд алгоритмов, отсутствующих в этих двух библиотеках.

19.1.5. Библиотека Netlib

Библиотека Netlib (**www.netlib.org**) — это хранилище математического программного обеспечения, содержащее большой объем кода, а также множество таблиц и статей. Здесь собраны ресурсы из самых разных источников, дополненные подробным индексом и механизмом поиска. Важными достоинствами библиотеки Netlib являются широкий спектр ее ресурсов и легкость доступа к ним. Если вам потребуется специализированная математическая программа, начните ее поиски с библиотеки Netlib.

Служба GAMS (Guide to Available Mathematical Software, путеводитель по существующему математическому программному обеспечению) индексирует библиотеку Netlib и другие хранилища математического программного обеспечения. Она поддерживается Национальным институтом стандартов и технологий США (NIST) и доступна по адресу **http://gams.nist.gov**.

19.1.6. Коллекция алгоритмов ассоциации ACM

Одним из первых механизмов распространения реализаций полезных алгоритмов была коллекция CALGO (Collected Algorithms of the ACM, сборник алгоритмов ассоциации ACM). Впервые коллекция была представлена в журнале "Communications of the ACM" в 1960 г., и тогда в нее вошли такие знаменитые алгоритмы, как алгоритм Флойда для создания пирамиды с линейным временем исполнения. В настоящее время коллекция сопровождается журналом "ACM Transactions on Mathematical Software". Для каждого алгоритма и его реализации дается краткое описание в статье журнала, а реализация проверяется и добавляется в коллекцию. Программы доступны по адресу **http://www.acm.org/calgo** и на веб-сайте библиотеки Netlib.

На сегодняшний день в коллекции представлено свыше 850 алгоритмов. Большинство программ написано на языке FORTRAN и относится к математическим вычислениям, хотя несколько интересных комбинаторных алгоритмов нашли в ней свое место. Поскольку реализации прошли экспертную оценку, они считаются более надежными, чем аналогичные программы из других источников.

19.1.7. Сайты SourceForge и CPAN

Сайт SourceForge (**http://sourceforge.net**) является самым крупным ресурсом программного обеспечения с открытым исходным кодом, насчитывающим свыше 160 тысяч зарегистрированных проектов. Большинство из этих проектов представляет интерес лишь для узкого круга потребителей, но вы тоже сможете найти много полезного материала, в частности, библиотеки JUNG и JGraphT для работы с графами и средства оптимизации lpsolve и JGAP.

Архив CPAN (Comprehensive Perl Archive Network, всеобъемлющая сеть архивов Perl) (**http://www.cpan.org**) содержит громадную коллекцию модулей и сценариев, написанных на языке Perl. Прежде чем пытаться самостоятельно реализовать что-либо на этом языке, поищите существующую реализацию в этом архиве.

19.1.8. Система Stanford GraphBase

Система Stanford GraphBase интересна во многих отношениях. Прежде всего, она была создана в соответствии с принципом "грамотного программирования", означающим, что ее можно читать. Если чьи-либо программы заслуживают, чтобы их читали, так это программы Дональда Кнута. Его книга [Knu94] содержит полный исходный код системы Stanford GraphBase. В качестве среды программирования используется система CWEB, позволяющая эффективно сочетать описание программы с ее кодом.

Система GraphBase содержит реализации многих важных комбинаторных алгоритмов, включая алгоритмы поиска паросочетаний, вычисления минимальных остовных деревьев и построения диаграмм Вороного, а также специализированные средства для создания графов-расширителей и создания комбинаторных объектов. Наконец, система содержит программы для решения многих развлекательных задач, включая создание "лесенок слов" (цепочек, в которых каждое слово отличается от предыдущего на одну букву) и определения доминирующих отношений среди футбольных команд. Домаш-

няя страница системы находится в Интернете по адресу **http://www-cs-faculty.stanford.edu/~knuth/sgb.html**.

С системой GraphBase интересно экспериментировать, но она мало пригодна для создания приложений общего характера. Ее можно применять, самое большее, как генератор экземпляров графов, которые могут послужить в качестве тестовых данных для других программ. Система содержит графы, полученные путем обработки взаимоотношений персонажей известных романов, статей из словаря синонимов Роджета, визуальных характеристик "Моны Лизы", графов-расширителей, а также модели экономики США. Помимо прочего, в системе GraphBase используются аппаратно-независимые генераторы случайных чисел, и поэтому построенные с ее помощью случайные графы могут быть воссозданы на других компьютерах, что делает их прекрасным материалом для экспериментального сравнения алгоритмов.

19.1.9. Пакет Combinatorica

Пакет Combinatorica (см. [PS03]) содержит реализации свыше 450 алгоритмов по комбинаторике и теории графов. Эти процедуры, написанные на языке пакета Mathematica, предназначены для работы в комплексе, что облегчает эксперименты с ними. Пакет Combinatorica широко используется как для исследовательских, так и для образовательных целей.

Хотя (с моей точки зрения) пакет Combinatorica более полон и лучше интегрирован, чем другие библиотеки комбинаторных алгоритмов, он работает очень медленно. Ответственность за его достоинства и недостатки лежит на среде Mathematica, которая предоставляет полнофункциональный интерпретируемый язык программирования высокого уровня, обладающий вследствие вышесказанного очень низкой эффективностью. Пакет Combinatorica лучше всего подходит для решения небольших задач, а также может послужить источником кратких описаний алгоритмов, пригодных для перевода на другие языки программирования (если вы сумеете разобраться в коде Mathematica).

Пакет Combinatorica и связанные с ним ресурсы доступны на веб-сайте **http://www.combinatorica.com**. Пакет также входит в состав стандартного дистрибутива Mathematica и находится в папке Packages/DiscreteMath/Combinatorica.m.

19.1.10. Программы из книг

Многие книги, посвященные алгоритмам, содержат работоспособные реализации алгоритмов, написанные на распространенных языках программирования. Хотя эти реализации предназначены, главным образом, для иллюстрации материала, их вполне можно использовать на практике. Поскольку они, как правило, аккуратно написаны и имеют небольшой размер, они могут послужить хорошей основой для простых приложений.

Далее приводятся описания наиболее интересных программ этого типа. Большинство из них доступны в хранилище алгоритмов по адресу **http://www.cs.sunysb.edu/~algorith**.

Книга "Programming Challenges"

Если вам нравятся программы на языке C, приведенные в первой части этой книги, вас, вероятно, заинтересуют программы, которые я написал для своей книги [SR03]. Возможно, наибольшую пользу принесут дополнительные примеры алгоритмов динамического программирования, процедуры вычислительной геометрии, такие как создание выпуклой оболочки, а также пакет bignum для выполнения арифметических операций с большими целыми числами. Эта библиотека алгоритмов доступна на сайтах: **http://www.cs.sunysb.edu/~skiena/392/programs/** и **http://www.programming-challenges.com**.

Книга "Combinatorial Algorithms for Computers and Calculators"

Эта книга (см. [NW78]) посвящена алгоритмам создания элементарных комбинаторных объектов, таких как перестановки, подмножества и разбиения. У этих алгоритмов, как правило, очень короткие описания, но их трудно найти, и в них не всегда легко разобраться. Для всех алгоритмов имеются реализации на языке FORTRAN, сопровождаемые обсуждением теоретических основ. Эти программы обычно невелики, что позволяет легко переписать их на каком-либо современном языке программирования. Именно так я и поступил при разработке пакета Combinatorica (см. *раздел 19.1.9*). В книге приводятся алгоритмы как для случайного, так и для последовательного генерирования объектов. Описания недавно появившихся алгоритмов решения некоторых задач (без предоставления кода программ) можно найти в книге [Wil89].

Реализации этих алгоритмов можно загрузить из нашего хранилища алгоритмов. Нам удалось обнаружить их у Нила Слоуна (Neil Sloane), который хранил эти реализации на магнитной ленте, в то время как у самих авторов их не было! В книге [NW78] предложена процедура проверки статистического распределения случайных объектов, создаваемых генераторами, которая позволяет убедиться в равномерности этого распределения. Мы настоятельно рекомендуем выполнять такое тестирование перед использованием этих программ для контроля того, что данные не были потеряны в процессе передачи.

Книга "Computational Geometry in C"

Эта книга (см. [O'R01]) является, пожалуй, самым лучшим введением в вычислительную геометрию, благодаря присутствию в ней аккуратно написанных на языке C реализаций основных алгоритмов. Она содержит реализации всех алгоритмов для решения элементарных задачи вычислительной геометрии, построения выпуклых оболочек, триангуляций и диаграмм Вороного, а также планирования перемещений. Хотя эти реализации, в основном, предназначены для иллюстрации материала, а не коммерческого использования, они, скорее всего, достаточно надежны. Программы можно загрузить с веб-сайта **http://maven.smith.edu/~orourke/code.html**.

Книга "Algorithms in C++"

Эта книга (см. [Sed98] и [SS02]) издана в нескольких вариантах, ориентированных на разные языки программирования, включая языки C, C++ и Java. От других книг ее от-

личает широкий диапазон рассматриваемых тем, в том числе алгоритмов для работы с числовыми, строковыми и геометрическими объектами.

Участки текста, специфичные для конкретного языка программирования, содержат много небольших фрагментов кода, а не законченные программы или процедуры, и относиться к ним следует как к моделям, а не рабочим реализациям. Фрагменты кода на языке C++ можно загрузить с веб-сайта **http://www.cs.princeton.edu/~rs/**.

Книга "Discrete Optimization Algorithms in Pascal"

Эта книга (см. **http://www.cs.princeton.edu/~rs/**) содержит коллекцию из 28 программ для решения задач дискретной оптимизации. Сюда входят программы для решения задач целочисленного и линейного программирования, задач о рюкзаке, о покрытии множества, а также задач коммивояжера, вершинной раскраски, календарного планирования и оптимизации сетей. Программы доступны по адресу **http://www.cs.sunysb.edu/~algorith**.

Этот пакет программ интересен тем, что содержащиеся в нем задачи и алгоритмы имеют непосредственное отношение к такой математической дисциплине, как исследование операций. Алгоритмы подбирались с учетом их ценности для решения практических задач.

19.2. Источники данных

При тестировании алгоритмов часто возникает необходимость использовать в качестве входа нетривиальные данные, позволяющие проверить правильность работы алгоритма или сравнить скорость работы разных алгоритмов. Поиск хороших тестовых данных может оказаться трудной задачей. Вот некоторые источники:

- *библиотека TSPLIB.* Эта широко известная библиотека содержит стандартную тестовую коллекцию сложных экземпляров задачи коммивояжера (см. [Rei91]). Имеющиеся в ней экземпляры задачи представляют собой большие графы, полученные из реальных приложений, таких как проектирование печатных плат и сетей. Библиотека доступна по адресу **http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/**. Относительно старые экземпляры задачи находятся также в библиотеке Netlib;
- *Stanford GraphBase.* Этот пакет программ, написанных Дональдом Кнутом (см. *раздел 19.1.8*), содержит машинно-независимые генераторы разнообразных графов, включая графы, построенные на основе матриц расстояний и произведений искусства и литературы, а также графы, представляющие чисто теоретический интерес;
- *материалы соревнований DIMACS.* Некоторые соревнования DIMACS были посвящены реализациям алгоритмов работы с графами и различными структурами данных, а также алгоритмов решения логических задач. Для этих соревнований были разработаны генераторы входных экземпляров задач каждого типа, причем особое внимание уделялось созданию трудных или репрезентативных тестовых данных. Материалы доступны по адресу **http://dimacs.rutgers.edu/Challenges**.

19.3. Библиографические ресурсы

Интернет предоставляет фантастические возможности людям, интересующимся алгоритмами, как, впрочем, и тем, кого интересуют другие темы. Перечислю ресурсы, которыми я пользуюсь чаще всего:

- *цифровая библиотека ACM.* Эта коллекция библиографических ссылок содержит ссылки на практически все когда-либо опубликованные технические доклады по теории вычислительных систем. Коллекция доступна по адресу **http://portal.acm.org**;
- *Google Scholar.* Этот бесплатный ресурс (**http://scholar.google.com**) ограничивается поиском среди научных работ, что позволяет получить более качественные результаты, чем при универсальном поиске. Особенно полезной является возможность выяснить, в каких работах упоминается данная. Таким образом вы можете обновить свой старый справочный материал и увидеть, что нового произошло в той или иной области, а также определить научную ценность конкретной статьи;
- *Amazon.com.* Этот обширный каталог книг (**www.amazon.com**) чрезвычайно полезен при поиске литературы, посвященной разработке алгоритмов, особенно с тех пор, как многие книги были оцифрованы и внесены в его индекс.

19.4. Профессиональные консалтинговые услуги

Консалтинговая фирма Algorist Technologies (**http://www.algorist.com**) предоставляет своим клиентам краткосрочные экспертные услуги по разработке и реализации алгоритмов. Обычно консультант компании выезжает на объект клиента для интенсивного обсуждения задачи с местным коллективом разработчиков в течение срока от одного до трех дней. Компанией Algorist Technologies накоплен впечатляющий список удачных решений по повышению производительности во многих компаниях в различных сферах деятельности. Мы также предоставляем долгосрочные услуги на контрактной основе.

Для получения дополнительной информации относительно услуг, предоставляемых компанией Algorist Technologies, звоните по телефону 212-222-9891 и пишите по адресу электронной почты **info@algorist.com**.

Algorist Technologies
215 West 92nd St. Suite 1F
New York, NY 10025
http://www.algorist.com

Список литературы

[AAAG95] O. Aichholzer, F. Aurenhammer, D. Alberts, and B. Gartner. A novel type of skeleton for polygons. J. Universal Computer Science, 1:752–761, 1995.

[ABCC07] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. *The Traveling Salesman Problem: A computational study*. Princeton University Press, 2007.

[Abd80] N. N. Abdelmalek. A Fortran subroutine for the $L1$ solution of overdetermined systems of linear equations. *ACM Trans. Math. Softw.*, 6(2):228–230, June 1980.

[ABF05] L. Arge, G. Brodal, and R. Fagerberg. Cache-oblivious data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, p. 34:1–34:27. Chapman and Hall / CRC, 2005.

[AC75] A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.

[AC91] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3:149–156, 1991.

[ACG+03] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, S. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer, 2003.

[ACH+91] E. M. Arkin, L. P. Chew, D. P. Huttenlocher, K. Kedem, and J. S. B. Mitchell. An efficiently computable metric for comparing polygonal shapes. *IEEE Trans. PAMI*, 13(3):209–216, 1991.

[ACI92] D. Alberts, G. Cattaneo, and G. Italiano. An empirical study of dynamic graph algorithms. In *Proc. Seventh ACM-SIAM Symp. Discrete Algorithms (SODA)*, p. 192–201, 1992.

[ACK01a] N. Amenta, S. Choi, and R. Kolluri. The power crust. In *Proc. 6th ACM Symp. on Solid Modeling*, p. 249–260, 2001.

[ACK01b] N. Amenta, S. Choi, and R. Kolluri. The power crust, unions of balls, and the medial axis transform. *Computational Geometry: Theory and Applications*, 19:127–153, 2001.

[ACP+07] H. Ahn, O. Cheong, C. Park, C. Shin, and A. Vigneron. Maximizing the overlap of two planar convex sets under rigid motions. *Computational Geometry: Theory and Applications*, 37:3–15, 2007.

[ADGM04] L. Aleksandrov, H. Djidjev, H. Guo, and A. Maheshwari. Partitioning planar graphs with costs and weights. In *Algorithm Engineering and Experiments: 4th International Workshop, ALENEX 2002*, 2004.

[ADKF70] V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. On economical construction of the transitive closure of a directed graph. *Soviet Mathematics, Doklady*, 11:1209–1210, 1970.

[Adl94] L. M. Adleman. Molecular computations of solutions to combinatorial problems. *Science*, 266:1021–1024, November 11, 1994.

[AE83] D. Avis and H. ElGindy. A combinatorial approach to polygon similarity. *IEEE Trans. Inform. Theory*, IT-2:148–150, 1983.

[AE04] G. Andrews and K. Eriksson. *Integer Partitions.* Cambridge Univ. Press, 2004.

[AF96] D. Avis and K. Fukuda. Reverse search for enumeration. *Disc. Applied Math.*, 65:21–46, 1996.

[AFH02] P. Agarwal, E. Flato, and D. Halperin. Polygon decomposition for efficient construction of Minkowski sums. *Computational Geometry: Theory and Applications*, 21:39–61, 2002.

[AG00] H. Alt and L. Guibas. Discrete geometric shapes: Matching, interpolation, and approximation. In J. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, p. 121–153. Elsevier, 2000.

[Aga04] P. Agarwal. Range searching. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, p. 809–837. CRC Press, 2004.

[AGSS89] A. Aggarwal, L. Guibas, J. Saxe, and P. Shor. A linear-time algorithm for computing the Voronoi diagram of a convex polygon. *Discrete and Computational Geometry*, 4:591–604, 1989.

[AGU72] A. Aho, M. Garey, and J. Ullman. The transitive reduction of a directed graph. *SIAM J. Computing*, 1:131–137, 1972.

[Aho90] A. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Algorithms and Complexity*, vol. A, p. 255–300. MIT Press, 1990.

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading MA, 1974.

[AHU83] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading MA, 1983.

[Aig88] M. Aigner. *Combinatorial Search.* Wiley-Teubner, 1988.

[AITT00] Y. Asahiro, K. Iwama, H. Tamaki, and T. Tokuyama. Greedily finding a dense subgraph. *J. Algorithms*, 34:203–221, 2000.

[AK89] E. Aarts and J. Korst. *Simulated annealing and Boltzman machines: A stochastic approach to combinatorial optimization and neural computing.* John Wiley and Sons, 1989.

[AKD83] J. H. Ahrens, K. D. Kohrt, and U. Dieter. Sampling from gamma and Poisson distributions. *ACM Trans. Math. Softw.*, 9(2):255–257, June 1983.

[AKS04] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*, 160:781–793, 2004.

[AL97] E. Aarts and J. K. Lenstra. *Local Search in Combinatorial Optimization.* John Wiley and Sons, West Sussex, England, 1997.

[AM93] S. Arya and D. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. Fourth ACM-SIAM Symp. Discrete Algorithms (SODA)*, p. 271–280, 1993.

[AMN+98] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM*, 45:891–923, 1998.

[AMO93] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows.* Prentice Hall, Englewood Cliffs NJ, 1993.

[AMWW88] H. Alt, K. Mehlhorn, H. Wagener, and E. Welzl. Congruence, similarity and symmetries of geometric objects. *Discrete Comput. Geom.*, 3:237–256, 1988.

[And98] G. Andrews. *The Theory of Partitions.* Cambridge Univ. Press, 1998.

[And05] A. Andersson. Searching and priority queues in $o(\log n)$ time. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, p. 39:1–39:14. Chapman and Hall / CRC, 2005.

[AP72] A. Aho and T. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM J. Computing*, 1:305–312, 1972.

[APT79] B. Aspvall, M. Plass, and R. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Info. Proc. Letters*, 8:121–123, 1979.

[Aro98] S. Arora. Polynomial time approximations schemes for Euclidean TSP and other geometric problems. *J. ACM*, 45:753–782, 1998.

[AS00] P. Agarwal and M. Sharir. Arrangements. In J. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, p. 49–119. Elsevier, 2000.

[Ata83] M. Atallah. A linear time algorithm for the Hausdorff distance between convex polygons. *Info. Proc. Letters*, 8:207–209, 1983.

[Ata84] M. Atallah. Checking similarity of planar figures. *Internat. J. Comput. Inform. Sci.*, 13:279–290, 1984.

[Ata98] M. Atallah. *Algorithms and Theory of Computation Handbook.* CRC, 1998.

[Aur91] F. Aurenhammer. Voronoi diagrams: a survey of a fundamental data structure. *ACM Computing Surveys*, 23:345–405, 1991.

[Bar03] A. Barabasi. *Linked: How Everything Is Connected to Everything Else and What It Means.* Plume, 2003.

[BBF99] V. Bafna, P. Berman, and T. Fujito. A 2-approximation algorithm for the undirected feedback vertex set problem. *SIAM J. Discrete Math.*, 12:289– 297, 1999.

[BBPP99] I. Bomze, M. Budinich, P. Pardalos, and M. Pelillo. The maximum clique problem. In D.-Z. Du and P.M. Pardalos, editors, *Handbook of Combinatorial Optimization*, vol. A sup., p. 1–74. Kluwer, 1999.

[BCGR92] D. Berque, R. Cecchini, M. Goldberg, and R. Rivenburgh. The SetPlayer system for symbolic computation on power sets. *J. Symbolic Computation*, 14:645–662, 1992.

[BCPB04] J. Boyer, P. Cortese, M. Patrignani, and G. Di Battista. Stop minding your p's and q's: Implementing a fast and simple DFS-based planarity testing and embedding algorithm. In *Proc. Graph Drawing (GD '03)*, vol. 2912 LNCS, p. 25–36, 2004.

[BCW90] T. Bell, J. Cleary, and I. Witten. *Text Compression.* Prentice Hall, Englewood Cliffs NJ, 1990.

[BD99] R. Bubley and M. Dyer. Faster random generation of linear extensions. *Disc. Math.*, 201:81–88, 1999.

[BDH97] C. Barber, D. Dobkin, and H. Huhdanpaa. The Quickhull algorithm for convex hulls. *ACM Trans. on Mathematical Software*, 22:469–483, 1997.

[BDN01] G. Bilardi, P. D'Alberto, and A. Nicolau. Fractal matrix multiplication: a case study on portability of cache performance. In *Workshop on Algorithm Engineering (WAE)*, 2001.

[BDY06] K. Been, E. Daiches, and C. Yap. Dynamic map labeling. *IEEE Trans. Visualization and Computer Graphics*, 12:773–780, 2006.

[Bel58] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[Ben75] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[Ben90] J. Bentley. *More Programming Pearls*. Addison-Wesley, Reading MA, 1990.

[Ben92a] J. Bentley. Fast algorithms for geometric traveling salesman problems. *ORSA J. Computing*, 4:387–411, 1992.

[Ben92b] J. Bentley. Software exploratorium: The trouble with qsort. *UNIX Review*, 10(2):85–93, February 1992.

[Ben99] J. Bentley. *Programming Pearls*. Addison-Wesley, Reading MA, second edition edition, 1999.

[Ber89] C. Berge. *Hypergraphs*. North-Holland, Amsterdam, 1989.

[Ber02] M. Bern. Adaptive mesh generation. In T. Barth and H. Deconinck, editors, *Error Estimation and Adaptive Discretization Methods in Computational Fluid Dynamics*, p. 1–56. Springer-Verlag, 2002.

[Ber04a] M. Bern. Triangulations and mesh generation. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, p. 563–582. CRC Press, 2004.

[Ber04b] D. Bernstein. Fast multiplication and its applications. **http://cr.yp.to/arith.html**, 2004.

[BETT99] G. Di Battista, P. Eades, R. Tamassia, and I. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.

[BF00] M. Bender and M. Farach. The LCA problem revisited. In *Proc. 4th Latin American Symp. on Theoretical Informatics*, p. 88–94. Springer-Verlag LNCS vol. 1776, 2000.

[BFP+72] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection. *J. Computer and System Sciences*, 7:448–461, 1972.

[BFV07] G. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. *ACM J. of Experimental Algorithmics*, 12, 2007.

[BG95] J. Berry and M. Goldberg. Path optimization and near-greedy analysis for graph partitioning: An empirical study. In *Proc. 6th ACM-SIAM Symposium on Discrete Algorithms*, p. 223–232, 1995.

[BGS95] M Bellare, O. Goldreich, and M. Sudan. Free bits, PCPs, and nonapproximability – towards tight results. In *Proc. IEEE 36th Symp. Foundations of Computer Science*, p. 422–431, 1995.

[BH90] F. Buckley and F. Harary. *Distances in Graphs*. Addison-Wesley, Redwood City, Calif., 1990.

[BH01] G. Barequet and S. Har-Peled. Efficiently approximating the minimumv. bounding box of a point set in three dimensions. *J. Algorithms*, 38:91–109, 2001.

[BHR00] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proc. String Processing and Information Retreival (SPIRE)*, p. 39–48, 2000.

[BIK+04] H. Bronnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. Toussaint. Space-efficient planar convex hull algorithms. *Theoretical Computer Science*, 321:25–40, 2004.

[BJL+94] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yanakakis. Linear approximation of shortest superstrings. *J. ACM*, 41:630–647, 1994.

[BJL06] C. Buchheim, M. Junger, and S. Leipert. Drawing rooted trees in linear time. *Software: Practice and Experience*, 36:651–665, 2006.

[BJLM83] J. Bentley, D. Johnson, F. Leighton, and C. McGeoch. An experimental study of bin packing. In *Proc. 21st Allerton Conf. on Communication, Control, and Computing*, p. 51–60, 1983.

[BK04] Y. Boykov and V. Kolmogorov. An experimental comparison of mincut/ max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Analysis and Machine Intelligence (PAMI)*, 26:1124–1137, 2004.

[BKRV00] A. Blum, G. Konjevod, R. Ravi, and S. Vempala. Semi-definite relaxations for minimum bandwidth and other vertex-ordering problems. *Theoretical Computer Science*, 235:25–42, 2000.

[BL76] K. Booth and G. Lueker. Testing for the consecutive ones property, interval graphs, and planarity using PQ-tree algorithms. *J. Computer System Sciences*, 13:335–379, 1976.

[BL77] B. P. Buckles and M. Lybanon. Generation of a vector from the lexicographical index. *ACM Trans. Math. Softw.*, 3(2):180–182, June 1977.

[BLS91] D. Bailey, K. Lee, and H. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *J. Supercomputing*, 4:357–371, 1991.

[Blu67] H. Blum. A transformation for extracting new descriptions of shape. In W. Wathen-Dunn, editor, *Models for the Perception of Speech and Visual Form*, p. 362–380. MIT Press, 1967.

[BLW76] N. L. Biggs, E. K. Lloyd, and R. J. Wilson. *Graph Theory 1736-1936*. Clarendon Press, Oxford, 1976.

[BM53] G. Birkhoff and S. MacLane. *A survey of modern algebra*. Macmillian, New York, 1953.

[BM77] R. Boyer and J. Moore. A fast string-searching algorithm. *Communications of the ACM*, 20:762–772, 1977.

[BM89] J. Boreddy and R. N. Mukherjee. An algorithm to find polygon similarity. *Inform. Process. Lett.*, 33(4):205–206, 1989.

[BM01] E. Bingham and H. Mannila. Random projection in dimensionality reduction: applications to image and text data. In *Proc. ACM Conf. Knowledge Discovery and Data Mining (KDD)*, p. 245–250, 2001.

[BM05] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1:485–509, 2005.

[BO79] J. Bentley and T. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28:643–647, 1979.

[BO83] M. Ben-Or. Lower bounds for algebraic computation trees. In *Proc. Fifteenth ACM Symp. on Theory of Computing*, p. 80–86, 1983.

[Bol01] B. Bollobas. *Random Graphs*. Cambridge Univ. Press, second edition, 2001.

[BP76] E. Balas and M. Padberg. Set partitioning — a survey. *SIAM Review*, 18:710–760, 1976.

[BR80] I. Barrodale and F. D. K. Roberts. Solution of the constrained L_1 linear approximation problem. *ACM Trans. Math. Softw.*, 6(2):231–235, June 1980.

[BR95] A. Binstock and J. Rex. *Practical Algorithms for Programmers*. Addison-Wesley, Reading MA, 1995.

[Bra99] R. Bracewell. *The Fourier Transform and its Applications*. McGraw-Hill, third edition, 1999.

[Bre73] R. Brent. *Algorithms for minimization without derivatives*. Prentice-Hall, Englewood Cliffs NJ, 1973.

[Bre74] R. P. Brent. A Gaussian pseudo-random number generator. *Comm. ACM*, 17(12):704–706, December 1974.

[Bre79] D. Brelaz. New methods to color the vertices of a graph. *Comm. ACM*, 22:251–256, 1979.

[Bri88] E. Brigham. *The Fast Fourier Transform*. Prentice Hall, Englewood Cliffs NJ, facimile edition, 1988.

[Bro95] F. Brooks. *The Mythical Man-Month*. Addison-Wesley, Reading MA, 20th anniversary edition, 1995.

[Bru07] P. Brucker. *Scheduling Algorithms*. Springer-Verlag, fifth edition, 2007.

[Brz64] J. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11:481–494, 1964.

[BS76] J. Bentley and M. Shamos. Divide-and-conquer in higher-dimensional space. In *Proc. Eighth ACM Symp. Theory of Computing*, p. 220–230, 1976.

[BS86] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986.

[BS96] E. Bach and J. Shallit. *Algorithmic Number Theory: Efficient Algorithms*, vol. 1. MIT Press, Cambridge MA, 1996.

[BS97] R. Bradley and S. Skiena. Fabricating arrays of strings. In *Proc. First Int. Conf. Computational Molecular Biology (RECOMB '97)*, p. 57–66, 1997.

[BS07] A. Barvinok and A. Samorodnitsky. Random weighting, asymptotic counting and inverse isoperimetry. *Israel Journal of Mathematics*, 158:159–191, 2007.

[BT92] J. Buchanan and P. Turner. *Numerical methods and analysis*. McGraw-Hill, New York, 1992.

[Buc94] A. G. Buckley. A Fortran 90 code for unconstrained nonlinear minimization. *ACM Trans. Math. Softw.*, 20(3):354–372, September 1994.

[BvG99] S. Baase and A. van Gelder. *Computer Algorithms*. Addison-Wesley, Reading MA, third edition, 1999.

[BW91] G. Brightwell and P. Winkler. Counting linear extensions. *Order*, 3:225–242, 1991.

[BW94] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[BW00] R. Borndorfer and R. Weismantel. Set packing relaxations of some integer programs. *Math. Programming A*, 88:425–450, 2000.

[Can87] J. Canny. *The complexity of robot motion planning*. MIT Press, Cambridge MA, 1987.

[Cas95] G. Cash. A fast computer algorithm for finding the permanent of adjacency matrices. *J. Mathematical Chemistry*, 18:115–119, 1995.

[CB04] C. Cong and D. Bader. The Euler tour technique and parallel rooted spanning tree. In *Int. Conf. Parallel Processing (ICPP)*, p. 448–457, 2004.

[CC92] S. Carlsson and J. Chen. The complexity of heaps. In *Proc. Third ACMSIAM Symp. on Discrete Algorithms*, p. 393–402, 1992.

[CC97] W. Cook and W. Cunningham. *Combinatorial Optimization*. Wiley, 1997.

[CC05] S. Chapra and R. Canale. *Numerical Methods for Engineers*. McGraw-Hill, fifth edition, 2005.

[CCDG82] P. Chinn, J. Chvatolva, A. K. Dewdney, and N. E. Gibbs. The bandwidth problem for graphs and matrices – a survey. *J. Graph Theory*, 6:223–254, 1982.

[CCPS98] W. Cook, W. Cunningham, W. Pulleyblank, and A. Schrijver. *Combinatorial Optimization*. Wiley, 1998.

[CD85] B. Chazelle and D. Dobkin. Optimal convex decompositions. In G. Toussaint, editor, *Computational Geometry*, p. 63–133. North-Holland, Amsterdam, 1985.

[CDL86] B. Chazelle, R. Drysdale, and D. Lee. Computing the largest empty rectangle. *SIAM J. Computing*, 15:300–315, 1986.

[CDT95] G. Carpento, M. Dell'Amico, and P. Toth. CDT: A subroutine for the exact solution of large-scale, asymmetric traveling salesman problems. *ACM Trans. Math. Softw.*, 21(4):410–415, December 1995.

[CE92] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments. *J. ACM*, 39:1–54, 1992.

[CFC94] C. Cheng, B. Feiring, and T. Cheng. The cutting stock problem — a survey. *Int. J. Production Economics*, 36:291–305, 1994.

[CFR06] D. Coppersmith, L. Fleischer, and A. Rudrea. Ordering by weighted number of wins gives a good ranking for weighted tournaments. In *Proc. 17th ACMSIAM Symp. Discrete Algorithms (SODA)*, p. 776–782, 2006.

[CFT99] A. Caprara, M. Fischetti, and P. Toth. A heuristic method for the set covering problem. *Operations Research*, 47:730–743, 1999.

[CFT00] A. Caprara, M. Fischetti, and P. Toth. Algorithms for the set covering problem. *Annals of Operations Research*, 98:353–371, 2000.

[CG94] B. Cherkassky and A. Goldberg. On implementing push-relabel method for the maximum flow problem. Technical Report 94-1523, Department of Computer Science, Stanford University, 1994.

[CGJ96] E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: a survey. In D. Hochbaum, editor, *Approximation algorithms*. PWS Publishing, 1996.

[CGJ98] C.R. Coullard, A.B. Gamble, and P.C. Jones. Matching problems in selective assembly operations. *Annals of Operations Research*, 76:95–107, 1998.

[CGK+97] C. Chekuri, A. Goldberg, D. Karger, M. Levine, and C. Stein. Experimental study of minimum cut algorithms. In *Proc. Symp. on Discrete Algorithms (SODA)*, p. 324–333, 1997.

[CGL85] B. Chazelle, L. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25:76–90, 1985.

[CGM+98] B. Cherkassky, A. Goldberg, P. Martin, J. Setubal, and J. Stolfi. Augment or push: a computational study of bipartite matching and unit-capacity flow algorithms. *J. Experimental Algorithmics*, 3, 1998.

[CGPS76] H. L. Crane Jr., N. F. Gibbs, W. G. Poole Jr., and P. K. Stockmeyer. Matrix bandwidth and profile reduction. *ACM Trans. Math. Softw.*, 2(4):375–377, December 1976.

[CGR99] B. Cherkassky, A. Goldberg, and T. Radzik. Shortest paths algorithms: theory and experimental evaluation. *Math. Prog.*, 10:129–174, 1999.

[CGS99] B. Cherkassky, A. Goldberg, and C. Silverstein. Buckets, heaps, lists, and monotone priority queues. *SIAM J. Computing*, 28:1326–1346, 1999.

[CH06] D. Cook and L. Holder. *Mining Graph Data*. Wiley, 2006.

[Cha91] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete and Computational Geometry*, 6:485–524, 1991.

[Cha00] B. Chazelle. A minimum spanning tree algorithm with inverse-Ackerman type complexity. *J. ACM*, 47:1028–1047, 2000.

[Cha01] T. Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. *J. ACM*, 48:1–12, 2001.

[Che85] L. P. Chew. Planing the shortest path for a disc in $O(n^2 \lg n)$ time. In *Proc. First ACM Symp. Computational Geometry*, p. 214–220, 1985.

[CHL07] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

[Chr76] N. Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. Technical report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh PA, 1976.

[Chu97] F. Chung. *Spectral Graph Theory*. AMS, Providence RI, 1997.

[Chv83] V. Chvatal. *Linear Programming*. Freeman, San Francisco, 1983.

[CIPR01] M Crochemore, C. Iliopolous, Y. Pinzon, and J. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Info. Processing Letters*, 80:279–285, 2001.

[CK94] A. Chetverin and F. Kramer. Oligonucleotide arrays: New concepts and possibilities. *Bio/Technology*, 12:1093–1099, 1994.

[CK07] W. Cheney and D. Kincaid. *Numerical Mathematics and Computing*. Brooks/Cole, Monterey CA, sixth edition, 2007.

[CKSU05] H. Cohn, R. Kleinberg, B. Szegedy, and C. Umans. Group-theoretic algorithms for matrix multiplication. In *Proc. 46th Symp. Foundations of Computer Science*, p. 379–388, 2005.

[CL98] M. Crochemore and T. Lecroq. Text data compression algorithms. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, p. 12.1–12.23. CRC Press Inc., Boca Raton, FL, 1998.

[Cla92] K. L. Clarkson. Safe and effective determinant evaluation. In *Proc. 31st IEEE Symposium on Foundations of Computer Science*, p. 387–395, Pittsburgh, PA, 1992.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge MA, second edition, 2001.

[CM69] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. 24th Nat. Conf. ACM*, p. 157–172, 1969.

[CM96] J. Cheriyan and K. Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15:521–549, 1996.

[CM99] G. Del Corso and G. Manzini. Finding exact solutions to the bandwidth minimization problem. *Computing*, 62:189–203, 1999.

[Coh94] E. Cohen. Estimating the size of the transitive closure in linear time. In *35th Annual Symposium on Foundations of Computer Science*, p. 190– 200. IEEE, 1994.

[Con71] J. H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.

[Coo71] S. Cook. The complexity of theorem proving procedures. In *Proc. Third ACM Symp. Theory of Computing*, p. 151–158, 1971.

[CP90] R. Carraghan and P. Pardalos. An exact algorithm for the maximum clique problem. In *Operations Research Letters*, vol. 9, p. 375–382, 1990.

[CP05] R. Crandall and C. Pomerance. *Prime Numbers: A Computational Perspective.* Springer, second edition, 2005.

[CPW98] B. Chen, C. Potts, and G. Woeginger. A review of machine scheduling: Complexity, algorithms and approximability. In D.-Z. Du and P. Pardalos, editors, *Handbook of Combinatorial Optimization*, vol. 3, p. 21–169. Kluwer, 1998.

[CR76] J. Cohen and M. Roth. On the implementation of Strassen's fast multiplication algorithm. *Acta Informatica*, 6:341–355, 1976.

[CR99] W. Cook and A. Rohe. Computing minimum-weight perfect matchings. *INFORMS Journal on Computing*, 11:138–148, 1999.

[CR01] G. Del Corso and F. Romani. Heuristic spectral techniques for the reduction of bandwidth and work-bound of sparse matrices. *Numerical Algorithms*, 28:117–136, 2001.

[CR03] M. Crochemore and W. Rytter. *Jewels of Stringology*.World Scientific, 2003.

[CS93] J. Conway and N. Sloane. *Sphere packings, lattices, and groups*. Springer-Verlag, New York, 1993.

[CSG05] A. Caprara and J. Salazar-Gonzalez. Laying out sparse graphs with provably minimum bandwidth. *INFORMS J. Computing*, 17:356–373, 2005.

[CT65] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[CT92] Y. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proc. IEEE*, 80:1412–1434, 1992.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Computation*, p. 251–280, 1990.

[Dan63] G. Dantzig. *Linear programming and extensions.* Princeton University Press, Princeton NJ, 1963.

[Dan94] V. Dancik. Expected length of longest common subsequences. PhD. thesis, Univ. of Warwick, 1994.

[DB74] G. Dahlquist and A. Bjorck. *Numerical Methods.* Prentice-Hall, Englewood Cliffs NJ, 1974.

[DB86] G. Davies and S. Bowsher. Algorithms for pattern matching. *Software – Practice and Experience*, 16:575–601, 1986.

[dBDK+98] M. de Berg, O. Devillers, M. Kreveld, O. Schwarzkopf, and M. Teillaud. Computing the maximum overlap of two convex polygons under translations. *Theoretical Computer Science*, 31:613–628, 1998.

[dBvKOS00] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, second edition, 2000.

[DEKM98] R. Durbin, S. Eddy, A. Krough, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 1998.

[Den05] L. Y. Deng. Efficient and portable multiple recursive generators of large order. *ACM Trans. on Modeling and Computer Simulation*, 15:1–13, 2005.

[Dey06] T. Dey. *Curve and Surface Reconstruction: Algorithms with Mathematical Analysis*. Cambridge Univ. Press, 2006.

[DF79] E. Denardo and B. Fox. Shortest-route methods: 1. reaching, pruning, and buckets. *Operations Research*, 27:161–186, 1979.

[DFJ54] G. Dantzig, D. Fulkerson, and S. Johnson. Solution of a large-scale travelingsalesman problem. *Operations Research*, 2:393–410, 1954.

[dFPP90] H. de Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10:41–51, 1990.

[DGH+02] E. Dantsin, A. Goerdt, E. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schoning. A deterministic $(2-2/(k+1))^n$ algorithm for k-SAT based on local search. *Theoretical Computer Science*, 289:69–83, 2002.

[DGKK79] R. Dial, F. Glover, D. Karney, and D. Klingman. A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees. *Networks*, 9:215–248, 1979.

[DH92] D. Du and F. Hwang. A proof of Gilbert and Pollak's conjecture on the Steiner ratio. *Algorithmica*, 7:121–135, 1992.

[DHS00] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley-Interscience, New York, second edition, 2000.

[Die04] M. Dietzfelbinger. *Primality Testing in Polynomial Time: From Randomized Algorithms to "PRIMES Is in P"*. Springer, 2004.

[Dij59] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DJ92] G. Das and D. Joseph. Minimum vertex hulls for polyhedral domains. *Theoret. Comput. Sci.*, 103:107–135, 1992.

[Dji00] H. Djidjev. Computing the girth of a planar graph. In *Proc. 27th Int. Colloquium on Automata, Languages and Programming (ICALP)*, p. 821–831, 2000.

[DJP04] E. Demaine, T. Jones, and M. Patrascu. Interpolation search for nonindependent data. In *Proc. 15th ACM-SIAM Symp. Discrete Algorithms (SODA)*, p. 522–523, 2004.

[DL76] D. Dobkin and R. Lipton. Multidimensional searching problems. *SIAM J. Computing*, 5:181–186, 1976.

[DLR79] D. Dobkin, R. Lipton, and S. Reiss. Linear programming is log-space hard for P. *Info. Processing Letters*, 8:96–97, 1979.

[DM80] D. Dobkin and J. I. Munro. Determining the mode. *Theoretical Computer Science*, 12:255–263, 1980.

[DM97] K. Daniels and V. Milenkovic. Multiple translational containment. part I: an approximation algorithm. *Algorithmica*, 19:148–182, 1997.

[DMBS79] J. Dongarra, C. Moler, J. Bunch, and G. Stewart. *LINPACK User's Guide*. SIAM Publications, Philadelphia, 1979.

[DMR97] K. Daniels, V. Milenkovic, and D. Roth. Finding the largest area axisparallel rectangle in a polygon. *Computational Geometry: Theory and Applications*, 7:125–148, 1997.

[DN07] P. D'Alberto and A. Nicolau. Adaptive Strassen's matrix multiplication. In *Proc. 21st Int. Conf. on Supercomputing*, p. 284–292, 2007.

[DP73] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Canadian Cartographer*, 10(2):112–122, December 1973.

[DPS02] J. Diaz, J. Petit, and M. Serna. A survey of graph layout problems. *ACM Computing Surveys*, 34:313–356, 2002.

[DR90] N. Dershowitz and E. Reingold. Calendrical calculations. *Software – Practice and Experience*, 20:899–928, 1990.

[DR02] N. Dershowitz and E. Reingold. *Calendrical Tabulations: 1900-2200*. Cambridge University Press, New York, 2002.

[DRR+95] S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, K. Sagonas, S. Skiena, T. Swift, and D. S. Warren. Unification factoring for efficient execution of logic programs. In *22nd ACM Symposium on Principles of Programming Languages (POPL '95)*, p. 247–258, 1995.

[DSR00] D. Du, J. Smith, and J. Rubinstein. *Advances in Steiner Trees*. Kluwer, 2000.

[DT04] M. Dorigo and T.Stutzle. *Ant Colony Optimization*. MIT Press, Cambridge MA, 2004.

[dVS82] G. de V. Smit. A comparison of three string matching algorithms. *Software – Practice and Experience*, 12:57–66, 1982.

[dVV03] S. de Vries and R. Vohra. Combinatorial auctions: A survey. *Informs J. Computing*, 15:284–309, 2003.

[DY94] Y. Deng and C. Yang. Waring's problem for pyramidal numbers. *Science in China (Series A)*, 37:377–383, 1994.

[DZ99] D. Dor and U. Zwick. Selecting the median. *SIAM J. Computing*, p. 1722–1758, 1999.

[DZ01] D. Dor and U. Zwick. Median selection requires $(2+\varepsilon)n$ comparisons. *SIAM J. Discrete Math.*, 14:312–325, 2001.

[Ebe88] J. Ebert. Computing Eulerian trails. *Info. Proc. Letters*, 28:93–97, 1988.

[ECW92] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24:441–476, 1992.

[Ede87] H. Edelsbrunner. *Algorithms for Combinatorial Geometry*. Springer-Verlag, Berlin, 1987.

[Ede06] H. Edelsbrunner. *Geometry and Topology for Mesh Generation*. Cambridge Univ. Press, 2006.

[Edm65] J. Edmonds. Paths, trees, and flowers. *Canadian J. Math.*, 17:449–467, 1965.

[Edm71] J. Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1:126–136, 1971.

[EE99] D. Eppstein and J. Erickson. Raising roofs, crashing cycles, and playing pool: applications of a data structure for finding pairwise interactions. *Disc. Comp. Geometry*, 22:569–592, 1999.

[EG60] P. Erd˝os and T. Gallai. Graphs with prescribed degrees of vertices. *Mat. Lapok (Hungarian)*, 11:264–274, 1960.

[EG89] H. Edelsbrunner and L. Guibas. Topologically sweeping an arrangement. *J. Computer and System Sciences*, 38:165–194, 1989.

[EG91] H. Edelsbrunner and L. Guibas. Corrigendum: Topologically sweeping an arrangement. *J. Computer and System Sciences*, 42:249–251, 1991.

[EGIN92] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification: A technique for speeding up dynamic graph algorithms. In *Proc. 33rd IEEE Symp. on Foundations of Computer Science (FOCS)*, p. 60–69, 1992.

[EGS86] H. Edelsbrunner, L. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Computing*, 15:317–340, 1986.

[EJ73] J. Edmonds and E. Johnson. Matching, Euler tours, and the Chinese postman. *Math. Programming*, 5:88–124, 1973.

[EK72] J. Edmonds and R. Karp. Theoretical improvements in the algorithmic efficiency for network flow problems. *J. ACM*, 19:248–264, 1972.

[EKA84] M. I. Edahiro, I. Kokubo, and T. Asano. A new point location algorithm and its practical efficiency – comparison with existing algorithms. *ACM Trans. Graphics*, 3:86–109, 1984.

[EKS83] H. Edelsbrunner, D. Kirkpatrick, and R. Seidel. On the shape of a set of points in the plane. *IEEE Trans. on Information Theory*, IT-29:551–559, 1983.

[EL01] S. Ehmann and M. Lin. Accurate and fast proximity queries between polyhedra using convex surface decomposition. *Comp. Graphics Forum*, 20:500– 510, 2001.

[EM94] H. Edelsbrunner and E. Mucke. Three-dimensional alpha shapes. *ACM Transactions on Graphics*, 13:43–72, 1994.

[ENSS98] G. Even, J. Naor, B. Schieber, and M. Sudan. Approximating minimum feedback sets and multi-cuts in directed graphs. *Algorithmica*, 20:151–174, 1998.

[Epp98] D. Eppstein. Finding the k shortest paths. *SIAM J. Computing*, 28:652–673, 1998.

[ES86] H. Edelsbrunner and R. Seidel. Voronoi diagrams and arrangements. *Discrete and Computational Geometry*, 1:25–44, 1986.

[ESS93] H. Edelsbrunner, R. Seidel, and M. Sharir. On the zone theorem for hyperplane arrangements. *SIAM J. Computing*, 22:418–429, 1993.

[ESV96] F. Evans, S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. In *Proc. IEEE Visualization '96*, p. 319–326, 1996.

[Eul36] L. Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii Academiae Scientiarum Petropolitanae*, 8:128–140, 1736.

[Eve79a] S. Even. *Graph Algorithms*. Computer Science Press, Rockville MD, 1979.

[Eve79b] G. Everstine. A comparison of three resequencing algorithms for the reduction of matrix profile and wave-front. *Int. J. Numerical Methods in Engr.*, 14:837–863, 1979.

[F48] I. Fary. On straight line representation of planar graphs. *Acta. Sci. Math. Szeged*, 11:229–233, 1948.

[Fei98] U. Feige. A threshold of ln n for approximating set cover. *J. ACM*, 45:634– 652, 1998.

[FF62] L. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton NJ, 1962.

[FG95] U. Feige and M. Goemans. Approximating the value of two prover proof systems, with applications to max 2sat and max dicut. In *Proc. 3rd Israel Symp. on Theory of Computing and Systems*, p. 182–189, 1995.

[FH06] E. Fogel and D. Halperin. Exact and efficient construction of Minkowski sums for convex polyhedra with applications. In *Proc. 6th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2006.

[FHW07] E. Fogel, D. Halperin, and C. Weibel. On the exact maximum complexity of minkowski sums of convex polyhedra. In *Proc. 23rd Symp. Computational Geometry*, p. 319–326, 2007.

[FJ05] M. Frigo and S. Johnson. The design and implementation of FFTW3. *Proc. IEEE*, 93:216–231, 2005.

[FJMO93] M. Fredman, D. Johnson, L. McGeoch, and G. Ostheimer. Data structures for traveling salesmen. In *Proc. 4th 7th Symp. Discrete Algorithms (SODA)*, p. 145–154, 1993.

[Fle74] H. Fleischner. The square of every two-connected graph is Hamiltonian. *J. Combinatorial Theory, B*, 16:29–34, 1974.

[Fle80] R. Fletcher. *Practical Methods of Optimization: Unconstrained Optimization*, vol. 1. John Wiley, Chichester, 1980.

[Flo62] R. Floyd. Algorithm 97 (shortest path). *Communications of the ACM*, 7:345, 1962.

[Flo64] R. Floyd. Algorithm 245 (treesort). *Communications of the ACM*, 18:701, 1964.

[FLPR99] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Symp. Foundations of Computer Science*, 1999.

[FM71] M. Fischer and A. Meyer. Boolean matrix multiplication and transitive closure. In *IEEE 12th Symp. on Switching and Automata Theory*, p. 129– 131, 1971.

[FM82] C. Fiduccia and R. Mattheyses. A linear time heuristic for improving network partitions. In *Proc. 19th IEEE Design Automation Conf.*, p. 175– 181, 1982.

[FN04] K. Fredriksson and G. Navarro. Average-optimal single and multiple approximate string matching. *ACM J. of Experimental Algorithmics*, 9, 2004.

[For87] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[For04] S. Fortune. Voronoi diagrams and Delauney triangulations. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, p. 513–528. CRC Press, 2004.

[FPR99] P. Festa, P. Pardalos, and M. Resende. Feedback set problems. In D.-Z. Du and P.M. Pardalos, editors, *Handbook of Combinatorial Optimization*, vol. A. Kluwer, 1999.

[FPR01] P. Festa, P. Pardalos, and M. Resende. Algorithm 815: Fortran subroutines for computing approximate solution to feedback set problems using GRASP. *ACM Transactions on Mathematical Software*, 27:456–464, 2001.

[FR75] R. Floyd and R. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18:165–172, 1975.

[FR94] M. Furer and B. Raghavachari. Approximating the minimum-degree Steiner tree to within one of optimal. *J. Algorithms*, 17:409–423, 1994.

[Fra79] D. Fraser. An optimized mass storage FFT. *ACM Trans. Math. Softw.*, 5(4):500–517, December 1979.

[Fre62] E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–499, 1962.

[Fre76] M. Fredman. How good is the information theory bound in sorting? *Theoretical Computer Science*, 1:355–361, 1976.

[FS03] N. Ferguson and B. Schneier. *Practical Cryptography*. Wiley, 2003.

[FSV01] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism. In *3rd IAPR TC-15 Workshop on Graphbased Representations in Pattern Recognition*, 2001.

[FT87] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34:596–615, 1987.

[FvW93] S. Fortune and C. van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. 9th ACM Symp. Computational Geometry*, p. 163– 172, 1993.

[FW77] S. Fiorini and R. Wilson. *Edge-colourings of graphs*. Research Notes in Mathematics 16, Pitman, London, 1977.

[FW93] M. Fredman and D. Willard. Surpassing the information theoretic bound with fusion trees. *J. Computer and System Sci.*, 47:424–436, 1993.

[FWH04] E. Folgel, R. Wein, and D. Halperin. Code flexibility and program efficiency by genericity: Improving CGAL's arrangements. In *Proc. 12th European Symposium on Algorithms (ESA'04)*, p. 664–676, 2004.

[Gab76] H. Gabow. An efficient implementation of Edmond's algorithm for maximum matching on graphs. *J. ACM*, 23:221–234, 1976.

[Gab77] H. Gabow. Two algorithms for generating weighted spanning trees in order. *SIAM J. Computing*, 6:139–150, 1977.

[Gal86] Z. Galil. Efficient algorithms for finding maximum matchings in graphs. *ACM Computing Surveys*, 18:23–38, 1986.

[Gal90] K. Gallivan. *Parallel Algorithms for Matrix Computations*. SIAM, Philadelphia, 1990.

[Gas03] S. Gass. *Linear Programming: Methods and Applications*. Dover, fifth edition, 2003.

[GBDS80] B. Golden, L. Bodin, T. Doyle, and W. Stewart. Approximate traveling salesman algorithms. *Operations Research*, 28:694–711, 1980.

[GBY91] G. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, Wokingham, England, second edition, 1991.

[Gen04] J. Gentle. *Random Number Generation and Monte Carlo Methods*. Springer, second edition, 2004.

[GGJ77] M. Garey, R. Graham, and D. Johnson. The complexity of computing Steiner minimal trees. *SIAM J. Appl. Math.*, 32:835–859, 1977.

[GGJK78] M. Garey, R. Graham, D. Johnson, and D. Knuth. Complexity results for bandwidth minimization. *SIAM J. Appl. Math.*, 34:477–495, 1978.

[GH85] R. Graham and P. Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7:43–57, 1985.

[GH06] P. Galinier and A. Hertz. A survey of local search methods for graph coloring. *Computers and Operations Research*, 33:2547–2562, 2006.

[GHMS93] L. J. Guibas, J. E. Hershberger, J. S. B. Mitchell, and J. S. Snoeyink. Approximating polygons and subdivisions with minimum link paths. *Internat. J. Comput. Geom. Appl.*, 3(4):383–415, December 1993.

[GHR95] R. Greenlaw, J. Hoover, and W. Ruzzo. *Limits to Parallel Computation: P-completeness theory*. Oxford University Press, New York, 1995.

[GI89] D. Gusfield and R. Irving. *The Stable Marriage Problem: structure and algorithms*. MIT Press, Cambridge MA, 1989.

[GI91] Z. Galil and G. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23:319–344, 1991.

[Gib76] N. E. Gibbs. A hybrid profile reduction algorithm. *ACM Trans. Math. Softw.*, 2(4):378–387, December 1976.

[Gib85] A. Gibbons. *Algorithmic Graph Theory*. Cambridge Univ. Press, 1985.

[GJ77] M. Garey and D. Johnson. The rectilinear Steiner tree problem is NPcomplete. *SIAM J. Appl. Math.*, 32:826–834, 1977.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the theory of NP-completeness*. W. H. Freeman, San Francisco, 1979.

[GJM02] M. Goldwasser, D. Johnson, and C. McGeoch, editors. *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, vol. 59. AMS, Providence RI, 2002.

[GJPT78] M. Garey, D. Johnson, F. Preparata, and R. Tarjan. Triangulating a simple polygon. *Info. Proc. Letters*, 7:175–180, 1978.

[GK95] A. Goldberg and R. Kennedy. An efficient cost scaling algorithm for the assignment problem. *Math. Programming*, 71:153–177, 1995.

[GK98] S. Guha and S. Khuller. Approximation algorithms for connected dominating sets. *Algorithmica*, 20:374–387, 1998.

[GKK74] F. Glover, D. Karney, and D. Klingman. Implementation and computational comparisons of primal-dual computer codes for minimum-cost network flow problems. *Networks*, 4:191–212, 1974.

[GKP89] R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading MA, 1989.

[GKS95] S. Gupta, J. Kececioglu, and A. Schaffer. Improving the practical space and time efficiency of the shortest-paths approach to sum-of-pairs multiple sequence alignment. *J. Computational Biology*, 2:459–472, 1995.

[GKT05] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *Proc. 31st Int. Conf on Very Large Data Bases*, p. 721–732, 2005.

[GKW06] A. Goldberg, H. Kaplan, and R. Werneck. Reach for A*: Efficient point-topoint shortest path algorithms. In *Proc. 8th Workshop on Algorithm Engineering and Experimentation (ALENEX)*, 2006.

[GKW07] A. Goldberg, H. Kaplan, and R.Werneck. Better landmarks within reach. In *Proc. 9th Workshop on Algorithm Engineering and Experimentation (ALENEX)*, p. 38–51, 2007.

[GL96] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins University Press, third edition, 1996.

[Glo90] F. Glover. Tabu search: A tutorial. *Interfaces*, 20 (4):74–94, 1990.

[GM86] G. Gonnet and J. I. Munro. Heaps on heaps. *SIAM J. Computing*, 15:964– 971, 1986.

[GM91] S. Ghosh and D. Mount. An output-sensitive algorithm for computing visibility graphs. *SIAM J. Computing*, 20:888–910, 1991.

[GMPV06] F. Gomes, C. Meneses, P. Pardalos, and G. Viana. Experimental analysis of approximation algorithms for the vertex cover and set covering problems. *Computers and Operations Research*, 33:3520–3534, 2006.

[GO04] J. Goodman and J. O'Rourke, editors. *Handbook of Discrete and Computational Geometry*. CRC Press, second edition, 2004.

[Goe97] M. Goemans. Semidefinite programming in combinatorial optimization. *Mathematical Programming*, 79:143–161, 1997.

[Gol93] L. Goldberg. *Efficient Algorithms for Listing Combinatorial Structures*. Cambridge University Press, 1993.

[Gol97] A. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *J. Algorithms*, 22:1–29, 1997.

[Gol01] A. Goldberg. Shortest path algorithms: Engineering aspects. In *12th International Symposium on Algorithms and Computation*, number 2223 in LNCS, p. 502–513. Springer, 2001.

[Gol04] M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*, vol. 57 of *Annals of Discrete Mathematics*. North Holland, second edition, 2004.

[Gon07] T. Gonzalez. *Handbook of Approximation Algorithms and Metaheuristics*. Chapman-Hall / CRC, 2007.

[GP68] E. Gilbert and H. Pollak. Steiner minimal trees. *SIAM J. Applied Math.*, 16:1–29, 1968.

[GP79] B. Gates and C. Papadimitriou. Bounds for sorting by prefix reversals. *Discrete Mathematics*, 27:47–57, 1979.

[GP07] G. Gutin and A. Punnen. *The Traveling Salesman Problem and Its Variations*. Springer, 2007.

[GPS76] N. Gibbs, W. Poole, and P. Stockmeyer. A comparison of several bandwidth and profile reduction algorithms. *ACM Trans. Math. Software*, 2:322–330, 1976.

[Gra53] F. Gray. Pulse code communication. US Patent 2632058, March 17, 1953.

[Gra72] R. Graham. An efficient algorithm for determining the convex hull of a finite planar point set. *Info. Proc. Letters*, 1:132–133, 1972.

[Gri89] D. Gries. *The Science of Programming*. Springer-Verlag, 1989.

[GS62] D. Gale and L. Shapely. College admissions and the stability of marriages. *American Math. Monthly*, 69:9–14, 1962.

[GS02] R. Giugno and D. Shasha. Graphgrep : A fast and universal method for querying graphs. In *International Conference on Pattern Recognition (ICPR)*, vol. 2, p. 112–115, 2002.

[GT88] A. Goldberg and R. Tarjan. A new approach to the maximum flow problem. *J. ACM*, p. 921–940, 1988.

[GT94] T. Gensen and B. Toft. *Graph Coloring Problems*. Wiley, 1994.

[GT05] M. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*. Wiley, fourth edition, 2005.

[GTV05] M. Goodrich, R. Tamassia, and L. Vismara. Data structures in JDSL. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, p. 43:1–43:22. Chapman and Hall / CRC, 2005.

[Gup66] R. P. Gupta. The chromatic index and the degree of a graph. *Notices of the Amer. Math. Soc.*, 13:719, 1966.

[Gus94] D. Gusfield. Faster implementation of a shortest superstring approximation. *Info. Processing Letters*, 51:271–274, 1994.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[GW95] M. Goemans and D. Williamson. .878-approximation algorithms for MAX CUT and MAX 2SAT. *J. ACM*, 42:1115–1145, 1995.

[GW96] I. Goldberg and D. Wagner. Randomness and the Netscape browser. *Dr. Dobb's Journal*, p. 66–70, 1996.

[GW97] T. Grossman and A. Wool. Computational experience with approximation algorithms for the set covering problem. *European J. Operational Research*, 101, 1997.

[Hai94] E. Haines. Point in polygon strategies. In P. Heckbert, editor, *Graphics Gemes IV*, p. 24–46. Academic Press, 1994.

[Hal04] D. Halperin. Arrangements. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 24, p. 529–562. CRC Press, Boca Raton, FL, 2004.

[Ham87] R. Hamming. *Numerical Methods for Scientists and Engineers*. Dover, second edition, 1987.

[Has82] H. Hastad. Clique is hard to approximate within $n^{1-\varepsilon}$. *Acta Mathematica*, 182:105–142, 182.

[Has97] J. Hastad. Some optimal inapproximability results. In *Proc. 29th ACM Symp. Theory of Comp.*, p. 1–10, 1997.

[HD80] P. Hall and G. Dowling. Approximate string matching. *ACM Computing Surveys*, 12:381–402, 1980.

[HDD03] M. Hilgemeier, N. Drechsler, and R. Drchsler. Fast heuristics for the edge coloring of large graphs. In *Proc. Euromicro Symp. on Digital Systems Design*, p. 230–239, 2003.

[HdlT01] J. Holm, K. de lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48:723–760, 2001.

[Hel01] M. Held. VRONI: An engineering approach to the reliable and efficient computation of Voronoi diagrams of points and line segments. *Computational Geometry: Theory and Applications*, 18:95–123, 2001.

[HFN05] H. Hyyro, K. Fredriksson, and G. Navarro. Increased bit-parallelism for approximate and multiple string matching. *ACM J. of Experimental Algorithmics*, 10, 2005.

[HG97] P. Heckbert and M. Garland. Survey of polygonal surface simplification algorithms. SIGGRAPH 97 Course Notes, 1997.

[HH00] I. Hanniel and D. Halperin. Two-dimensional arrangements in CGAL and adaptive point location for parametric curves. In *Proc. 4th International Workshop on Algorithm Engineering (WAE), LNCS v. 1982*, p. 171–182, 2000.

[HHS98] T. Haynes, S. Hedetniemi, and P. Slater. *Fundamentals of Domination in Graphs*. CRC Press, Boca Raton, 1998.

[Hir75] D. Hirschberg. A linear-space algorithm for computing maximum common subsequences. *Communications of the ACM*, 18:341–343, 1975.

[HK73] J. Hopcroft and R. Karp. An $n^{5/3}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Computing*, 2:225–231, 1973.

[HK90] D. P. Huttenlocher and K. Kedem. Computing the minimum Hausdorff distance for point sets under translation. In *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, p. 340–349, 1990.

[HLD04] W. Hormann, J. Leydold, and G. Derfinger. *Automatic Nonuniform Random Variate Generation*. Springer, 2004.

[HM83] S. Hertel and K. Mehlhorn. Fast triangulation of simple polygons. In *Proc. 4th Internat. Conf. Found. Comput. Theory*, p. 207–218. Lecture Notes in Computer Science, Vol. 158, 1983.

[HM99] X. Huang and A. Madan. Cap3: A DNA sequence assembly program. *Genome Research*, 9:868–877, 1999.

[HMS03] J. Hershberger, M. Maxel, and S. Suri. Finding the k shortest simple paths: A new algorithm and its implementation. In *Proc. 5th Workshop on Algorithm Engineering and Experimentation (ALEN EX)*, 2003.

[HMU06] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, third edition, 2006.

[Hoa61] C. A. R. Hoare. Algorithm 63 (partition) and algorithm 65 (find). *Communications of the ACM*, 4:321–322, 1961.

[Hoa62] C. A. R. Hoare. Quicksort. *Computer Journal*, 5:10–15, 1962.

[Hoc96] D. Hochbaum, editor. *Approximation Algorithms for NP-hard Problems*. PWS Publishing, Boston, 1996.

[Hof82] C. M. Hoffmann. *Group-theoretic algorithms and graph isomorphism*, vol. 136 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, 1982.

[Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[Hol81] I. Holyer. The NP-completeness of edge colorings. *SIAM J. Computing*, 10:718–720, 1981.

[Hol92] J. H. Holland. Genetic algorithms. *Scientific American*, 267(1):66–72, July 1992.

[Hop71] J. Hopcroft. An $n\log n$ algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The theory of machines and computations*, p. 189–196. Academic Press, New York, 1971.

[Hor80] R. N. Horspool. Practical fast searching in strings. *Software – Practice and Experience*, 10:501–506, 1980.

[HP73] F. Harary and E. Palmer. *Graphical enumeration*. Academic Press, New York, 1973.

[HPS+05] M. Holzer, G. Prasinos, F. Schulz, D.Wagner, and C. Zaroliagis. Engineering planar separator algorithms. In *Proc. 13th European Symp. on Algorithms (ESA)*, p. 628–637, 2005.

[HRW92] R. Hwang, D. Richards, and P.Winter. *The Steiner Tree Problem*, vol. 53 of *Annals of Discrete Mathematics*. North Holland, Amsterdam, 1992.

[HS77] J. Hunt and T. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20:350–353, 1977.

[HS94] J. Hershberger and J. Snoeyink. An $O(n \log n)$ implementation of the Douglas-Peucker algorithm for line simplification. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, p. 383–384, 1994.

[HS98] J. Hershberger and J. Snoeyink. Cartographic line simplification and polygon CSG formulae in $O(n\log^* n)$ time. *Computational Geometry: Theory and Applications*, 11:175–185, 1998.

[HS99] J. Hershberger and S. Suri. An optimal algorithm for Euclidean shortest paths in the plane. *SIAM J. Computing*, 28:2215–2256, 1999.

[HSS87] J. Hopcroft, J. Schwartz, and M. Sharir. *Planning, geometry, and complexity of robot motion*. Ablex Publishing, Norwood NJ, 1987.

[HSS07] R. Hardin, N. Sloane, and W. Smith. Maximum volume spherical codes. **http://www.research.att.com/~njas/maxvolumes/**, 2007.

[HSWW05] M. Holzer, F. Schultz, D. Wagner, and T. Willhalm. Combining speed-up techniques for shortest-path computations. *ACM J. of Experimental Algorithmics*, 10, 2005.

[HT73a] J. Hopcroft and R. Tarjan. Dividing a graph into triconnected components. *SIAM J. Computing*, 2:135–158, 1973.

[HT73b] J. Hopcroft and R. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16:372–378, 1973.

[HT74] J. Hopcroft and R. Tarjan. Efficient planarity testing. *J. ACM*, 21:549–568, 1974.

[HT84] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13:338–355, 1984.

[Hub06] M. Huber. Fast perfect sampling from linear extensions. *Disc. Math.*, 306:420–428, 2006.

[Huf52] D. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the IRE*, 40:1098–1101, 1952.

[HUW02] E. Haunschmid, C. Ueberhuber, and P. Wurzinger. Cache oblivious high performance algorithms for matrix multiplication, 2002.

[HW74] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs. In *Proc. Sixth Annual ACM Symposium on Theory of Computing*, p. 172–184, 1974.

[HWA+03] X. Huang, J. Wang, S. Aluru, S. Yang, and L. Hillier. PCAP: A wholegenome assembly program. *Genome Research*, 13:2164–2170, 2003.

[HWK94] T. He, S.Wang, and A. Kaufman.Wavelet-based volume morphing. In *Proc. IEEE Visualization '94*, p. 85–92, 1994.

[IK75] O. Ibarra and C. Kim. Fast approximation algorithms for knapsack and sum of subset problems. *J. ACM*, 22:463–468, 1975.

[IM04] P. Indyk and J. Matousek. Low-distortion embeddings of finite metric spaces. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*. CRC Press, 2004.

[Ind98] P. Indyk. Faster algorithms for string matching problems: matching the convolution bound. In *Proc. 39th Symp. Foundations of Computer Science*, 1998.

[Ind04] P. Indyk. Nearest neighbors in high-dimensional spaces. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, p. 877–892. CRC Press, 2004.

[IR78] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM J. Computing*, 7:413–423, 1978.

[Ita78] A. Itai. Two commodity flow. *J. ACM*, 25:596–611, 1978.

[J92] J. JaJa. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[Jac89] G. Jacobson. Space-efficient static trees and graphs. In *Proc. Symp. Foundations of Computer Science (FOCS)*, p. 549–554, 1989.

[JAMS91] D. Johnson, C. Aragon, C. McGeoch, and D. Schevon. Optimization by simulated annealing: an experimental evaluation; part II, graph coloring and number partitioning. In *Operations Research*, vol. 39, p. 378– 406, 1991.

[Jar73] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Info. Proc. Letters*, 2:18–21, 1973.

[JD88] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Englewood Cliffs NJ, 1988.

[JLR00] S. Janson, T. Luczak, and A. Rucinski. *Random Graphs*. Wiley, 2000.

[JM93] D. Johnson and C. McGeoch, editors. *Network Flows and Matching: First DIMACS Implementation Challenge*, vol. 12. American Mathematics Society, Providence RI, 1993.

[JM03] M. Junger and P. Mutzel. *Graph Drawing Software*. Springer-Verlag, 2003.

[Joh63] S. M. Johnson. Generation of permutations by adjacent transpositions. *Math. Computation*, 17:282–285, 1963.

[Joh74] D. Johnson. Approximation algorithms for combinatorial problems. *J. Computer and System Sciences*, 9:256–278, 1974.

[Joh90] D. S. Johnson. A catalog of complexity classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Algorithms and Complexity*, vol. A, p. 67–162. MIT Press, 1990.

[Jon86] D. W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29:300–311, 1986.

[Jos99] N. Josuttis. *The C++ Standard Library: A tutorial and reference*. Addison- Wesley, 1999.

[JR93] T. Jiang and B. Ravikumar. Minimal NFA problems are hard. *SIAM J. Computing*, 22:1117–1141, 1993.

[JS01] A. Jagota and L. Sanchis. Adaptive, restart, randomized greedy heuristics for maximum clique. *J. Heuristics*, 7:1381–1231, 2001.

[JSV01] M. Jerrum, A. Sinclair, and E. Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with non-negative entries. In *Proc. 33rd ACM Symp. Theory of Computing*, p. 712–721, 2001.

[JT96] D. Johnson and M. Trick. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, vol. 26. AMS, Providence RI, 1996.

[KA03] P. Ko and S. Aluru. Space-efficient linear time construction of suffix arrays,. In *Proc. 14th Symp. on Combinatorial Pattern Matching (CPM)*, p. 200– 210. Springer-Verlag LNCS, 2003.

[Kah67] D. Kahn. *The Code breakers: the story of secret writing*. Macmillan, New York, 1967.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, p. 85–103. Plenum Press, 1972.

[Kar84] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[Kar96] H. Karloff. How good is the Goemans-Williamson MAX CUT algorithm? In *Proc. Twenty-Eighth Annual ACM Symposium on Theory of Computing*, p. 427–434, 1996.

[Kar00] D. Karger. Minimum cuts in near-linear time. *J. ACM*, 47:46–76, 200.

[Kei00] M. Keil. Polygon decomposition. In J.R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, p. 491–518. Elsevier, 2000.

[KGV83] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[Kha79] L. Khachian. A polynomial algorithm in linear programming. *Soviet Math. Dokl.*, 20:191–194, 1979.

[Kir79] D. Kirkpatrick. Efficient computation of continuous skeletons. In *Proc. 20th IEEE Symp. Foundations of Computing*, p. 28–35, 1979.

[Kir83] D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Computing*, 12:28–35, 1983.

[KKT95] D. Karger, P. Klein, and R. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42:321–328, 1995.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, p. 291–307, 1970.

[KM72] V. Klee and G. Minty. How good is the simplex algorithm. In *Inequalities III*, p. 159–172, New York, 1972. Academic Press.

[KM95] J. D. Kececioglu and E. W. Myers. Combinatorial algorithms for DNA sequence assembly. *Algorithmica*, 13(1/2):7–51, January 1995.

[KMP77] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J. Computing*, 6:323–350, 1977.

[KMP+04] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. In *Proc. 12th European Symp. on Algorithms (ESA'04)*, p. 702–713. www.mpiinf. mpg.de/~mehlhorn/ftp/ClassRoomExamples.ps, 2004.

[KMS96] J. Komlos, Y. Ma, and E. Szemeredi. Matching nuts and bolts in $o(n \log n)$ time. In *Proc. 7th Symp. Discrete Algorithms (SODA)*, p. 232–241, 1996.

[KMS97] S. Khanna, M. Muthukrishnan, and S. Skiena. Efficiently partitioning arrays. In *Proc. ICALP '97*, vol. 1256, p. 616–626. Springer-Verlag LNCS, 1997.

[Knu94] D. Knuth. *The Stanford GraphBase: a platform for combinatorial computing*. ACM Press, New York, 1994.

[Knu97a] D. Knuth. *The Art of Computer Programming, V. 1: Fundamental Algorithms*. Addison-Wesley, Reading MA, third edition, 1997.

[Knu97b] D. Knuth. *The Art of Computer Programming, V. 2: Seminumerical Algorithms*. Addison-Wesley, Reading MA, third edition, 1997.

[Knu98] D. Knuth. *The Art of Computer Programming, V. 3: Sorting and Searching*. Addison-Wesley, Reading MA, second edition, 1998.

[Knu05a] D. Knuth. *The Art of Computer Programming, V. 4 Fascicle 2: Generating All Tuples and Permutations*. Addison Wesley, 2005.

[Knu05b] D. Knuth. *The Art of Computer Programming, V. 4 Fascicle 3: Generating All Combinations and Partitions*. Addison Wesley, 2005.

[Knu06] D. Knuth. *The Art of Computer Programming, V. 4 Fascicle 4: Generating All Trees; History of Combinationatorial Generation*. Addison Wesley, 2006.

[KO63] A. Karatsuba and Yu. Ofman. Multiplication of multi-digit numbers on automata. *Sov. Phys. Dokl.*, 7:595–596, 1963.

[Koe05] H. Koehler. A contraction algorithm for finding minimal feedback sets. In *Proc. 28th Australasian Computer Science Conference (ACSC)*, p. 165– 174, 2005.

[KOS91] A. Kaul, M. A. O'Connor, and V. Srinivasan. Computing Minkowski sums of regular polygons. In *Proc. 3rd Canad. Conf. Comput. Geom.*, p. 74–77, 1991.

[KP98] J. Kececioglu and J. Pecqueur. Computing maximum-cardinality matchings in sparse general graphs. In *Proc. 2nd Workshop on Algorithm Engineering*, p. 121–132, 1998.

[KPP04] H. Kellerer, U. Pferschy, and P. Pisinger. *Knapsack Problems*. Springer, 2004.

[KR87] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Research and Development*, 31:249–260, 1987.

[KR91] A. Kanevsky and V. Ramachandran. Improved algorithms for graph fourconnectivity. *J. Comp. Sys. Sci.*, 42:288–306, 1991.

[Kru56] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. of the American Mathematical Society*, 7:48–50, 1956.

[KS74] D.E. Knuth and J.L. Szwarcfiter. A structured program to generate all topological sorting arrangements. *Information Processing Letters*, 2:153– 157, 1974.

[KS85] M. Keil and J. R. Sack. *Computational Geometry: Minimum decomposition of geometric objects*, p. 197–216. North-Holland, 1985.

[KS86] D. Kirkpatrick and R. Siedel. The ultimate planar convex hull algorithm? *SIAM J. Computing*, 15:287–299, 1986.

[KS90] K. Kedem and M. Sharir. An efficient motion planning algorithm for a convex rigid polygonal object in 2-dimensional polygonal space. *Discrete and Computational Geometry*, 5:43–75, 1990.

[KS99] D. Kreher and D. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search*. CRC Press, 1999.

[KS02] M. Keil and J. Snoeyink. On the time bound for convex decomposition of simple polygons. *Int. J. Comput. Geometry Appl.*, 12:181–192, 2002.

[KS05a] H. Kaplan and N. Shafrir. The greedy algorithm for shortest superstrings. *Info. Proc. Letters*, 93:13–17, 2005.

[KS05b] J. Kelner and D. Spielman. A randomized polynomial-time simplex algorithm for linear programming. *Electronic Colloquim on Computational Complexity*, 156:17, 2005.

[KS07] H. Kautz and B. Selman. The state of SAT. *Disc. Applied Math.*, 155:1514– 1524, 2007.

[KSB05] J. Karkkainen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 2005.

[KSBD07] H. Kautz, B. Selman, R. Brachman, and T. Dietterich. *Satisfiability Testing*. Morgan and Claypool, 2007.

[KSPP03] D Kim, J. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. 14th Symp. Combinatorial Pattern Matching (CPM)*, p. 186–199, 2003.

[KST93] J. Kobler, U. Schoning, and J. Turan. *The Graph Isomorphism Problem: its structural complexity*. Birhauser, Boston, 1993.

[KSV97] D. Keyes, A. Sameh, and V. Venkatarishnan. *Parallel Numerical Algorithms*. Springer, 1997.

[KT06] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison Wesley, 2006.

[Kuh75] H. W. Kuhn. Steiner's problem revisited. In G. Dantzig and B. Eaves, editors, *Studies in Optimization*, p. 53–70. Mathematical Association of America, 1975.

[Kur30] K. Kuratowski. Sur le probl`eme des courbes gauches en topologie. *Fund. Math.*, 15:217–283, 1930.

[KW01] M. Kaufmann and D. Wagner. *Drawing Graphs: Methods and Models*. Springer-Verlag, 2001.

[Kwa62] M. Kwan. Graphic programming using odd and even points. *Chinese Math.*, 1:273–277, 1962.

[LA04] J. Leung and J. Anderson, editors. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC/Chapman-Hall, 2004.

[LA06] J. Lien and N. Amato. Approximate convex decomposition of polygons. *Computational Geometry: Theory and Applications*, 35:100–123, 2006.

[Lam92] J.-L. Lambert. Sorting the sums ($xi + yj$) in $o(n2)$ comparisons. *Theoretical Computer Science*, 103:137–141, 1992.

[Lat91] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, 1991.

[Lau98] J. Laumond. *Robot Motion Planning and Control*. Springer-Verlag, Lectures Notes in Control and Information Sciences 229, 1998.

[LaV06] S. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[Law76] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, Fort Worth TX, 1976.

[LD03] R. Laycock and A. Day. Automatically generating roof models from building footprints. In *Proc. 11th Int. Conf. Computer Graphics, Visualization and Computer Vision (WSCG)*, 2003.

[Lec95] T. Lecroq. Experimental results on string matching algorithms. *Software – Practice and Experience*, 25:727–765, 1995.

[Lee82] D. T. Lee. Medial axis transformation of a planar shape. *IEEE Trans. Pattern Analysis and Machine Intelligence*, PAMI-4:363–369, 1982.

[Len87a] T. Lengauer. Efficient algorithms for finding minimum spanning forests of hierarchically defined graphs. *J. Algorithms*, 8, 1987.

[Len87b] H. W. Lenstra. Factoring integers with elliptic curves. *Annals of Mathematics*, 126:649–673, 1987.

[Len89] T. Lengauer. Hierarchical planarity testing algorithms. *J. ACM*, 36(3):474– 509, July 1989.

[Len90] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Wiley, Chichester, England, 1990.

[Lev92] J. L. Leva. A normal random number generator. *ACM Trans. Math. Softw.*, 18(4):454–455, December 1992.

[Lew82] J. G. Lewis. The Gibbs-Poole-Stockmeyer and Gibbs-King algorithms for reordering sparse matrices. *ACM Trans. Math. Softw.*, 8(2):190–194, June 1982.

[LL96] A. LaMarca and R. Ladner. The influence of caches on the performance of heaps. *ACM J. Experimental Algorithmics*, 1, 1996.

[LL99] A. LaMarca and R. Ladner. The influence of caches on the performance of sorting. *J. Algorithms*, 31:66–104, 1999.

[LLK83] J. K. Lenstra, E. L. Lawler, and A. Rinnooy Kan. *Theory of Sequencing and Scheduling*. Wiley, New York, 1983.

[LLKS85] E. Lawler, J. Lenstra, A. Rinnooy Kan, and D. Shmoys. *The Traveling Salesman Problem*. John Wiley, 1985.

[LLS92] L. Lam, S.-W. Lee, and C. Suen. Thinning methodologies – a comprehensive survey. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 14:869–885, 1992.

[LM04] M. Lin and D. Manocha. Collision and proximity queries. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, p. 787–807. CRC Press, 2004.

[LMM02] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: A survey. *European J. Operations Research*, 141:241–252, 2002.

[LMS06] L. Lloyd, A. Mehler, and S. Skiena. Identifying co-referential names across large corpora. In *Combinatorial Pattern Matching (CPM 2006)*, p. 12– 23. Lecture Notes in Computer Science, v. 4009, 2006.

[LP86] L. Lovasz and M. Plummer. *Matching Theory*. North-Holland, Amsterdam, 1986.

[LP02] W. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer, 2002.

[LP07] A. Lodi and A. Punnen. TSP software. In G. Gutin and A. Punnen, editors, *The Traveling Salesman Problem and Its Variations*, p. 737–749. Springer, 2007.

[LPW79] T. Lozano-Perez and M. Wesley. An algorithm for planning collision-free paths among polygonal obstacles. *Comm. ACM*, 22:560–570, 1979.

[LR93] K. Lang and S. Rao. Finding near-optimal cuts: An empirical evaluation. In *Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '93)*, p. 212–221, 1993.

[LS87] V. Lumelski and A. Stepanov. Path planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 3:403–430, 1987.

[LS95] Y.-L. Lin and S. Skiena. Algorithms for square roots of graphs. *SIAM J. Discrete Mathematics*, 8:99–118, 1995.

[LSCK02] P. L'Ecuyer, R. Simard, E. Chen, and W. D. Kelton. An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50:1073–1075, 2002.

[LT79] R. Lipton and R. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36:346–358, 1979.

[LT80] R. Lipton and R. Tarjan. Applications of a planar separator theorem. *SIAM J. Computing*, 9:615–626, 1980.

[Luc91] E. Lucas. *Recreations Mathematiques*. Gauthier-Villares, Paris, 1891.

[Luk80] E. M. Luks. Isomorphism of bounded valence can be tested in polynomial time. In *Proc. of the 21st Annual Symposium on Foundations of Computing*, p. 42–49. IEEE, 1980.

[LV88] G. Landau and U. Vishkin. Fast string matching with k differences. *J. Comput. System Sci.*, 37:63–78, 1988.

[LV97] M. Li and P. Vitanyi. *An introduction to Kolmogorov complexity and its applications*. Springer-Verlag, New York, second edition, 1997.

[LW77] D. T. Lee and C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9:23–29, 1977.

[LW88] T. Lengauer and E. Wanke. Efficient solution of connectivity problems on hierarchically defined graphs. *SIAM J. Computing*, 17:1063–1080, 1988.

[Mah76] S. Maheshwari. Traversal marker placement problems are NP-complete. Technical Report CU-CS-09276, Department of Computer Science, University of Colorado, Boulder, 1976.

[Mai78] D. Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25:322–336, 1978.

[Mak02] R. Mak. *Java Number Cruncher: The Java Programmer's Guide to Numerical Computing*. Prentice Hall, 2002.

[Man89] U. Manber. *Introduction to Algorithms*. Addison-Wesley, Reading MA, 1989.

[Mar83] S. Martello. An enumerative algorithm for finding Hamiltonian circuits in a directed graph. *ACM Trans. Math. Softw.*, 9(1):131–138, March 1983.

[Mat87] D. W. Matula. Determining edge connectivity in $O(nm)$. In *28th Ann. Symp. Foundations of Computer Science*, p. 249–251. IEEE, 1987.

[McC76] E. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23:262–272, 1976.

[McK81] B. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45– 87, 1981.

[McN83] J. M. McNamee. A sparse matrix package – part II: Special cases. *ACM Trans. Math. Softw.*, 9(3):344–345, September 1983.

[MDS01] D. Musser, G. Derge, and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley Professional, second edition, 2001.

[Meg83] N. Megiddo. Linear time algorithm for linear programming in r^3 and related problems. *SIAM J. Computing*, 12:759–776, 1983.

[Men27] K. Menger. Zur allgemeinen Kurventheorie. *Fund. Math.*, 10:96–115, 1927.

[Mey01] S. Meyers. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley Professional, 2001.

[MF00] Z. Michalewicz and D. Fogel. *How to Solve it: Modern Heuristics*. Springer, Berlin, 2000.

[MG92] J. Misra and D. Gries. A constructive proof of Vizing's theorem. *Info. Processing Letters*, 41:131–133, 1992.

[MG06] J. Matousek and B. Gartner. *Understanding and Using Linear Programming*. Springer, 2006.

[MGH81] J. J. More, B. S. Garbow, and K. E. Hillstrom. Fortran subroutines for testing unconstrained optimization software. *ACM Trans. Math. Softw.*, 7(1):136–140, March 1981.

[MH78] R. Merkle and M. Hellman. Hiding and signatures in trapdoor knapsacks. *IEEE Trans. Information Theory*, 24:525–530, 1978.

[Mie58] W. Miehle. Link-minimization in networks. *Operations Research*, 6:232–243. 1958.

[Mil76] G. Miller. Riemann's hypothesis and tests for primality. *J. Computer and System Sciences*, 13:300–317, 1976.

[Mil97] V. Milenkovic.Multiple translational containment. part II: exact algorithms. *Algorithmica*, 19:183–218, 1997.

[Min78] H. Minc. *Permanents*, vol. 6 of *Encyclopedia of Mathematics and its Applications*. Addison-Wesley, Reading MA, 1978.

[Mit99] J. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-mst, and related problems. *SIAM J. Computing*, 28:1298–1309, 1999.

[MKT07] E. Mardis, S. Kim, and H. Tang, editors. *Advances in Genome Sequencing Technology and Algorithms*. Artech House Publishers, 2007.

[MM93] U. Manber and G. Myers. Suffix arrays: A new method for on–line string searches. *SIAM J. Computing*, p. 935–948, 1993.

[MM96] K. Mehlhorn and P. Mutzel. On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. *Algorithmica*, 16:233–242, 1996.

[MMI72] D. Matula, G. Marble, and J. Isaacson. Graph coloring algorithms. In R. C. Read, editor, *Graph Theory and Computing*, p. 109–122. Academic Press, 1972.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *39th Design Automation Conference (DAC)*, 2001.

[MN98] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation*, 8:3–30, 1998.

[MN99] K. Mehlhorn and S. Naher. *LEDA: A platform for combinatorial and geometric computing*. Cambridge University Press, 1999.

[MN07] V. Makinen and G. Navarro. Compressed full text indexes. *ACM Computing Surveys*, 39, 2007.

[MO63] L. E. Moses and R. V. Oakford. *Tables of Random Permutations*. Stanford University Press, Stanford, Calif., 1963.

[Moe90] S. Moen. Drawing dynamic trees. *IEEE Software*, 7-4:21–28, 1990.

[Moo59] E. F. Moore. The shortest path in a maze. In *Proc. International Symp. Switching Theory*, p. 285–292. Harvard University Press, 1959.

[MOS06] K. Mehlhorn, R. Osbild, and M. Sagraloff. Reliable and efficient computational geometry via controlled perturbation. In *Proc. Int. Coll. on Automata, Languages, and Programming (ICALP)*, vol. 4051, p. 299–310. Springer Verlag, Lecture Notes in Computer Science, 2006.

[Mou04] D. Mount. Geometric intersection. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, p. 857–876. CRC Press, 2004.

[MOV96] A. Menezes, P. Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, 1996.

[MP80] W. Masek and M. Paterson. A faster algorithm for computing string edit distances. *J. Computer and System Sciences*, 20:18–31, 1980.

[MPC+06] S. Mueller, D. Papamichial, J.R. Coleman, S. Skiena, and E. Wimmer. Reduction of the rate of poliovirus protein synthesis through large scale codon deoptimization causes virus attenuation of viral virulence by lowering specific infectivity. *J. of Virology*, 80:9687–96, 2006.

[MPT99] S. Martello, D. Pisinger, and P. Toth. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science*, 45:414–424, 1999.

[MPT00] S. Martello, D. Pisinger, and P. Toth. New trends in exact algorithms for the 0-1 knapsack problem. *European Journal of Operational Research*, 123:325–332, 2000.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, 1995.

[MR01] W. Myrvold and F. Ruskey. Ranking and unranking permutations in linear time. *Info. Processing Letters*, 79:281–284, 2001.

[MR06] W. Mulzer and G. Rote. Minimum weight triangulation is NP-hard. In *Proc. 22nd ACM Symp. on Computational Geometry*, p. 1–10, 2006.

[MRRT53] N. Metropolis, A.W. Rosenbluth, M. N. Rosenbluth, and A. H. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6):1087–1092, June 1953.

[MS91] B. Moret and H. Shapiro. *Algorithm from P to NP: Design and Efficiency*. Benjamin/Cummings, Redwood City, CA, 1991.

[MS93] M. Murphy and S. Skiena. Ranger: A tool for nearest neighbor search in high dimensions. In *Proc. Ninth ACM Symposium on Computational Geometry*, p. 403–404, 1993.

[MS95a] D. Margaritis and S. Skiena. Reconstructing strings from substrings in rounds. Proc. 36th IEEE Symp. Foundations of Computer Science (FOCS), 1995.

[MS95b] J. S. B. Mitchell and S. Suri. Separation and approximation of polyhedral objects. *Comput. Geom. Theory Appl.*, 5:95–114, 1995.

[MS00] M. Mascagni and A. Srinivasan. Algorithm 806: Sprng: A scalable library for pseudorandom number generation. *ACM Trans. Mathematical Software*, 26:436–461, 2000.

[MS05] D. Mehta and S. Sahni. *Handbook of Data Structures and Applications*. Chapman and Hall / CRC, Boca Raton, FL, 2005.

[MT85] S. Martello and P. Toth. A program for the *0-1* multiple knapsack problem. *ACM Trans. Math. Softw.*, 11(2):135–140, June 1985.

[MT87] S. Martello and P. Toth. Algorithms for knapsack problems. In S. Martello, editor, *Surveys in Combinatorial Optimization*, vol. 31 of *Annals of Discrete Mathematics*, p. 213–258. North-Holland, 1987.

[MT90a] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. Wiley, New York, 1990.

[MT90b] K. Mehlhorn and A. Tsakalidis. Data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Algorithms and Complexity*, vol. A, p. 301–341. MIT Press, 1990.

[MU05] M. Mitzenmacher and E. Upfal. *robability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[Mul94] K. Mulmuley. *Computational Geometry: an introduction through randomized algorithms*. Prentice-Hall, New York, 1994.

[Mut05] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Now Publishers, 2005.

[MV80] S. Micali and V. Vazirani. An $o(\sqrt{|V|}\,|e|)$ algorithm for finding maximum matchings in general graphs. In *Proc. 21st. Symp. Foundations of Computing*, p. 17–27, 1980.

[MV99] B. McCullough and H. Vinod. The numerical reliability of econometical software. *J. Economic Literature*, 37:633–665, 1999.

[MY07] K. Mehlhorn and C. Yap. *Robust Geometric Computation*. manuscript, **http://cs.nyu.edu/yap/book/egc/**, 2007.

[Mye86] E. Myers. An $O(nd)$ difference algorithm and its variations. *Algorithmica*, 1:514–534, 1986.

[Mye99a] E. Myers. Whole-genome DNA sequencing. *IEEE Computational Engineering and Science*, 3:33–43, 1999.

[Mye99b] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *J. ACM*, 46:395–415, 1999.

[Nav01a] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33:31–88, 2001.

[Nav01b] G. Navarro. Nr-grep: a fast and flexible pattern matching tool. *Software Practice and Experience*, 31:1265–1312, 2001.

[Nel96] M. Nelson. Fast searching with suffix trees. *Dr. Dobbs Journal*, August 1996.

[Neu63] J. Von Neumann. Various techniques used in connection with random digits. In A. H. Traub, editor, *John von Neumann, Collected Works*, vol. 5. Macmillan, 1963.

[NI92] H. Nagamouchi and T. Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM J. Disc. Math*, 5:54–55, 1992.

[NMB05] W. Nooy, A. Mrvar, and V. Batagelj. *Exploratory Social Network Analysis with Pajek*. Cambridge University Press, 2005.

[NOI94] H. Nagamouchi, T. Ono, and T. Ibaraki. Implementing an efficient minimum capacity cut algorithm. *Math. Prog.*, 67:297–324, 1994.

[Not02] C. Notredame. Recent progress in multiple sequence alignment: a survey. *Pharmacogenomics*, 3:131–144, 2002.

[NR00] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM J. of Experimental Algorithmics*, 5, 2000.

[NR04] T. Nishizeki and S. Rahman. *Planar Graph Drawing*. World Scientific, 2004.

[NR07] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, 2007.

[NS07] G. Narasimhan and M. Smid. *Geometric Spanner Networks*. Cambridge Univ. Press, 2007.

[Nuu95] E. Nuutila. Efficient transitive closure computation in large digraphs. **http://www.cs.hut.fi/~enu/thesis.html**, 1995.

[NW78] A. Nijenhuis and H. Wilf. *Combinatorial Algorithms for Computers and Calculators*. Academic Press, Orlando FL, second edition, 1978.

[NZ80] I. Niven and H. Zuckerman. *An Introduction to the Theory of Numbers*. Wiley, New York, fourth edition, 1980.

[NZ02] S. Naher and O. Zlotowski. Design and implementation of efficient data types for static graphs. In *European Symposium on Algorithms (ESA)*, p. 748–759, 2002.

[OBSC00] A. Okabe, B. Boots, K. Sugihara, and S. Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. Wiley, 2000.

[Ogn93] R. Ogniewicz. *Discrete Voronoi Skeletons*. Hartung-Gorre Verlag, Konstanz, Germany, 1993.

[O'R85] J. O'Rourke. Finding minimal enclosing boxes. *Int. J. Computer and Information Sciences*, 14:183–199, 1985.

[O'R87] J. O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, Oxford, 1987.

[O'R01] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, New York, second edition, 2001.

[Ort88] J. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems*. Plenum, New York, 1988.

[OS04] J. O'Rourke and S. Suri. Polygons. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, p. 583–606. CRC Press, 2004.

[OvL81] M. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Computer and System Sciences*, 23:166–204, 1981.

[OW85] J. O'Rourke and R. Washington. Curve similarity via signatures. In G. T. Toussaint, editor, *Computational Geometry*, p. 295–317. North-Holland, Amsterdam, Netherlands, 1985.

[P57] G. Polya. *How to Solve It*. Princeton University Press, Princeton NJ, second edition, 1957.

[Pan06] R. Panigrahy. *Hashing, Searching, Sketching*. PhD thesis, Stanford University, 2006.

[Pap76a] C. Papadimitriou. The complexity of edge traversing. *J. ACM*, 23:544–554, 1976.

[Pap76b] C. Papadimitriou. The NP-completeness of the bandwidth minimization problem. *Computing*, 16:263–270, 1976.

[Par90] G. Parker. A better phonetic search. *C Gazette*, 5-4, June/July 1990.

[Pas97] V. Paschos. A survey of approximately optimal solutions to some covering and packing problems. *Computing Surveys*, 171-209:171–209, 1997.

[Pas03] V. Paschos. Polynomial approximation and graph-coloring. *Computing*, 70:41–86, 2003.

[Pav82] T. Pavlidis. *Algorithms for Graphics and Image Processing*. Computer Science Press, Rockville MD, 1982.

[Pec04] M. Peczarski. New results in minimum-comparison sorting. *Algorithmica*, 40:133–145, 2004.

[Pec07] M. Peczarski. The Ford-Johnson algorithm still unbeaten for less than 47 elements. *Info. Processing Letters*, 101:126–128, 2007.

[Pet03] J. Petit. Experiments on the minimum linear arrangement problem. *ACM J. of Experimental Algorithmics*, 8, 2003.

[PFTV07] W. Press, B. Flannery, S. Teukolsky, and W. T. Vetterling. *Numerical Recipes: the art of scientific computing*. Cambridge University Press, third edition, 2007.

[PH80] M. Padberg and S. Hong. On the symmetric traveling salesman problem: a computational study. *Math. Programming Studies*, 12:78–107, 1980.

[PIA78] Y. Perl, A. Itai, and H. Avni. Interpolation search – a log log *n* search. *Comm. ACM*, 21:550–554, 1978.

[Pin02] M. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Prentice Hall, second edition, 2002.

[PL94] P. A. Pevzner and R. J. Lipshutz. Towards DNA sequencing chips. In *19th Int. Conf. Mathematical Foundations of Computer Science*, vol. 841, p. 143–158, Lecture Notes in Computer Science, 1994.

[PLM06] F. Panneton, P. L'Ecuyer, and M. Matsumoto. Improved long-period generators based on linear recurrences modulo 2. *ACM Trans. Mathematical Software*, 32:1–16, 2006.

[PM88] S. Park and K. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31:1192–1201, 1988.

[PN04] Shortest Paths and Networks. J. Mitchell. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, p. 607–641. CRC Press, 2004.

[Pom84] C. Pomerance. The quadratic sieve factoring algorithm. In T. Beth, N. Cot, and I. Ingemarrson, editors, *Advances in Cryptology*, vol. 209, p. 169–182. Lecture Notes in Computer Science, Springer-Verlag, 1984.

[PP06] M. Penner and V. Prasanna. Cache-friendly implementations of transitive closure. *ACM J. of Experimental Algorithmics*, 11, 2006.

[PR86] G. Pruesse and F. Ruskey. Generating linear extensions fast. *SIAM J. Computing*, 23:1994, 373-386.

[PR02] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *J. ACM*, 49:16–34, 2002.

[Pra75] V. Pratt. Every prime has a succinct certificate. *SIAM J. Computing*, 4:214– 220, 1975.

[Pri57] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[Pru18] H. Prufer. Neuer Beweis eines Satzes uber Permutationen. *Arch. Math. Phys.*, 27:742–744, 1918.

[PS85] F. Preparata and M. Shamos. *Computational Geometry*. Springer-Verlag, New York, 1985.

[PS98] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, 1998.

[PS02] H. Promel and A. Steger. *The Steiner Tree Problem: a tour through graphs, algorithms, and complexity*. Friedrick Vieweg and Son, 2002.

[PS03] S. Pemmaraju and S. Skiena. *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Cambridge University Press, New York, 2003.

[PSL90] A. Pothen, H. Simon, and K. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Analysis*, 11:430–452, 1990.

[PSS07] F. Putze, P. Sanders, and J. Singler. Cache-, hash-, and space-efficient bloom filters. In *Proc. 6th Workshop on Experimental Algorithms (WEA), LNCS 4525*, p. 108–121, 2007.

[PST07] S. Puglisi, W. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39, 2007.

[PSW92] T. Pavlides, J. Swartz, and Y. Wang. Information encoding with twodimensional bar-codes. *IEEE Computer*, 25:18–28, 1992.

[PT05] A. Pothen and S. Toledo. Cache-oblivious data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, p. 59:1–59:29. Chapman and Hall / CRC, 2005.

[Pug86] G. Allen Pugh. Partitioning for selective assembly. *Computers and Industrial Engineering*, 11:175–179, 1986.

[PV96] M. Pocchiola and G. Vegter. Topologically sweeping visibility complexes via pseudo-triangulations. *Discrete and Computational Geometry*, 16:419–543, 1996.

[Rab80] M. Rabin. Probabilistic algorithm for testing primality. *J. Number Theory*, 12:128–138, 1980.

[Rab95] F. M. Rabinowitz. A stochastic algorithm for global optimization with constraints. *ACM Trans. Math. Softw.*, 21(2):194–213, June 1995.

[Ram05] R. Raman. Data structures for sets. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, p. 33:1–33:22. Chapman and Hall / CRC, 2005.

[Raw92] G. Rawlins. *Compared to What?* Computer Science Press, New York, 1992.

[RBT04] H. Romero, C. Brizuela, and A. Tchernykh. An experimental comparison of approximation algorithms for the shortest common superstring problem. In *Proc. Fifth Mexican Int. Conf. in Computer Science (ENC'04)*, p. 27–34, 2004.

[RC55] Rand-Corporation. *A million random digits with 100,000 normal deviates*. The Free Press, Glencoe, IL, 1955.

[RD01] E. Reingold and N. Dershowitz. *Calendrical Calculations: The Millennium Edition*. Cambridge University Press, New York, 2001.

[RDC93] E. Reingold, N. Dershowitz, and S. Clamen. Calendrical calculations II: Three historical calendars. *Software – Practice and Experience*, 22:383–404, 1993.

[Rei72] E. Reingold. On the optimality of some set algorithms. *J. ACM*, 19:649–659, 1972.

[Rei91] G. Reinelt. TSPLIB – a traveling salesman problem library. *ORSA J. Computing*, 3:376–384, 1991.

[Rei94] G. Reinelt. The traveling salesman problem: Computational solutions for TSP applications. In *Lecture Notes in Computer Science 840*, p. 172– 186. Springer-Verlag, Berlin, 1994.

[RF06] S. Roger and T. Finley. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett, 2006.

[RFS98] M. Resende, T. Feo, and S. Smith. Algorithm 787: Fortran subroutines for approximate solution of maximum independent set problems using GRASP. *ACM Transactions on Mathematical Software*, 24:386–394, 1998.

[RHG07] S. Richter, M. Helert, and C. Gretton. A stochastic local search approach to vertex cover. In *Proc. 30th German Conf. on Artificial Intelligence (KI- 2007)*, 2007.

[RHS89] A. Robison, B. Hafner, and S. Skiena. Eight pieces cannot cover a chessboard. *Computer Journal*, 32:567–570, 1989.

[Riv92] R. Rivest. The MD5 message digest algorithm. RFC 1321, 1992.

[RR99] C.C. Ribeiro and M.G.C. Resende. Algorithm 797: Fortran subroutines for approximate solution of graph planarization problems using GRASP. *ACM Transactions on Mathematical Software*, 25:341–352, 1999.

[RS96] H. Rau and S. Skiena. Dialing for documents: an experiment in information theory. *Journal of Visual Languages and Computing*, p. 79–95, 1996.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[RSL77] D. Rosenkrantz, R. Stearns, and P. M. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM J. Computing*, 6:563–581, 1977.

[RSN+01] A. Rukihin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo. A statistical test suite for the validation of random number generators and pseudo random number generators for cryptographic applications. Technical Report Special Publication 800-22, NIST, 2001.

[RSS02] E. Rafalin, D. Souvaine, and I. Streinu. Topological sweep in degenerate cases. In *Proc. 4th Workshop on Algorithm Engineering and Experiments (ALENEX)*, p. 273–295, 2002.

[RSST96] N. Robertson, D. Sanders, P. Seymour, and R. Thomas. Efficiently fourcoloring planar graphs. In *Proc. 28th ACM Symp. Theory of Computing*, p. 571–575, 1996.

[RT81] E. Reingold and J. Tilford. Tidier drawings of trees. *IEEE Trans. Software Engineering*, 7:223–228, 1981.

[Rus03] F. Ruskey. Combinatorial Generation. Manuscript in preparation. Draft available at **http://www.1stworks.com/ref/RuskeyCombGen.pdf**, 2003.

[Ryt85] W. Rytter. Fast recognition of pushdown automata and context-free languages. *Information and Control*, 67:12–22, 1985.

[RZ05] G. Robins and A. Zelikovsky. Improved Steiner tree approximation in graphs. *Tighter Bounds for Graph Steiner Tree Approximation*, p. 122–134, 2005.

[SA95] M. Sharir and P. Agarwal. *Davenport-Schinzel sequences and their geometric applications*. Cambridge University Press, New York, 1995.

[Sah05] S. Sahni. Double-ended priority queues. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, p. 8:1–8:23. Chapman and Hall / CRC, 2005.

[Sal06] D. Salomon. *Data Compression: The Complete Reference*. Springer-Verlag, fourth edition, 2006.

[Sam05] H. Samet. Multidimensional spatial data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, p. 16:1– 16:29. Chapman and Hall / CRC, 2005.

[Sam06] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[San00] P. Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5, 2000.

[Sav97] C. Savage. A survey of combinatorial gray codes. *SIAM Review*, 39:605–629, 1997.

[Sax80] J. B. Saxe. Dynamic programming algorithms for recognizing smallbandwidth graphs in polynomial time. *SIAM J. Algebraic and Discrete Methods*, 1:363–369, 1980.

[Say05] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann, third edition, 2005.

[SB01] A. Samorodnitsky and A. Barvinok. The distance approach to approximate combinatorial counting. *Geometric and Functional Analysis*, 11:871–899, 2001.

[Sch96] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. Wiley, New York, second edition, 1996.

[Sch98] A. Schrijver. Bipartite edge-coloring in $O(\delta m)$ time. *SIAM J. Computing*, 28:841–846, 1998.

[SD75] M. Syslo and J. Dzikiewicz. Computational experiences with some transitive closure algorithms. *Computing*, 15:33–39, 1975.

[SD76] D. C. Schmidt and L. E. Druffel. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *J. ACM*, 23:433– 445, 1976.

[SDK83] M. Syslo, N. Deo, and J. Kowalik. *Discrete Optimization Algorithms with Pascal Programs*. Prentice Hall, Englewood Cliffs NJ, 1983.

[Sed77] R. Sedgewick. Permutation generation methods. *Computing Surveys*, 9:137–164, 1977.

[Sed78] R. Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21:847–857, 1978.

[Sed98] R. Sedgewick. *Algorithms in C++, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms*. Addison-Wesley, Reading MA, third edition, 1998.

[Sei04] R. Seidel. Convex hull computations. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, p. 495–512. CRC Press, 2004.

[SF92] T. Schlick and A. Fogelson. TNPACK – a truncated Newton minimization package for large-scale problems: I. algorithm and usage. *ACM Trans. Math. Softw.*, 18(1):46–70, March 1992.

[SFG82] M. Shore, L. Foulds, and P. Gibbons. An algorithm for the Steiner problem in graphs. *Networks*, 12:323–333, 1982.

[SH75] M. Shamos and D. Hoey. Closest point problems. In *Proc. Sixteenth IEEE Symp. Foundations of Computer Science*, p. 151–162, 1975.

[SH99] W. Shih and W. Hsu. A new planarity test. *Theoretical Computer Science*, 223(1–2):179–191, 1999.

[Sha87] M. Sharir. Efficient algorithms for planning purely translational collisionfree motion in two and three dimensions. In *Proc. IEEE Internat. Conf. Robot. Autom.*, p. 1326–1331, 1987.

[Sha04] M. Sharir. Algorithmic motion planning. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, p. 1037– 1064. CRC Press, 2004.

[She97] J. R. Shewchuk. Robust adaptive floating-point geometric predicates. *Disc. Computational Geometry*, 18:305–363, 1997.

[Sho05] V. Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, 2005.

[Sip05] M. Sipser. *Introduction to the Theory of Computation*. Course Technology, second edition, 2005.

[SK86] T. Saaty and P. Kainen. *The Four-Color Problem*. Dover, New York, 1986.

[SK99] D. Sankoff and J. Kruskal. *Time Warps, String Edits, and Macromolecules: the theory and practice of sequence comparison*. CSLI Publications, Stanford University, 1999.

[SK00] R. Skeel and J. Keiper. *Elementary Numerical computing with Mathematica*. Stipes Pub Llc., 2000.

[Ski88] S. Skiena. Encroaching lists as a measure of presortedness. *BIT*, 28:775–784, 1988.

[Ski90] S. Skiena. *Implementing Discrete Mathematics*. Addison-Wesley, Redwood City, CA, 1990.

S. Skiena. Who is interested in algorithms and why?: lessons from the stony brook algorithms repository. *ACM SIGACT News*, p. 65–74, September 1999.

[SL07] M. Singh and L. Lau. Approximating minimum bounded degree spanning tree to within one of optimal. In *Proc. 39th Symp. Theory Computing (STOC)*, p. 661–670, 2007.

[SLL02] J. Siek, L. Lee, and A. Lumsdaine. *The Boost Graph Library: user guide and reference manual*. Addison Wesley, Boston, 2002.

[SM73] L. Stockmeyer and A. Meyer. Word problems requiring exponential time. In *Proc. Fifth ACM Symp. Theory of Computing*, p. 1–9, 1973.

[Smi91] D. M. Smith. A Fortran package for floating-point multiple-precision arithmetic. *ACM Trans. Math. Softw.*, 17(2):273–283, June 1991.

[Sno04] J. Snoeyink. Point location. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, p. 767–785. CRC Press, 2004.

[SR83] K. Supowit and E. Reingold. The complexity of drawing trees nicely. *Acta Informatica*, 18:377–392, 1983.

[SR03] S. Skiena and M. Revilla. *Programming Challenges: The Programming Contest Training Manual*. Springer-Verlag, 2003.

[SS71] A. Schonhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. *Computing*, 7:281–292, 1971.

[SS02] R. Sedgewick and M. Schidlowsky. *Algorithms in Java, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms*. Addison-Wesley Professional, third edition, 2002.

[SS07] K. Schurmann and J. Stoye. An incomplex algorithm for fast suffix array construction. *Software: Practice and Experience*, 37:309–329, 2007.

[ST04] D. Spielman and S. Teng. Smoothed analysis: Why the simplex algorithm usually takes polynomial time. *J. ACM*, 51:385–463, 2004.

[Sta06] W. Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall, fourth edition, 2006.

[Str69] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14:354–356, 1969.

[SV87] J. Stasko and J. Vitter. Pairing heaps: Experiments and analysis. *Communications of the ACM*, 30(3):234–249, 1987.

[SV88] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, December 1988.

[SW86a] D. Stanton and D. White. *Constructive Combinatorics*. Springer-Verlag, New York, 1986.

[SW86b] Q. Stout and B.Warren. Tree rebalancing in optimal time and space. *Comm. ACM*, 29:902–908, 1986.

[SWA03] S. Schlieimer, D. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proc. ACM SIGMOD Int. Conf. on Management of data*, p. 76–85, 2003.

[Swe99] Z. Sweedyk. A 2.5-approximation algorithm for shortest superstring. *SIAM J. Computing*, 29:954–986, 1999.

[SWM95] J. Shallit, H. Williams, and F. Moraine. Discovery of a lost factoring machine. *The Mathematical Intelligencer*, 17-3:41–47, Summer 1995.

[Szp03] G. Szpiro. *Kepler's Conjecture: How Some of the Greatest Minds in History Helped Solve One of the Oldest Math Problems in the World*. Wiley, 2003.

[Tam08] R. Tamassia. *Handbook of Graph Drawing and Visualization*. Chapman-Hall / CRC, 2008.

[Tar95] G. Tarry. Le probleme de labyrinthes. *Nouvelles Ann. de Math.*, 14:187, 1895.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1:146–160, 1972.

[Tar75] R. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22:215–225, 1975.

[Tar79] R. Tarjan. A class of algorithms which require non-linear time to maintain disjoint sets. *J. Computer and System Sciences*, 18:110–127, 1979.

[Tar83] R. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, 1983.

[TH03] R. Tam and W. Heidrich. Shape simplification based on the medial axis transform. In *Proc. 14th IEEE Visualization (VIS-03)*, p. 481–488, 2003.

[THG94] J. Thompson, D. Higgins, and T. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–80, 1994.

[Thi03] H. Thimbleby. The directed chinese postman problem. *Software Practice and Experience.*, 33:1081–1096, 2003.

[Tho68] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11:419–422, 1968.

[Tin90] G. Tinhofer. Generating graphs uniformly at random. *Computing*, 7:235– 255, 1990.

[TNX08] K. Thulasiraman, T. Nishizeki, and G. Xue. *The Handbook of Graph Algorithms and Applications*, vol. 1: Theory and Optimization. Chapman- Hall/CRC, 2008.

[Tro62] H. F. Trotter. Perm (algorithm 115). *Comm. ACM*, 5:434–435, 1962.

[Tur88] J. Turner. Almost all k-colorable graphs are easy to color. *J. Algorithms*, 9:63–82, 1988.

[TV01] R. Tamassia and L. Vismara. A case study in algorithm engineering for geometric computing. *Int. J. Computational Geometry and Applications*, 11(1):15–70, 2001.

[TW88] R. Tarjan and C. Van Wyk. An $O(n \lg \lg n)$ algorithm for triangulating a simple polygon. *SIAM J. Computing*, 17:143–178, 1988.

[Ukk92] E. Ukkonen. Constructing suffix trees on-line in linear time. In *Intern. Federation of Information Processing (IFIP '92)*, p. 484–492, 1992.

[Val79] L. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[Val02] G. Valiente. *Algorithms on Trees and Graphs*. Springer, 2002.

[Van98] B. Vandegriend. Finding hamiltonian cycles: Algorithms, graphs and performance. M.S. Thesis, Dept. of Computer Science, Univ. of Alberta, 1998.

[Vaz04] V. Vazirani. *Approximation Algorithms*. Springer, 2004.

[VB96] L. Vandenberghe and S. Boyd. Semidefinite programming. *SIAM Review*, 38:49–95, 1996.

[vEBKZ77] P. van Emde Boas, R. Kaas, and E. Zulstra. Design and implementation of an efficient priority queue. *Math. Systems Theory*, 10:99–127, 1977.

[Vit01] J. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33:209–271, 2001.

[Viz64] V. Vizing. On an estimate of the chromatic class of a p-graph (in Russian). *Diskret. Analiz*, 3:23–30, 1964.

[vL90a] J. van Leeuwen. Graph algorithms. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Algorithms and Complexity*, vol. A, p. 525–631. MIT Press, 1990.

[vL90b] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science: Algorithms and Complexity*, vol. A. MIT Press, 1990.

[VL05] F. Viger and M. Latapy. Efficient and simple generation of random simple connected graphs with prescribed degree sequence. In *Proc. 11th Conf. on Computing and Combinatorics (COCOON)*, p. 440–449, 2005.

[Vos92] S. Voss. Steiner's problem in graphs: heuristic methods. *Discrete Applied Mathematics*, 40:45 – 72, 1992.

[Wal99] J. Walker. *A Primer on Wavelets and Their Scientific Applications*. CRC Press, 1999.

[War62] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9:11–12, 1962.

[Wat03] B. Watson. A new algorithm for the construction of minimal acyclic DFAs. *Science of Computer Programming*, 48:81–97, 2003.

[Wat04] D. Watts: *Six Degrees: The Science of a Connected Age*. W.W. Norton, 2004.

[WBCS77] J.Weglarz, J. Blazewicz, W. Cellary, and R. Slowinski. An automatic revised simplex method for constrained resource network scheduling. *ACM Trans. Math. Softw.*, 3(3):295–300, September 1977.

[WC04a] B. Watson and L. Cleophas. Spare parts: a C++ toolkit for string pattern recognition. *Software—Practice and Experience,*, 34:697–710, 2004.

[WC04b] B. Wu and K Chao. *Spanning Trees and Optimization Problems*. Chapman-Hall / CRC, 2004.

[Wei73] P. Weiner. Linear pattern-matching algorithms. In *Proc. 14th IEEE Symp. on Switching and Automata Theory*, p. 1–11, 1973.

[Wei06] M. Weiss. *Data Structures and Algorithm Analysis in Java*. AddisonWesley, second edition, 2006.

[Wel84] T. Welch. A technique for high-performance data compression. *IEEE Computer*, 17-6:8–19, 1984.

[Wes83] D. H. West. Approximate solution of the quadratic assignment problem. *ACM Trans. Math. Softw.*, 9(4):461–466, December 1983.

[Wes00] D. West. *Introduction to Graph Theory*. Prentice-Hall, Englewood Cliffs NJ, second edition, 2000.

[WF74] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21:168–173, 1974.

[Whi32] H. Whitney. Congruent graphs and the connectivity of graphs. *American J. Mathematics*, 54:150–168, 1932.

[Wig83] A. Wigerson. Improving the performance guarantee for approximate graph coloring. *J. ACM*, 30:729–735, 1983.

[Wil64] J. W. J. Williams. Algorithm 232 (heapsort). *Communications of the ACM*, 7:347–348, 1964.

[Wil84] H. Wilf. Backtrack: An O(1) expected time algorithm for graph coloring. *Info. Proc. Letters*, 18:119–121, 1984.

[Wil85] D. E. Willard. New data structures for orthogonal range queries. *SIAM J. Computing*, 14:232–253, 1985.

[Wil89] H. Wilf. *Combinatorial Algorithms: an update*. SIAM, Philadelphia PA, 1989.

[Win68] S. Winograd. A new algorithm for inner product. *IEEE Trans. Computers*, C-17:693–694, 1968.

[Win80] S. Winograd. *Arithmetic Complexity of Computations*. SIAM, Philadelphia, 1980.

[WM92a] S. Wu and U. Manber. Agrep — a fast approximate pattern-matching tool. In *Usenix Winter 1992 Technical Conference*, p. 153–162, 1992.

[WM92b] S. Wu and U. Manber. Fast text searching allowing errors. *Comm. ACM*, 35:83–91, 1992.

[Woe03] G. Woeginger. Exact algorithms for NP-hard problems: A survey. In *Combinatorial Optimization — Eureka! You shrink!*, vol. 2570 Springer-Verlag LNCS, p. 185–207, 2003.

[Wol79] T. Wolfe. *The Right Stuff*. Bantam Books, Toronto, 1979.

[WW95] F. Wagner and A. Wolff. Map labeling heuristics: provably good and practically useful. In *Proc. 11th ACM Symp. Computational Geometry*, p. 109–118, 1995.

[WWZ00] D. Warme, P. Winter, and M. Zachariasen. Exact algorithms for plane Steiner tree problems: A computational study. In D. Du, J. Smith, and J. Rubinstein, editors, *Advances in Steiner Trees*, p. 81–116. Kluwer, 2000.

[WY05] X. Wang and H. Yu. How to break MD5 and other hash functions. In *EUROCRYPT, LNCS* vol. 3494, p. 19–35, 2005.

[Yan03] S. Yan. *Primality Testing and Integer Factorization in Public-Key Cryptography.* Springer, 2003.

[Yao81] A. C. Yao. A lower bound to finding convex hulls. *J. ACM*, 28:780–787, 1981.

[Yap04] C. Yap. Robust geometric computation. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, p. 607–641. CRC Press, 2004.

[YLCZ05] R Yeung, S-Y. Li, N. Cai, and Z. Zhang. *Network Coding Theory.* **http://www.nowpublishers.com/**, Now Publishers, 2005.

[You67] D. Younger. Recognition and parsing of context-free languages in time $O(n^3)$. *Information and Control*, 10:189–208, 1967.

[YS96] F. Younas and S. Skiena. Randomized algorithms for identifying minimal lottery ticket sets. *Journal of Undergraduate Research*, 2-2:88–97, 1996.

[YZ99] E. Yang and Z. Zhang. The shortest common superstring problem: Average case analysis for both exact and approximate matching. *IEEE Trans. Information Theory*, 45:1867–1886, 1999.

[Zar02] C. Zaroliagis. Implementations and experimental studies of dynamic graph algorithms. In *Experimental algorithmics: from algorithm design to robust and efficient software*, p. 229–278. Springer-Verlag LNCS, 2002.

[ZL78] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory*, IT-23:337–343, 1978.

[ZS04] Z. Zaritsky and M. Sipper. The preservation of favored building blocks in the struggle for fitness: The puzzle algorithm. *IEEE Trans. Evolutionary Computation*, 8:443–455, 2004.

[Zwi01] U. Zwick. Exact and approximate distances in graphs – a survey. In *Proc. 9th Euro. Symp. Algorithms (ESA)*, p. 33–48, 2001.

Предметный ука затель

Б

В

Г

Д

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

Стивен С. Скиена

Алгоритмы. Руководство по разработке

2-е издание

Перевод с английского

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Татьяна Лапина*
Зав. редакцией	*Григорий Добин*
Перевод с английского	*Сергея Таранушенко*
Редактор	*Ирина Иноземцева*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 29.04.11.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 58,05.
Тираж 1500 экз. Заказ № 3741
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.005770.05.09 от 26.05.2009 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

Наиболее полное руководство по разработке эффективных алгоритмов!

Алгоритмы. Руководство по разработке. *Второе издание*

– Эта книга является настоящей сокровищницей алгоритмов, собрать которые в одном месте было работой не из легких. Каталог задач и обширная библиография делают книгу неоценимым подспорьем для любого, кто интересуется этой темой.

Обозрение Ассоциации вычислительной техники (www.reviews.com)

Книга является наиболее полным руководством по разработке эффективных алгоритмов. Первая часть книги содержит практические рекомендации по разработке алгоритмов: приводятся основные понятия, дается анализ алгоритмов, рассматриваются типы структур данных, основные алгоритмы сортировки, операции обхода графов и алгоритмы для работы со взвешенными графами, примеры использования комбинаторного поиска, эвристических методов и динамического программирования. Вторая часть книги содержит каталог наиболее распространенных алгоритмических задач, для которых перечислены существующие программные реализации. Приведен обширный список литературы.

- Большой объем обучающего материала и упражнений
- Выделение основных понятий в конце каждой главы
- Уникальный каталог наиболее часто встречающихся на практике 75 алгоритмических задач
- Ссылки на литературу и интернет-ресурсы по реализации алгоритмов на языках C, C++ и Java
- Примеры задач для соискателей при приеме на работу в компании по разработке программного обеспечения

Книгу можно использовать в качестве справочника по алгоритмам для программистов, исследователей и в качестве учебного пособия для студентов соответствующих специальностей.

Об авторе: Стивен С. Скиена (Steven S. Skiena), профессор кафедры вычислительной техники университета Стоуни — Брук, известный исследователь алгоритмов, лауреат премии института IEEE, автор популярной книги «Programming Challenges: The Programming Contest Training Manual» («Олимпиадные задачи по программированию. Руководство по подготовке к соревнованиям»).

ISBN 978-5-9775-0560-4

Полный набор слайдов лекций автора книги и другой дополнительный материал находится на веб-сайте www.algorist.com

БХВ-Петербург
190005, Санкт-Петербург, Измайловский пр., 29
E-mail: mail@bhv.ru
Internet: www.bhv.ru
Тел.: (812) 251-42-44
Факс: (812) 320-01-79