O'REILLY®

Это практическое руководство научит разработчиков приложений и аналитиков данных извлекать максимум пользы из анализа графов при решении разных задач – от строительства динамических сетей до предсказания поведения в реальном мире. Вы узнаете, как графовые алгоритмы помогают использовать связи внутри данных при разработке интеллектуальных решений и совершенствовать модели машинного обучения.

Марк Нидхем и Эми Ходлер из Neo4j поясняют, как графовые алгоритмы раскрывают сложные структуры и выявляют скрытые закономерности – от выявления уязвимостей и узких мест системы до обнаружения сообществ и повышения точности машинных прогнозов. Вы познакомитесь с практическими примерами использования графовых алгоритмов на платформах Apache Spark и Neo4j.

«Авторы составили полезный и наглядный путеводитель по удивительному миру графов, от базовых концепций до фундаментальных алгоритмов, платформ обработки и практических примеров».

Кирк Борн,
старший советник
по науке и анализу данных,
Booz Allen Hamilton, Inc.

«Полезное и емкое руководство по изучению связанных данных путем поиска закономерностей и структур при помощи графовых алгоритмов. Эту книгу должны прочесть все разработчики, имспользующие графовые базы данных».

Луанна Мискитта,
вице-президент по технологиям,
GraphAware

- Узнайте, как при помощи анализа графов извлечь более предсказательные элементы из современных данных.
- Изучите популярные графовые алгоритмы и научитесь их применять.
- Узнайте, какие алгоритмы применяются для различных типов задач.
- Исследуйте примеры рабочего кода и наборов данных для Spark и Neo4j.
- Пройдите полный цикл создания машиннообучаемой модели, используя комбинацию Neo4j и Spark.

Марк Нидхем, инженер по связанным данным в Neo4j, помогает пользователям освоить графы и Neo4j, находя продвинутые решения сложных проблем с данными.
Эми Ходлер – преданный энтузиаст сетевых наук, руководитель программ по анализу графов в Neo4j. Эми сотрудничает с такими компаниями, как EDS, Microsoft, Hewlett-Packard (HP), Hitachi IoT и Cray Inc.

ISBN 978-5-97060-799-2

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК «Галактика»
books@alians-kniga.ru

ДМК ПРЕСС ИЗДАТЕЛЬСТВО
www.дмк.рф

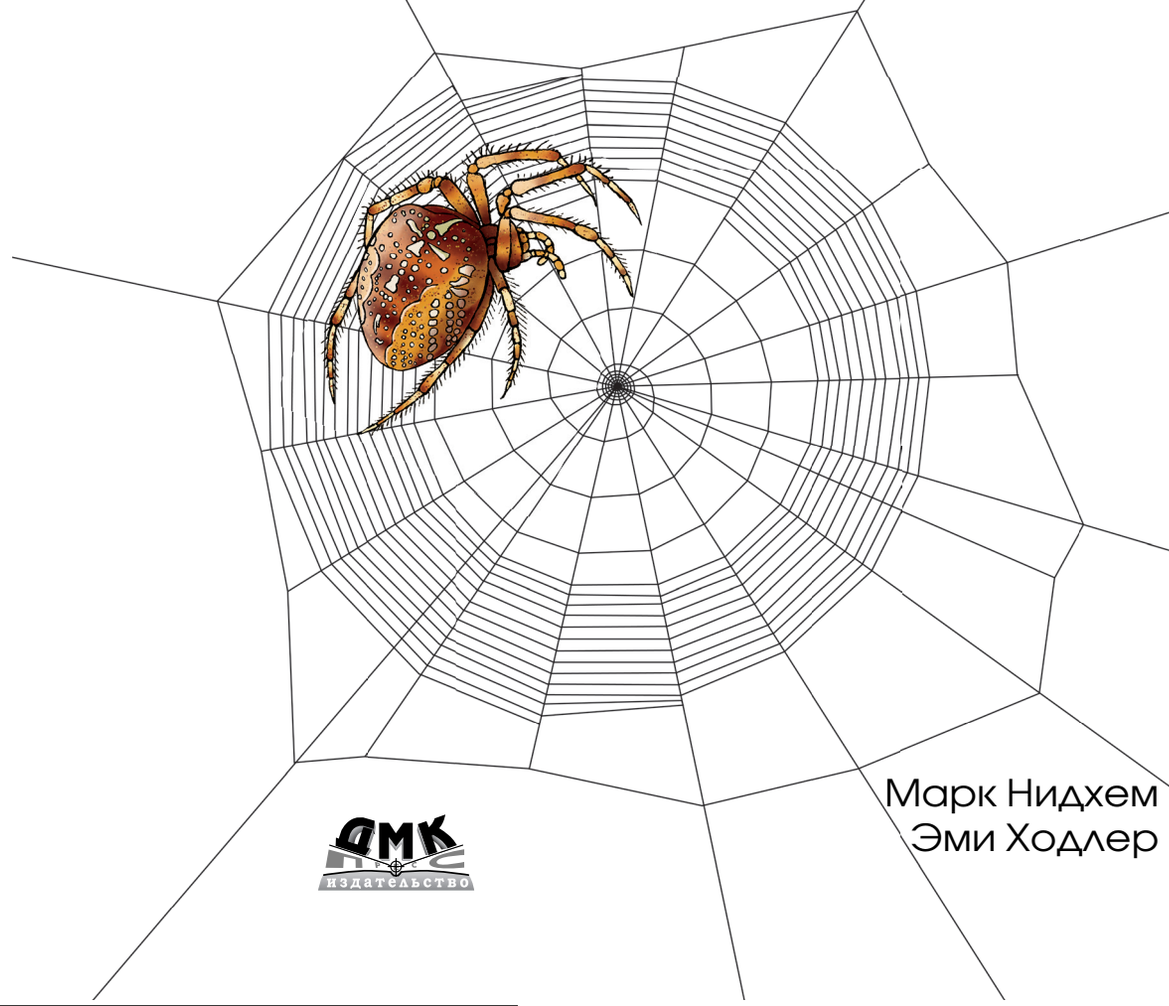
Графовые алгоритмы

O'REILLY®

O'REILLY®

Графовые алгоритмы

Практическая реализация на платформах Apache Spark и Neo4j



Марк Нидхем
Эми Ходлер

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Марк Нидхем
Эми Ходлер

Графовые алгоритмы

Graph Algorithms

Practical Examples in Apache Spark and Neo4j

Mark Needham and Amy E. Hodler

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

Графовые алгоритмы

Практическая реализация
на платформах Apache Spark и Neo4j

Марк Нидхем
Эми Ходлер

Перевод с английского Яценкова В. С.

Москва, 2020

УДК 004.021
ББК 32.973.1
Н60

Н60 Марк Нидхем, Эми Ходлер

Графовые алгоритмы. Практическая реализация на платформах Apache Spark и Neo4j. / пер. с англ. В. С. Яценкова – М.: ДМК Пресс, 2020. – 258 с.: ил.

ISBN 978-5-97060-799-2

Каждую секунду во всем мире собирается и динамически обновляется огромный объем информации. Графовые алгоритмы, которые основаны на математике, специально разработанной для изучения взаимосвязей между данными, помогают разобраться в этих гигантских объемах. И, что особенно важно в наши дни, они улучшают контекстную информацию для искусственного интеллекта.

Эта книга представляет собой практическое руководство по началу работы с графовыми алгоритмами. В начале описания каждой категории алгоритмов приводится таблица, которая поможет быстро выбрать нужный алгоритм и ознакомиться с примерами его использования.

Издание предназначено для разработчиков и специалистов по анализу данных. Для изучения материала книги желателен опыт использования платформ Apache Spark™ или Neo4j, но она пригодится и для изучения более общих понятий теории графов, независимо от выбора графовых технологий.

УДК 004.021
ББК 32.973.1

ISBN 978-1-49204-768-1 (англ.)
ISBN 978-5-97060-799-2 (рус.)

Оглавление

Предисловие

Миром правят связи – повсеместно, от финансовых и коммуникационных систем до социальных и биологических процессов. Выявление скрытого смысла этих связей приводит к прорывным решениям различных задач, таких как выявление мошеннических звонков, оптимизация связей в рабочей группе или прогнозирование каскадных сетевых сбоев.

Поскольку связность мира продолжает нарастать, неудивительно, что возрастает интерес к графовым алгоритмам, потому что они основаны на математике, специально разработанной для изучения взаимосвязей между данными. Анализ графов может раскрыть работу сложных систем и сетей в огромных масштабах – для любой организации.

Мы искренне увлечены полезностью и важностью анализа графов и с удовольствием расшифровываем тонкости внутренней работы сложных сценариев. До недавнего времени применение анализа графов требовало значительного опыта и упорства, потому что инструменты и средства интеграции были трудными в освоении, и лишь немногие знали, как применять графовые алгоритмы к своим задачам. Наша цель – помочь вам преодолеть трудности. Мы написали эту книгу, чтобы исследователи данных начали в полной мере использовать анализ графов, а значит, могли делать новые открытия и быстрее разрабатывать интеллектуальные решения.

О чем эта книга

Эта книга представляет собой практическое руководство по началу работы с графовыми алгоритмами для разработчиков и специалистов по анализу данных, которые имеют опыт использования Apache Spark™ или Neo4j. Хотя в наших примерах алгоритмов используются платформы Spark и Neo4j, эта книга также пригодится для изучения более общих понятий теории графов, независимо от вашего выбора графовых технологий.

Первые две главы содержат введение в теорию, анализ графов и графовые алгоритмы. В третьей главе кратко рассмотрены платформы, используемые в этой книге, прежде чем мы углубимся в следующие три главы, посвященные классическим графовым алгоритмам – нахождению пути, вычислению центральности и выделению сообществ. Мы завершим книгу двумя главами, показывающими, как графовые алгоритмы используются в рабочих процессах – один для общего анализа и другой для машинного обучения.

В начале описания каждой категории алгоритмов есть справочная таблица, которая поможет вам быстро выбрать нужный алгоритм. Для каждого алгоритма вы найдете:

- объяснение того, что делает алгоритм;
- примеры использования алгоритма и ссылки, где вы можете узнать больше;
- примеры кода, демонстрирующие способы реализации алгоритма в Spark и Neo4j.

Условные обозначения, принятые в книге

В книге имеются следующие условные обозначения:

Курсив

Используется для смыслового выделения важных положений, новых терминов, URL-адресов и адресов электронной почты в интернете, имен команд и утилит, а также имен и расширений файлов и каталогов.

`Моноширинный шрифт`

Используется для листингов программ, а также в обычном тексте для обозначения имен переменных, функций, типов, объектов, баз данных, переменных среды, операторов, ключевых слов и других программных конструкций и элементов исходного кода.

`Моноширинный полужирный шрифт`

Используется для обозначения команд или фрагментов текста, которые пользователь должен ввести дословно, без изменений.

`Моноширинный курсив`

Используется для обозначения в исходном коде или в командах шаблонных меток-заполнителей, которые должны быть заменены соответствующими контексту реальными значениями.

Такая пиктограмма обозначает совет или рекомендацию.

Обозначает указание или примечание общего характера.

Эта пиктограмма обозначает предупреждение или особое внимание к потенциально опасным объектам.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте **www.dmkpress.com**, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте **www.dmkpress.com** или **www.дмк.рф** на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры, для того чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и O'Reilly очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты **dmkpress@gmail.com** со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Благодарности

Нам очень понравилось работать над этой книгой благодаря помощи наших друзей. Мы особенно благодарны Майклу Хангеру (Michael Hunger) за его руководство, Джиму Уэбберу (Jim Webber) за его бесценные правки и Томазу Братаничу (Tomaz Bratanic) за его увлекательные исследования. Наконец, мы искренне благодарим каталог Yelp, который позволяет нам использовать его богатый набор данных в качестве практических примеров.

Вступительное слово рецензента

Как вы думаете, что общего имеют следующие вещи: анализ маркетинговых взаимосвязей, противодействие отмыванию денег, моделирование поездок клиентов, анализ инцидентов безопасности, анализ литературных источников, обнаружение мошеннических сетей, анализ поисковых узлов в интернете, создание картографических приложений, анализ распространения эпидемий и изучение пьес Уильяма Шекспира? Как вы уже догадались, общим для всех них является использование графов, доказывающих, что Шекспир был прав, когда заявил: «Весь мир – это граф!»

Ну хорошо, великий бард с берегов Эйвона на самом деле не говорил про граф, он сказал «театр». Тем не менее обратите внимание, что приведенные выше примеры включают в себя сущности и отношения между ними, как прямые, так и косвенные. Сущности – вершины графа – могут быть людьми, событиями, объектами, концепциями или местами. Отношения между вершинами – это ребра графа. Но разве сама суть шекспировской пьесы не заключается в изображении героев (вершин) и их отношений (ребер)? Следовательно, Шекспир в своей знаменитой фразе имел полное право сказать про граф.

Что делает графовые алгоритмы и графовые базы данных такими интересными и мощными, так это не простые отношения между двумя сущностями, где *A* связан с *B*. В конце концов, стандартная реляционная модель баз данных использует эти типы отношений уже несколько десятилетий. Что на самом деле делает графы настолько удивительными – это *направленные и транзитивные отношения*. В направленных отношениях *A* может вызвать *B*, но не наоборот. В транзитивных отношениях *A* может быть непосредственно связан с *B*, а *B* может быть напрямую связан с *C*, в то время как *A* не имеет прямого отношения к *C*. Следовательно, А *транзитивно связан* с С.

Благодаря транзитивным отношениям – особенно когда они многочисленны и разнообразны, с различными паттернами отношений и степеней разделения между объектами – графовая модель раскрывает связи между объектами, которые в противном случае могут показаться не связанными или независимыми в обычной реляционной базе данных. Следовательно, во многих задачах сетевого анализа можно эффективно применять графовую модель.

Рассмотрим пример *маркетинговой атрибуции*[1] (marketing attribution): человек *A* видит маркетинговую кампанию; человек *A* пишет комментарий в социальных сетях; человек *B* связан с человеком *A* и видит комментарий; и впоследствии человек *B* покупает продукт. С точки зрения менеджера маркетинговой кампании, стандартная реляционная модель не может определить атрибуцию, поскольку персонаж *B* не видел кампанию, а персонаж A не купил продукт. Кампания выглядит как провал, но ее фактический успех (и положительная рентабельность инвестиций) обнаруживается алгоритмом анализа графов через транзитивные отношения между маркетинговой кампанией, посредником и конечной покупкой.

Далее рассмотрим пример противодействия отмыванию денег: лица *A* и *C* подозреваются в обороте денег, полученных незаконным путем. Любая прямая сделка между ними – например, транзакция через расчетно-кассовый центр – будет без труда зафиксирована и подвергнута строгому контролю. Однако, если *A* и *C* никогда не совершают взаимные сделки, а вместо этого проводят финансовые операции через надежные, уважаемые и незапятнанные финансовые органы *B*, что может провалить сделку? Алгоритм анализа графов! Графовый движок обнаружит транзитивные отношения между *A* и *C* через посредника *B*.

Отвечая на поисковый запрос, основные поисковые системы интернета используют алгоритмы на основе графов, чтобы найти самый авторитетный сайт во всем интернете для любого заданного набора поисковых слов. В этом случае принципиально важна направленность связей, поскольку авторитетным сайтом в сети является тот, на который ссылается большинство других сайтов.

Сегодня существует особая отрасль анализа данных – *литературный поиск* (literature-based discovery, LBD), технология на основе графов, позволяющая искать открытия в базе знаний тысяч (или даже миллионов) статей исследовательских журналов. Скрытые знания обнаруживаются только через связи между опубликованными результатами исследований, которые могут иметь много этапов переходных отношений. LBD активно применяется для поиска методов лечения рака, где обширная база семантических медицинских знаний о симптомах, диагнозах, методах лечения, взаимодействиях лекарств, генетических маркерах, краткосрочных результатах и долгосрочных последствиях может таить в себе ранее неизвестные или потенциально полезные методы лечения для самых безнадежных случаев. Это невероятно – знание уже может лежать в сети, и нам остается лишь соединить точки, чтобы найти его.

---

[1] Маркетинговая атрибуция – это анализ отдачи от точек взаимодействия с клиентом. Любимая поговорка маркетологов гласит: «Половина денег, которые я трачу на рекламу, выбрасывается впустую. Беда в том, что я не знаю, какая половина». Атрибуция призвана дать ответ на этот вопрос. – *Прим. перев.*

Подобные описания возможностей, реализуемых через построение графов, могут быть даны и для других упомянутых выше примеров – по сути, это примеры сетевого анализа с помощью графовых алгоритмов. Каждый случай глубоко исследует сущности (люди, объекты, события, действия, концепции и места) и их отношения (как причинно-следственные связи, так и простые ассоциации).

Обсуждая выгоды от построения графов, мы должны помнить, что в реальных ситуациях, возможно, самым мощным фактором в графовой модели является *контекст*. Он может включать время, местоположение, связанные события, соседние объекты и многое другое. Включение контекста в граф в виде вершин и ребер может дать впечатляющий толчок прогнозирующей и описательной аналитике.

Цель книги «*Графовые алгоритмы*» Марка Нидхема и Эми Ходлер – расширить наши знания и навыки в прикладном анализе графов, включая алгоритмы, концепции и практическое применение алгоритмов в машинном обучении. Авторы составили полезный и наглядный путеводитель по удивительному миру графов – от базовых концепций до фундаментальных алгоритмов, платформ обработки и практического использования.

Кирк Борн (Kirk Borne),
доктор наук, старший советник по науке и анализу данных,
консалтинговая компания Booz Allen Hamilton, Inc.
март 2019

Глава 1

Введение

«Графы являются одним из объединяющих понятий информатики – абстрактное представление, которое описывает организацию транспортных систем, взаимодействие между людьми и телекоммуникационные сети. То, что с помощью одного формального представления можно смоделировать так много различных структур, является источником огромной силы для образованного программиста».

Стивен С. Скиена[1]*,*
заслуженный профессор кафедры информатики,
университет Стони Брук

В настоящее время наиболее насущные проблемы анализа данных связаны с отношениями, а не просто с размером таблицы дискретных данных. Графовые технологии и анализ графов предоставляют мощные инструменты для работы со связанными данными, которые используются в исследованиях, социальных инициативах и бизнес-решениях, например:

- моделирование динамических сред от финансовых рынков до IT-сервисов;
- прогнозирование распространения эпидемий болезней, а также периодических задержек и сбоев в компьютерных сетях;
- поиск прогностических признаков для машинного обучения систем борьбы с финансовыми преступлениями;
- выявление шаблонов поведения для персонализированного опыта и рекомендаций.

Поскольку данные становятся все более взаимосвязанными, а системы – все более сложными, крайне важно использовать обширные отношения, скрытые в наших данных.

Эта глава содержит введение в анализ графов и графовые алгоритмы. Но прежде чем обсуждать графовые алгоритмы и объяснять разницу между графовыми базами данных и обработкой графов, мы начнем с крат-

[1] The Algorithm Design Manual, by *Steven S. Skiena*, Springer.

кого рассказа о происхождении графов. Затем мы рассмотрим природу современных данных и убедимся, что информация, содержащаяся в связях, гораздо сложнее, чем то, что мы можем выявить с помощью обычных статистических методов. Глава завершится обсуждением вариантов использования графовых алгоритмов.

Что такое графы?

Графы ведут свою историю с 1736 года, когда Леонард Эйлер решил знаменитую задачу «Семи мостов Кенигсберга». Вопрос заключался в том, можно ли посетить все четыре района города, соединенные семью мостами, при этом пересекая каждый мост только один раз.

Размышляя над задачей, Эйлер понял, что для решения имеют значение только связи, и тем самым заложил основы теории графов и ее математики. На рис. 1.1 изображен ход рассуждений Эйлера и один из его оригинальных набросков из статьи «*Solutio problematis ad geometriam situs pertinentis*».

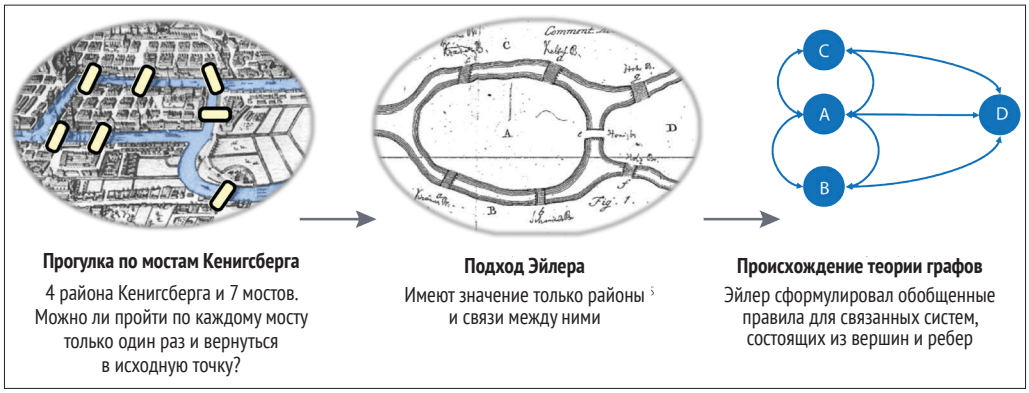


Рис. 1.1. Истоки теории графов. Город Кенигсберг включал два больших острова, соединенных друг с другом и материковой частью города семью мостами. Задача состояла в том, чтобы построить замкнутый маршрут, проходящий по каждому мосту только один раз

Хотя понятие графов зародилось в математике, они также являются удобным и точным способом моделирования и анализа данных. Объекты, составляющие граф, называются *узлами* или *вершинами*, а связи между ними называются *отношениями*, *связями* или *ребрами*. В этой книге мы используем термины *вершины* и *ребра*. Вы можете думать о вершинах как о существительных в предложении, а о ребрах как о глаголах, создающих смысловой контекст, т. е. связи. В английском языке графы, графики и различные диаграммы обозначаются одним словом *graph*, поэтому мы сразу хотим подчеркнуть, что графы, о которых пойдет речь в этой книге, не имеют ничего общего с построением графиков уравнений или диаграмм, изображенных в правой части рис. 1.2.

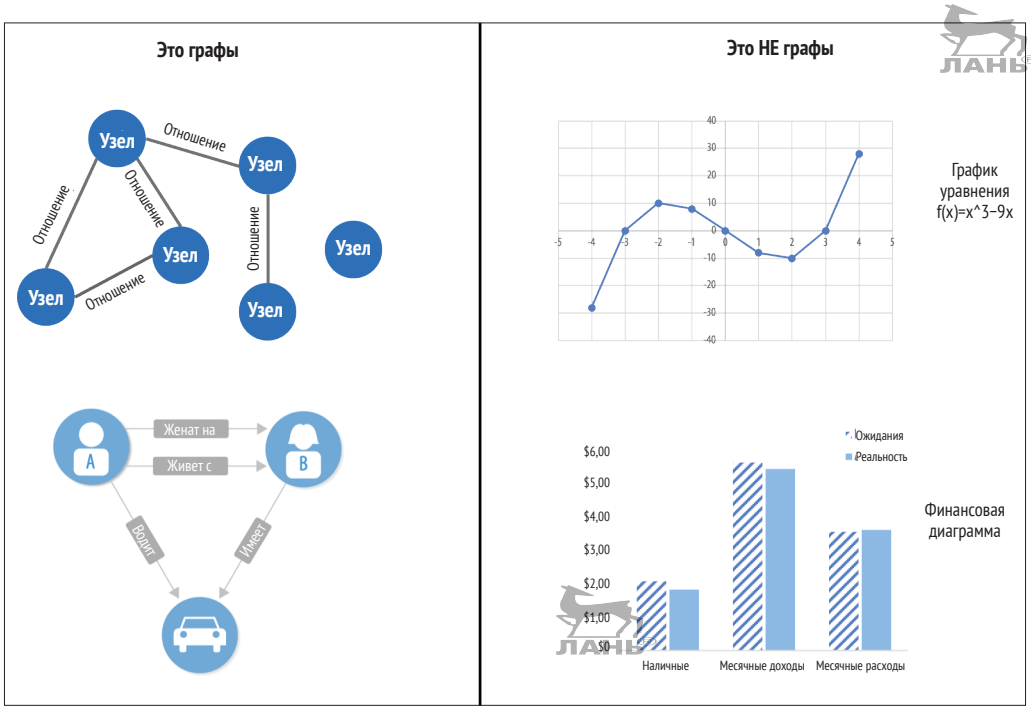


Рис. 1.2. Граф представляет собой схему взаимосвязей, часто изображаемую в виде кружков, называемых вершинами, и соединительных линий, называемых ребрами

Глядя на граф человека в левой части рис. 1.2, мы можем легко составить несколько предложений, которые описывают его как личность. Например, человек *A* живет с человеком *B*, которому принадлежит автомобиль, и человек *A* ведет автомобиль, принадлежащий человеку *B*. Такой подход к моделированию убедителен, поскольку он отлично сопоставляется с реальным миром и, как говорится, «whiteboard friendly», – схему отношений можно запросто набросать маркером на лекционной доске. Это помогает согласовать моделирование и анализ данных.

Но построение графов – это только половина дела. Мы ведь захотим обработать их, чтобы выявить скрытый смысл. Тут-то и вступают в дело графовые алгоритмы.

Что такое графовые алгоритмы и анализ графов?

Графовые алгоритмы являются подмножеством инструментария для анализа графов. В свою очередь, *анализ графов* – то, чем мы с вами занимаемся – это использование любого графового подхода для анализа связанных данных. Мы можем использовать разные методы – извлекать данные из графа, использовать обычную статистику, исследовать графы визуально

или включать их в задачи машинного обучения. Графовые запросы на основе шаблонов часто используются для локального анализа данных, тогда как вычислительные графовые алгоритмы обычно используют более глобальный и итеративный анализ. Хотя использование этих типов анализа частично совпадает, мы используем термин *графовые алгоритмы* для обозначения последнего, более вычислительного анализа и использования графов в науке о данных.

Наука о сетях

Наука о сетях (network science) – это академическая область науки, которая прочно опирается на теорию графов и связана с математическими моделями отношений между объектами. Ученые в этой области вынуждены полагаться на графовые алгоритмы и системы управления базами данных из-за размера, связности и сложности данных, с которыми им приходится работать.

Есть много великолепных ресурсов, посвященных науке о сетях и связанных данных. Вот несколько ссылок для самостоятельного изучения:

- http://networksciencebook.com/ – хорошая вводная онлайн-книга;
- https://www.complexityexplorer.org – онлайн-курс Института Санта-Фе;
- https://necsi.edu – различные ресурсы и документы на сайте Института сложных систем Новой Англии.

Графовые алгоритмы предоставляют один из наиболее эффективных подходов к анализу связанных данных, потому что их математические вычисления специально созданы для работы с отношениями. Они описывают действия, которые необходимо предпринять для обработки графа, чтобы выявить его общие качества или конкретные количества. Основываясь на математике теории графов, алгоритмы используют отношения между вершинами, чтобы проанализировать организацию и динамику сложных систем. В науке о сетях эти алгоритмы применяют для выявления скрытой информации, проверки гипотез и прогнозирования поведения.

Графовые алгоритмы обладают широким потенциалом – от предотвращения мошенничества и поиска оптимальных маршрутов телефонных звонков до прогнозирования распространения гриппа. Например, мы можем оценить, какие подстанции подвержены наибольшему риску при перегрузке энергосистемы. Или мы можем обнаружить группировки в графе, способствующие перегрузке транспортной системы.

И в самом деле, в 2010 году в системах воздушных перевозок США произошли два серьезных инцидента, связанных с перегрузкой нескольких аэропортов, которые впоследствии были изучены с использованием ана-

лиза графов. Сетевые ученые П. Флёркин, Дж. Рамаско и В. М. Игалез использовали графовые алгоритмы для выявления событий, вызывающих систематические каскадные задержки, и использовали эту информацию для составления рекомендаций, как описано в их статье[2] «Распространение системной задержки в аэропортах США».

Мартин Гранджин создал визуализацию всемирной сети воздушного транспорта (рис. 1.3) в своей статье[3] «Связанный мир: распутывая сеть воздушного движения». Эта иллюстрация ясно показывает сильно связанную структуру авиатранспортных кластеров. Многие транспортные системы демонстрируют концентрированное распределение связей с четкими узорами типа «ступица и спица», которые влияют на задержки.

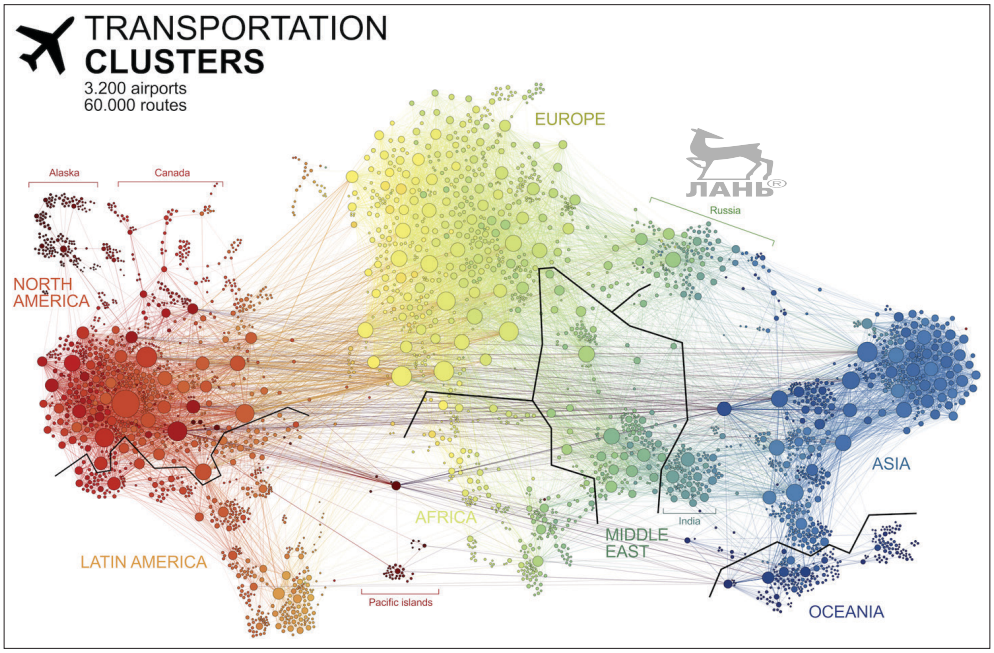


Рис. 1.3. Авиатранспортные сети содержат структуры «ступица и спицы» разной величины. От этих структур зависит перемещение потоков пассажиров и грузов

Графы также помогают раскрыть, как очень маленькие взаимодействия в динамике приводят к глобальным мутациям. Графы связывают воедино микро- и макроуровни, точно отражая, какие сущности взаимодействуют внутри глобальных структур. Эти ассоциации применяются для прогнозирования поведения и определения недостающих связей. Рисунок 1.4 – это сеть взаимодействий видов лугопастбищных угодий, которая использует анализ графов для оценки иерархической организации и взаимодействий видов, а затем предсказывает недостающие взаимосвязи, как подробно

[2] *P. Fleurquin, J. J. Ramasco,* and *V. M. Eguíluz,* «Systemic Delay Propagation in the US Airport Network».

[3] *Martin Grandjean,* «Connected World: Untangling the Air Traffic Network».

описано в статье[4] А. Клаусета, К. Мура и М. Э. Ньюмана «Иерархическая структура и прогнозирование отсутствующих ссылок в сети».

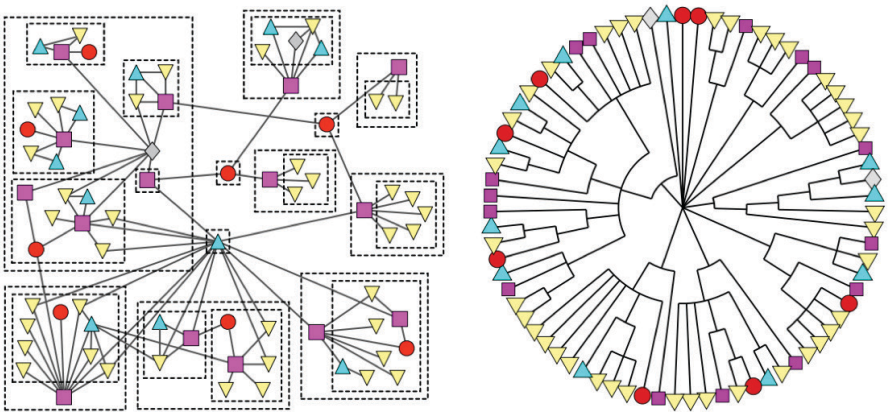


Рис. 1.4. Эта пищевая сеть видов пастбищ использует графы для выявления корреляции мелких взаимодействий с образованием более крупных структур

Обработка графов, базы данных, запросы и алгоритмы

Обработка графов включает методы, с помощью которых выполняются рабочие операции и задачи, связанные с графами. Большинство запросов к графу рассматривает конкретные части графа (например, начальную вершину), и работа обычно сосредоточена на окружающем подграфе. Мы называем этот тип взаимодействия *локальным*, и он подразумевает декларативные обращения к структуре графа, о чем рассказано в книге[5] Яна Робинсона, Джима Вебера и Эмиля Эифрема. Данная разновидность локальной обработки графа часто используется для транзакций в реальном времени и запросов на основе шаблонов.

Говоря о графовых алгоритмах, мы обычно ищем глобальные шаблоны и структуры. Входными данными для алгоритма обычно является *весь* граф, а выходными данными может быть обогащенный граф или какое-либо совокупное значение, такое как оценка. Мы называем такое взаимодействие *глобальным*, и это подразумевает обработку полной структуры графа с использованием вычислительных алгоритмов – зачастую итеративно. Подобный подход раскрывает общий характер сети через ее связи. Организации, как правило, используют графовые алгоритмы для моделирования

[4] *A. Clauset*, *C. Moore*, and *M. E. J. Newman*, «Hierarchical Structure and the Prediction of Missing Links in Network».

[5] *Ян Робинсон*, *Джим Вебер*, *Эмиль Эифрем* «Графовые базы данных. Новые возможности для работы со связанными данными», ДМК Пресс, 2016.

систем и прогнозирования поведения на основе анализа распространения взаимодействий, выявления сообществ, оценки важности компонентов и общей надежности системы.

Эти определения могут частично пересекаться – иногда мы можем использовать алгоритм для ответа на локальный запрос или наоборот, – но, упрощенно говоря, действия над целым графом выполняются вычислительными алгоритмами, а операции с подграфами реализуются через запросы к базам данных.

Традиционно выполнение и анализ транзакций были разделены – противоестественный раскол, основанный на технологических ограничениях. По нашему мнению, анализ графов способствует выполнению более разумных транзакций, что создает новые данные и возможности для последующего анализа. В последнее время появились подходы, преодолевающие упомянутое разделение и способствующие более оперативным решениям.

OLTP и OLAP

Оперативная обработка транзакций (online transaction processing, OLTP) – это, как правило, короткие действия, такие как бронирование билета, пополнение счета, резервирование товара и т. д. OLTP подразумевает массовую обработку запросов с низкой задержкой и высокую целостность данных. Хотя OLTP может включать только небольшое количество записей на транзакцию, системы обрабатывают много транзакций одновременно.

Оперативный анализ данных (online analytical processing, OLAP) облегчает более сложные запросы и анализ исторических данных. Подобный анализ может задействовать несколько источников данных, форматов и типов. Типичными случаями использования OLAP являются обнаружение тенденций, выполнение сценариев типа «*что, если*», прогнозирование и выявление структурных шаблонов. По сравнению с OLTP системы OLAP обрабатывают меньшее количество более длительных транзакций по многим записям. Системы OLAP склонны к быстрому чтению без ожидания подтверждения транзакций, присущего OLTP, поэтому обычной практикой является пакетная обработка.

Однако в последнее время грань между OLTP и OLAP начала стираться. Современные приложения с интенсивным использованием данных теперь сочетают транзакции в реальном времени с аналитикой. Это слияние вызвано современными достижениями в программном обеспечении, такими как более масштабируемое управление транзакциями и добавочная обработка потоков, а также удешевлением быстродействующего оборудования с большой памятью.

Объединение аналитики и транзакций позволяет проводить непрерывный анализ данных как естественную часть регулярных операций. Совре-

менная аналитика обеспечивает возможность выработки рекомендаций и решений по мере сбора данных – с компьютеров в кассовых терминалах (point-of-sale, POS), производственных систем или устройств интернета вещей (internet of things, IoT) – т. е. в режиме реального времени. Эта тенденция проявилась несколько лет назад и описывается неуклюжим длинным термином *транслитеральная и гибридная обработка и анализ транзакций* (translytics and hybrid transactional and analytical processing, HTAP). На рис. 1.5 показано, как для объединения различных типов обработки применяется защищенная репликация баз данных.

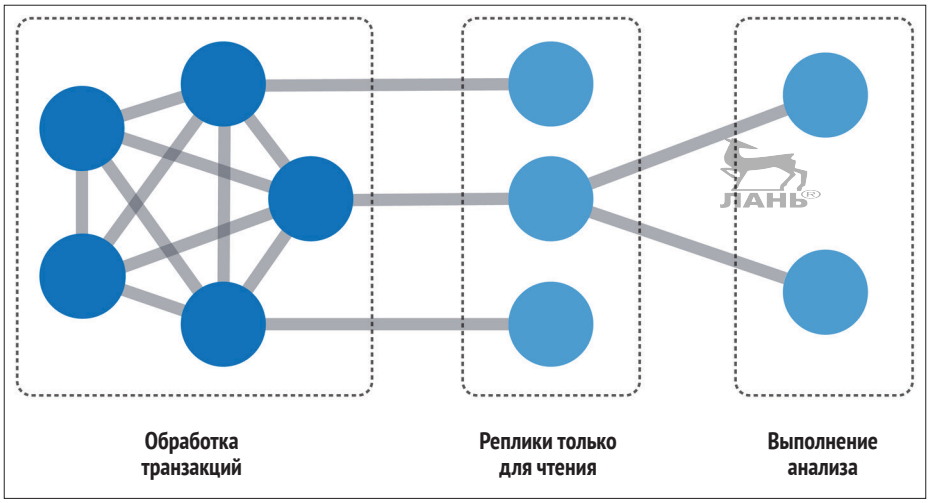


Рис. 1.5. Гибридная платформа поддерживает обработку запросов с низкой задержкой и высокую целостность данных, необходимые для транзакций, и в то же время выполняет сложный анализ больших объемов данных

Как сказано в докладе исследовательской и консалтинговой компании Gartner[6]:

> *«[HTAP] потенциально может изменить способ выполнения некоторых бизнес-процессов, поскольку расширенная аналитика в реальном времени (например, планирование, прогнозирование и анализ «что, если») становится неотъемлемой частью самого процесса, а не отдельной операцией, выполняемой постфактум. Это позволит создать новые способы принятия решений в режиме реального времени. В конечном счете HTAP станет ключевой архитектурой для интеллектуальных бизнес-процессов».*

Поскольку OLTP и OLAP становятся более интегрированными и начинают поддерживать общие функции, больше нет нужды использовать разные форматы данных или разные системы для наших рабочих задач – мы

[6] https://www.gartner.com/imagesrv/media-products/pdf/Kx/KX-1-3CZ44RH.pdf

можем упростить нашу архитектуру, используя одну и ту же платформу для того и другого. Это означает, что наши аналитические запросы могут получать данные в реальном времени, а мы сможем упростить итеративный процесс анализа.

Почему мы должны изучать графовые алгоритмы?

Графовые алгоритмы применяются для углубленного понимания связанных данных. Мы видим отношения в различных реальных системах – от взаимодействия белков до социальных сетей, от систем связи до электросетей, от розничных продаж до планирования миссий на Марсе. Понимание сетей и связей внутри них несет в себе невероятный потенциал для открытий и инноваций.

Графовые алгоритмы уникально подходят для изучения структур и выявления шаблонов в наборах данных, которые тесно связаны между собой. Нет ничего более очевидного, чем большие данные, снабженные наглядными связями. Каждую секунду во всем мире собирается, объединяется и динамически обновляется ошеломляющий объем информации. Именно графовые алгоритмы помогают разобраться в гигантских объемах связанных данных с помощью более сложного анализа, использующего отношения и – что особенно важно сегодня – улучшают контекстную информацию для искусственного интеллекта.

Поскольку наши данные становятся все более связанными, нам важно вовремя понимать их взаимосвязи и взаимозависимости. Ученые, изучающие рост сетей, отмечают, что со временем вероятность подключения увеличивается, но не равномерно. Одной из теорий о том, как динамика роста влияет на структуру сети, является механизм *предпочтительного присоединения* (preferential attachment). Эта идея, показанная на рис. 1.6, описывает склонность узла сети связываться с теми узлами, которые уже имеют много соединений[7].

В своей книге[8] «Синхронизация: как порядок возникает из хаоса во Вселенной, природе и повседневной жизни» Стивен Строгац приводит примеры и объясняет различные способы самоорганизации реальных систем. Независимо от объяснения причин многие исследователи считают, что развитие сетей неотделимо от их конечных форм и иерархий. Сети передачи данных склонны к образованию очень плотных групп, причем сложность сети растет вместе с размером данных. Сегодня мы наблюдаем такую кластеризацию связей в большинстве реальных сетей, от интернета до социальных сетей, таких как игровое сообщество (рис. 1.7).

---

[7] Наглядный пример – нарастающее число новых подписчиков у знаменитых блогеров. – *Прим. перев.*

[8] «Sync: How Order Emerges from Chaos in the Universe, Nature, and Daily Life», *Steven Strogatz*, *Hachette*.

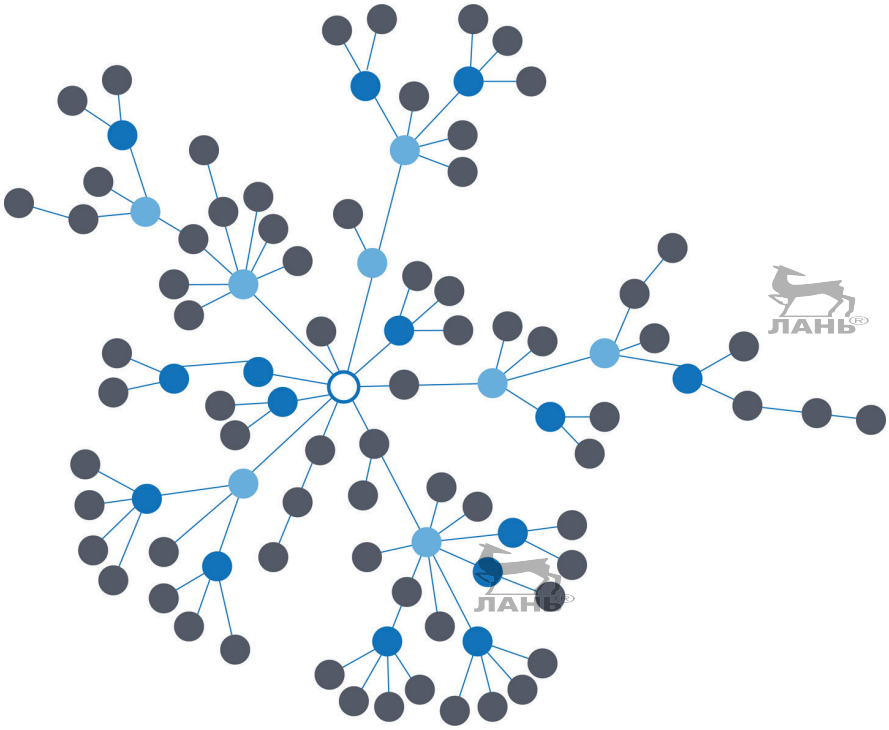


Рис. 1.6. Предпочтительное присоединение – это явление, когда чем больше связан узел сети, тем больше вероятность того, что он получит новые связи. Это приводит к возникновению неравномерности концентрации и выраженных кластеров

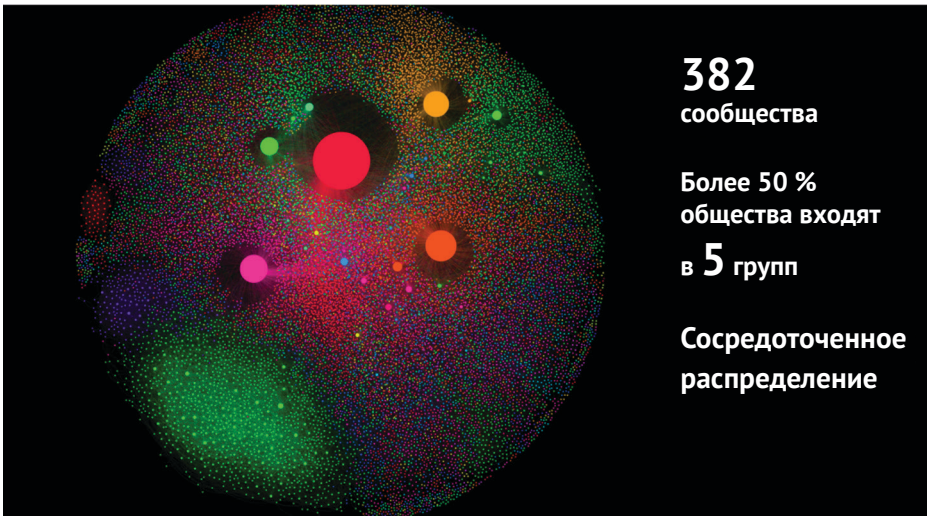


Рис. 1.7. Этот анализ игрового сообщества показывает наибольшую концентрацию связей лишь у пяти из 382 сообществ

Анализ сетевого сообщества, проиллюстрированный на рис. 1.7, был выполнен Франческо Д'Орацио из компании Pulsar с целью изучить виральность контента и стратегии распространения информации. Д'Орацио обнаружил корреляцию между распределением концентрации связей и скоростью распространения фрагмента контента в сообществе.

Эта картина разительно отличается от того, что предсказывает модель среднего распределения, где большинство узлов имеет приблизительно одинаковое количество связей. Например, если бы Всемирная паутина подчинялась среднему распределению связей, у большинства страниц было бы примерно одинаковое количество входящих и исходящих ссылок. Модели среднего распределения утверждают, что большинство узлов одинаково связаны, но многие типы графов и реальные сети демонстрируют кластеризацию. Интернет, так же как и транспортные маршруты или социальные отношения, описывается степенным распределением, т. е. несколько узлов сети буквально облеплены связями, а остальные узлы довольствуются небольшим количеством связей.

Степенной закон

Степенной закон (power law, также называемый *законом подобия*) описывает отношения между двумя величинами, когда одна величина изменяется как степень другой. Например, площадь куба связана с длиной его сторон степенью 3. Хорошо известным примером является распределение Парето, или «правило 80/20», первоначально использовавшееся для описания ситуации, когда 20 % населения контролирует 80 % богатства. Мы встречаем различные степенные законы в мире природы и сетях.

Попытка «усреднить» сеть, как правило, неприемлема для исследования взаимосвязей или прогнозирования, поскольку реальным сетям присуще неравномерное распределение узлов и взаимосвязей. Более того, эта неравномерность сама по себе несет важную информацию. На рис. 1.8 мы можем наблюдать, как использование среднего значения характеристик для данных, которые являются неравномерными, приведет к неверным результатам.

Поскольку сильно связанные данные не соответствуют среднему распределению, сетевые ученые используют анализ графов для поиска и интерпретации структур и распределений отношений в реальных данных.

> *В природе не существует известных нам сетей, которые можно было бы описать случайной моделью.*
>
> Альберт-Ласло Барабаши,
> директор Центра по изучению сложных сетей при Северо-Восточном университете, автор многочисленных книг в области науки о сетях.

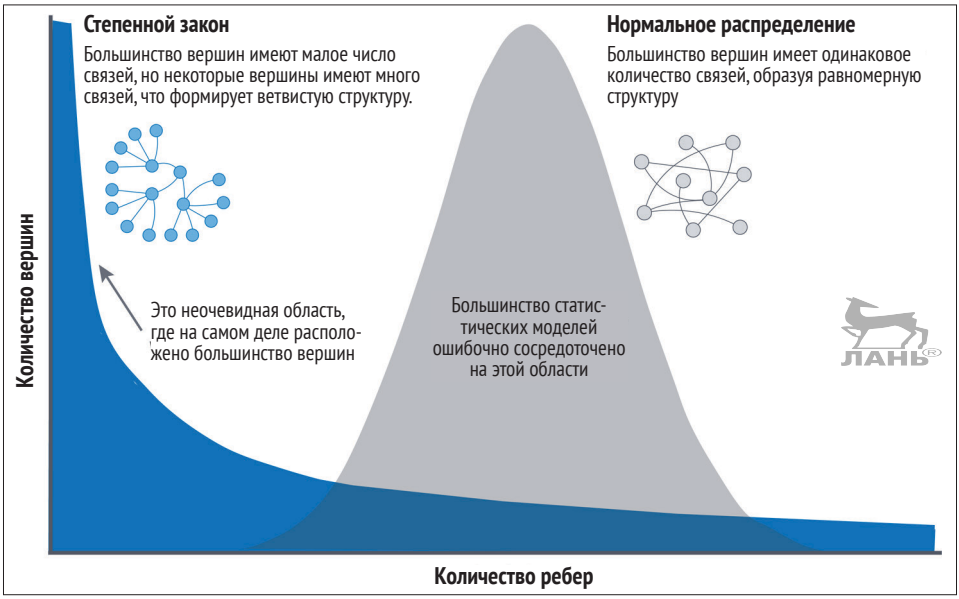


Рис. 1.8. Реальные сети имеют неравномерное распределение узлов и связей, представленных в крайнем случае степенным распределением. Среднее распределение предполагает, что большинство узлов имеет приблизительно одинаковое количество связей

Проблема для большинства исследователей заключается в том, что данные с плотными и неравномерными связями трудно анализировать с помощью традиционных инструментов. В данных наверняка есть какая-то структура, но ее трудно найти. Заманчиво использовать усредненный подход к запутанным данным, но это спрячет закономерности, а результаты вашего анализа будут отражать что угодно, только не реальные группы. Например, если вы усредните демографическую информацию обо всех ваших клиентах и предложите опыт, основанный исключительно на средних показателях, вы гарантированно пропустите большинство сообществ! Ведь сообщества, как правило, объединяются через связывающие факторы, такие как возраст и род занятий, семейное положение и место проживания.

Кроме того, с помощью моментального снимка данных практически невозможно отследить динамическое поведение, особенно в отношении внезапных событий и всплесков. Например, если вы рассматриваете социальную группу с растущими отношениями, вполне естественно, что ее члены будут интенсивно общаться между собой. Это может привести к тому, что наступит переломный этап общения, который приведет к образованию прочной коалиции или, наоборот, к формированию и поляризации подгрупп, например на выборах. Для прогнозирования развития сетей применяются весьма сложные методы, но мы можем сделать вывод

о возможном поведении, если будем видеть структуры и взаимодействия в наших данных. В данном случае анализ графов может спрогнозировать устойчивость социальной группы, поскольку сосредоточен именно на отношениях.

Где и когда применяется анализ графов?

На самом абстрактном уровне анализ графов применяется для прогнозирования поведения и предписания действий для динамических групп. Для этого требуется понимание отношений и структуры внутри группы. Графовые алгоритмы достигают этого понимания путем изучения общей природы сетей через их соединения. Благодаря такому подходу вы в результате сможете понять топологию связанных систем и смоделировать их процессы.

Существует три основных блока вопросов, от которых зависит, уместно ли использование анализа графов и графовых алгоритмов в вашей задаче (рис. 1.9).

Рис. 1.9. Типы вопросов, на которые отвечает анализ графов

Итак, вот несколько типов задач, в которых используются графовые алгоритмы. Похожи ли на них ваши рабочие задачи?

- изучение путей распространения заболевания или каскадного транспортного сбоя;
- выявление наиболее уязвимых или вредоносных компонентов при сетевой атаке;
- определение наименее затратного или самого быстрого способа маршрутизации информации или ресурсов;
- предсказание недостающие связей в ваших данных;
- выявление прямого и косвенного влияния в сложной системе;
- обнаружение невидимых иерархий и зависимостей;
- прогнозирование, будут ли группы объединяться или распадаться;

- поиск перегруженных или недогруженных узлов в сетях;
- выявление сообщества на основе поведения и создание персональных рекомендаций;
- уменьшение количества ложных срабатываний при обнаружении мошенничества и аномалий;
- извлечение дополнительных признаков для машинного обучения.

Заключение

В этой главе мы говорили о том, что современные данные, как правило, образуют сильно связанные структуры. В научной среде для анализа групповой динамики и взаимоотношений давно применяются надежные инструменты, однако подобные решения не всегда являются обычным делом в бизнесе. Оценивая передовые методы анализа, мы должны учитывать природу наших данных и хорошо понимать атрибуты сообщества при прогнозировании сложного поведения. Если наши данные представляют собой сеть, нам следует избегать соблазна снизить число параметров путем усреднения. Вместо этого мы должны использовать правильные инструменты, которые соответствуют нашим данным и ожиданиям от анализа.

В следующей главе мы рассмотрим связанные с графами понятия и термины.

Глава 2

Теория и концепции графов

В этой главе мы определяем область исследований и раскрываем терминологию теории графов и графовых алгоритмов. В объяснении основных понятий теории графов сделан акцент на определениях, которые наиболее актуальны для практикующего специалиста.

Мы рассмотрим способы представления графов, а затем изучим различные типы графов и их атрибуты. Это пригодится позже, так как характеристики исследуемого графа будут определять выбор алгоритма и помогут нам интерпретировать результаты. Мы закончим главу обзором типов графовых алгоритмов, подробно описанных в этой книге.

Терминология

Помеченный граф (labeled graph) является одним из самых популярных способов моделирования связанных данных.

Метка отмечает вершину как часть группы. На рис. 2.1 у нас есть две группы узлов: `человек` и `автомобиль`. Хотя в классической теории графов метка применяется к одной вершине, в настоящее время она обычно используется для обозначения группы вершин. Отношения классифицируются исходя из *типа отношения* (relationship type). Наш пример на рис. 2.1 включает типы отношений `ВОДИТ`, `ИМЕЕТ`, `ЖИВЕТ_С` и `ЖЕНАТ_НА`.

Свойства (property) являются синонимами *атрибутов* (attribute) и могут содержать данные разных типов – от чисел и строк до пространственных и временных данных. На рис. 2.1 мы присваиваем свойства в виде пар `имя:значение`, где сначала указывается имя свойства, а затем его значение. Например, вершина `человек` слева имеет свойство `имя:Дэн`, а отношение `ЖЕНАТ_НА` имеет свойство `начало:01.01.2013`.

Подграф (subgraph) – это граф внутри большего графа. Подграфы полезны в качестве фильтров, например когда нам нужно извлечь подмножество с конкретными характеристиками для целенаправленного анализа.

Путь (path) – это группа вершин и их взаимосвязей. Пример простого пути на основе графа, изображенного на рис. 2.1, может содержать вершины `Дэн`, `Анна` и `автомобиль` и ребра `ВОДИТ` и `ИМЕЕТ`.

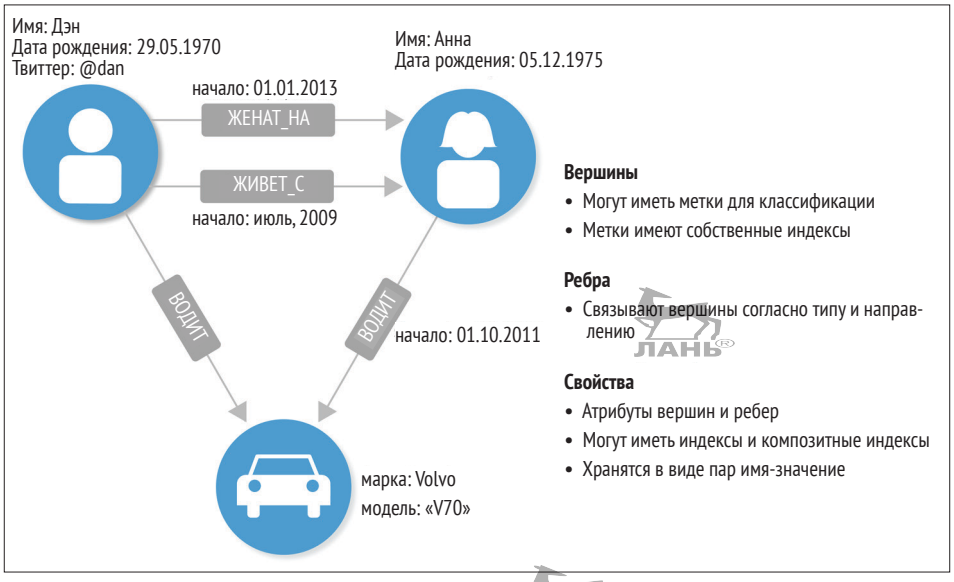


Рис. 2.1. Модель помеченного графа и его свойств представляет собой гибкий и лаконичный способ представления связанных данных

Графы различаются по типу, форме и размеру, а также по различным атрибутам, которые можно использовать для анализа. Далее мы опишем виды графов, наиболее подходящие для графовых алгоритмов. Имейте в виду, что эти объяснения применимы как к графам, так и к подграфам.

Типы и структуры графов

В классической теории графов термин *граф* приравнивается к *простому* (simple), или *строгому* (strict), графу, где две вершины связывает только одно отношение, как показано в левой части рис. 2.2. Однако большинство графов реального мира имеют несколько связей между вершинами и даже связи, замкнутые на себя. Сегодня этот термин обычно используется для всех трех типов графов на рис. 2.2, поэтому мы тоже будем использовать расширенное толкование.

Случайные, локализованные и безмасштабные сети

Графы принимают разные формы. На рис. 2.3 изображены три наиболее типичных разновидности сети:

- *случайная* (random) – при полностью равномерном распределении связей образуется случайная сеть без иерархий. Отражающий такую сеть бесформенный граф является «плоским» и не имеет выраженных паттернов. Все вершины имеют одинаковую вероятность присоединения к любой другой вершине;

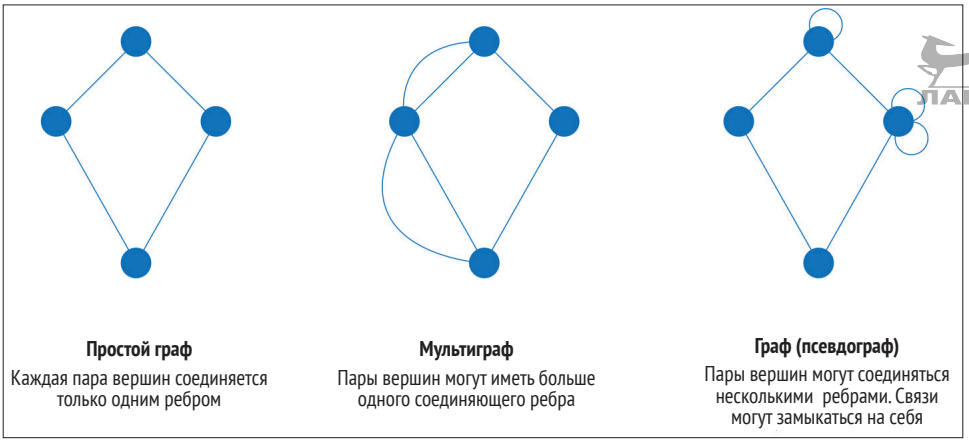


Рис. 2.2. В этой книге мы используем термин *граф* для любого из этих классических типов графов

Рис. 2.3. Три сетевые структуры с характерными графами и поведением

- *локализованная*[1], или *сеть small-world*, – эта разновидность чрезвычайно распространена в социальных сетях; она имеет ограниченную структуру и выраженный паттерн в виде *центрально-лучевой архитектуры* (hub-and-spoke architecture). Самым известным примером локализованной сети является знаменитая теория шести рукопожатий. Хотя вы общаетесь в основном с небольшой группой друзей, для связи с любым другим человеком достаточно цепочки из нескольких звеньев, даже если он известный актер или живет на другой стороне планеты;
- *безмасштабная* (scale-free) – подобная сеть возникает при наличии степенных распределений и сохранении центрально-лучевой архитектуры независимо от масштаба, как, например, во Всемирной сети World Wide Web.

[1] В русском языке нет общепринятого перевода и часто употребляется исходный термин *small-world*. – *Прим. перев.*

На основе перечисленных в списке типов сетей создают графы с выраженными структурами, распределениями и поведением. Работая с графовыми алгоритмами в следующих главах книги, вы обнаружите в результатах уже знакомые из этой главы признаки и паттерны.

Разновидности графов

Чтобы получить максимальную отдачу от графовых алгоритмов, важно иметь представление о наиболее характерных признаках графов, с которыми вы столкнетесь. В табл. 2.1 сведены ключевые обобщенные признаки графов. В следующих разделах мы рассмотрим эти разновидности графов более подробно.

Таблица 2.1. Обобщенные разновидности графов

Признак	Ключевой фактор	Особенности алгоритмов
Связные и несвязные	Существует ли путь между любыми двумя вершинами графа независимо от расстояния	Острова могут вызывать неожиданное поведение, например «застревание» алгоритма или сбой при обработке отключенных компонентов
Взвешенные и невзвешенные	Существуют ли весовые значения, присвоенные ребрам или вершинам	Многие алгоритмы учитывают веса, и мы увидим существенные различия в результатах, если их игнорировать
Направленные и ненаправленные	Определяют ли связи в явном виде начальную и конечную вершину	Это свойство предоставляет алгоритму простор для выявления дополнительных смыслов. В некоторых алгоритмах вы можете явно задать направление в одну сторону, в обе стороны или отсутствие направленности
Циклические и ациклические	Граф циклический, если путь обхода завершается на исходной вершине	Циклические графы распространены, но алгоритмы должны быть аккуратны (как правило, за счет сохранения состояния обхода), иначе алгоритм может работать бесконечно. Ациклические графы (или каркасы) также являются основой для многих графовых алгоритмов
Плотные и разреженные	Отношение количества вершин к количеству ребер	Чрезвычайно плотные или крайне разреженные графы могут привести к различным результатам
Однодольные, двудольные и k-дольные	Зависит от того, связана ли вершина только с одним типом других вершин (например, пользователям нравятся фильмы) или с несколькими типами других вершин (например, пользователям нравятся пользователи, которым нравятся фильмы)	Полезно для создания отношений, помогающих анализировать и проектировать более полезные графы

Связные и несвязные графы

Граф называется *связным* (connected), если между всеми вершинами есть ребра (связи). Если на графе есть острова, он *несвязный* (disconnected). Если узлы на этих островах связаны, они называются *компонентами* (или *кластерами*), как показано на рис. 2.4 слева.

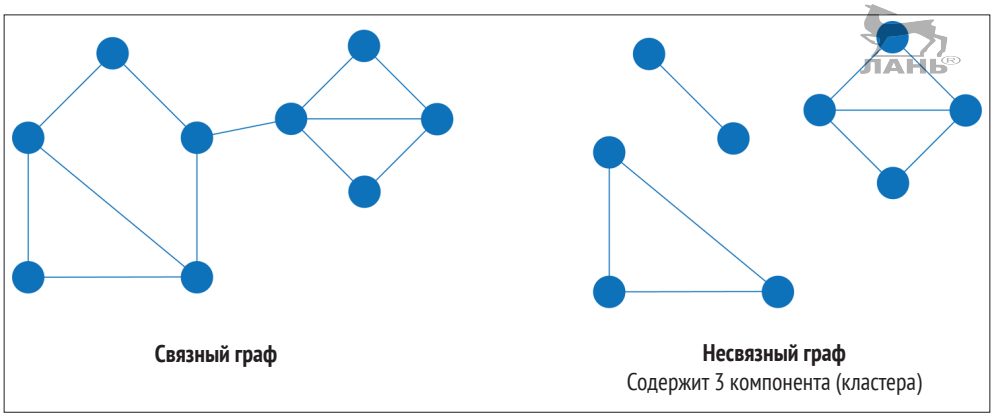


Рис. 2.4. Если в графе есть острова, это несвязный граф

Некоторые алгоритмы плохо работают с несвязными графами и могут давать ошибочные результаты. Если вы получили неожиданные результаты, то первым делом тщательно проверьте структуру графа на связность.

Невзвешенные и взвешенные графы

Невзвешенные (unweighted) графы не имеют весовых значений, назначенных их узлам или отношениям. Для *взвешенных* (weighted) графов эти значения могут представлять различные показатели, такие как стоимость, время, расстояние, пропускная способность или даже приоритетность конкретной области. Разница проиллюстрирована на рис. 2.5.

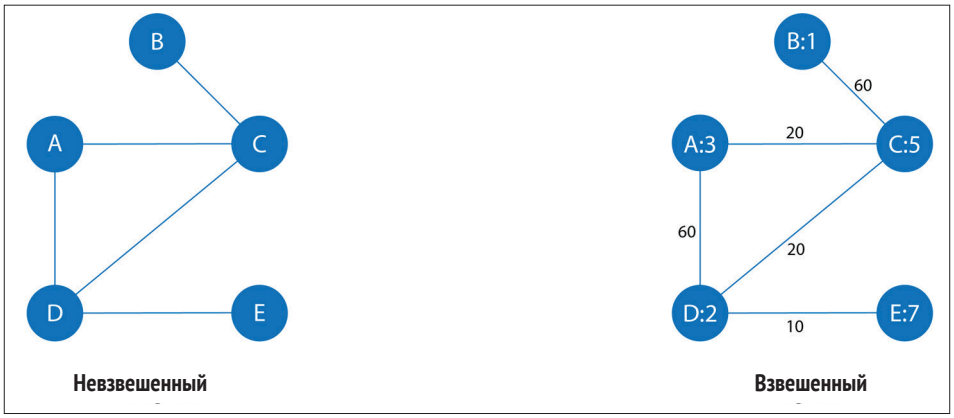


Рис. 2.5. Ребрам и вершинам взвешенного графа могут быть присвоены значения

Основные графовые алгоритмы могут использовать весовые коэффициенты в качестве представления силы или значения отношений. Многие алгоритмы вычисляют метрики, которые затем применяются в качестве весов при последующей обработке. Некоторые алгоритмы обновляют значения весов по мере анализа, чтобы найти кумулятивные итоги, минимальные значения или оптимумы.

Классическим применением взвешенных графов являются алгоритмы поиска пути. Такие алгоритмы лежат в основе картографических приложений на наших телефонах и вычисляют самые короткие, самые дешевые или самые быстрые транспортные маршруты между точками. Например, на рис. 2.6 используются два разных метода вычисления кратчайшего пути.

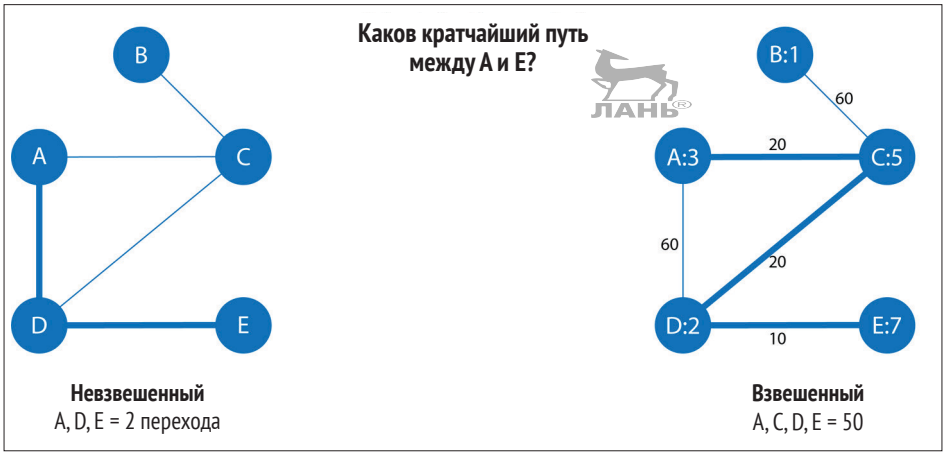


Рис. 2.6. При использовании невзвешенных и взвешенных графов кратчайшие пути могут различаться

Без учета весов наш кратчайший маршрут рассчитывается исходя из количества ребер (обычно называемых *переходами* или *прыжками*). *A* и *E* имеют кратчайший путь из двух переходов, между которыми расположена только одна промежуточная вершина *D*. Однако кратчайший *взвешенный путь* от *A* до *E* переносит нас через вершины *C* и *D*. Если веса представляют собой физическое расстояние в километрах, общее расстояние составит 50 км. В этом случае самый короткий путь с точки зрения количества переходов будет соответствовать более длинному физическому маршруту – 70 км.

Ненаправленные и ориентированные графы

В *ненаправленном* (undirected) графе отношения считаются двусторонними (например, дружеские отношения). В *ориентированном*[2] (directed) графе, часто называемом *орграфом* (digraph), связи имеют определенное

[2] Направленный граф исторически принято называть ориентированным или орграфом, а обычный граф по умолчанию считается ненаправленным. – *Прим. перев.*

направление. Связь, направленная к данной вершине извне, называется *входящей связью* (in-link), а направленная наружу, к другой вершине, – *исходящей* (out-link).

Направление добавляет еще одно измерение информации. Связи одного типа, но в противоположных направлениях, имеют разное семантическое значение, выражая зависимость или обозначая поток. Направление можно использовать в качестве индикатора доверия или влияния группы. Личные предпочтения и социальные отношения очень хорошо обозначаются направлением.

Например, если мы предположим, что ориентированный граф на рис. 2.7 – это сеть студентов, а направления соответствуют «лайкам», то без труда вычислим, что студенты *A* и *C* более популярны.

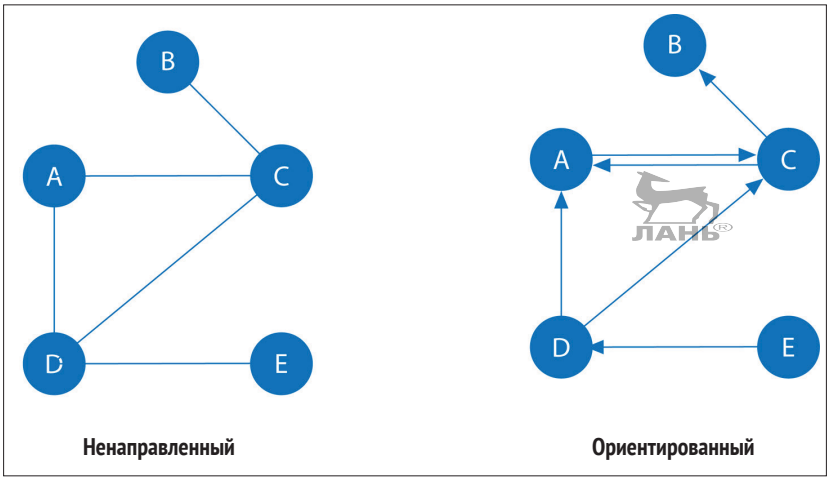


Рис. 2.7. Многие алгоритмы позволяют нам выполнять анализ с учетом направления связей между вершинами

Дорожные сети демонстрируют пример одновременного использования двух типов связей. Например, по автомагистрали между городами можно ехать в обоих направлениях. Однако внутри городов по отдельным улицам разрешено только одностороннее движение. Сказанное верно и для некоторых информационных потоков!

Выполняя алгоритм для ориентированного и ненаправленного графа, мы получим разные результаты. Если направление не указано в явном виде – например, для шоссе или дружеских отношений, – мы предполагаем, что все связи двусторонние.

Если мы переосмыслим рис. 2.7 как направленную дорожную сеть, вы можете доехать до *A* из *C* и *D*, но из пункта *C* вы можете только выехать. Кроме того, при отсутствии исходящей связи от *A* до *C* в пункте *A* получился бы тупик. Подобная ситуация маловероятна для дорожной сети, но вовсе не редкость для процесса или веб-страницы.

Ациклические и циклические графы

В теории графов *циклы* – это пути через ребра и вершины, которые начинаются и заканчиваются в одной и той же вершине. *Ациклический* (acyclic) граф не имеет таких циклов. Как показано на рис. 2.8, циклы могут присутствовать как в направленных, так и в ненаправленных графах, но в случае ориентированного графа обход цикла следует за направлением связей. *Ориентированный ациклический граф* (directed acyclic graph, DAG), отмеченный как Граф 1 на рис. 2.8, по определению всегда будет иметь тупики (также называемые *конечными вершинами* или *листьями*).

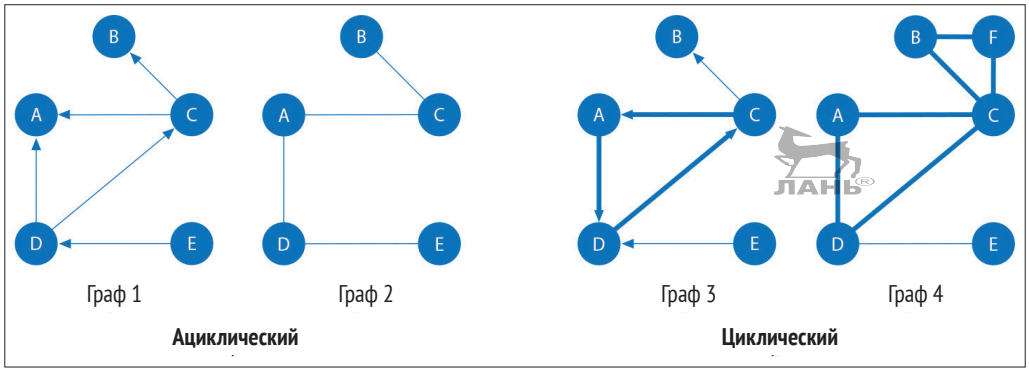


Рис. 2.8. В ациклических графах невозможно начинать и заканчивать на одном и том же узле, не повторяя наши шаги

Графы 1 и 2 не имеют циклов, так как невозможно начать и закончить в одной и той же вершине, не проходя по одному ребру дважды. В предыдущей главе мы говорили, что на однократном прохождении основана проблема семи мостов Кенигсберга, положившая начало теории графов! Граф 3 на рис. 2.8 иллюстрирует простой цикл *A-D-C-A* без повторений. Граф 4 демонстрирует более интересный ненаправленный циклический граф. В нем есть замкнутый цикл с повторяющимся узлом C и маршрутом *B-F-C-D-A-C-B*. Фактически Граф 4 содержит несколько циклов.

Иногда нам приходится преобразовывать циклические графы в ациклические путем сокращения ребер, чтобы устранить проблемы обработки. Ориентированные ациклические графы естественным образом возникают в планировании, генеалогии и истории версий программных продуктов.

Деревья

В классической теории графов *ненаправленный ациклический граф* называется *деревом*. Однако в информатике деревья также могут иметь направления. Более общим определением дерева будет граф, в котором любые два узла связаны только одним путем. Деревья важны для понимания гра-

фовых структур и многих алгоритмов. Они играют ключевую роль в проектировании сетей, структур данных и поисковой оптимизации для улучшения категоризации или иерархии.

Деревьям и их вариациям посвящено множество публикаций. Рисунок 2.9 иллюстрирует наиболее общие разновидности деревьев, с которыми вы можете столкнуться.

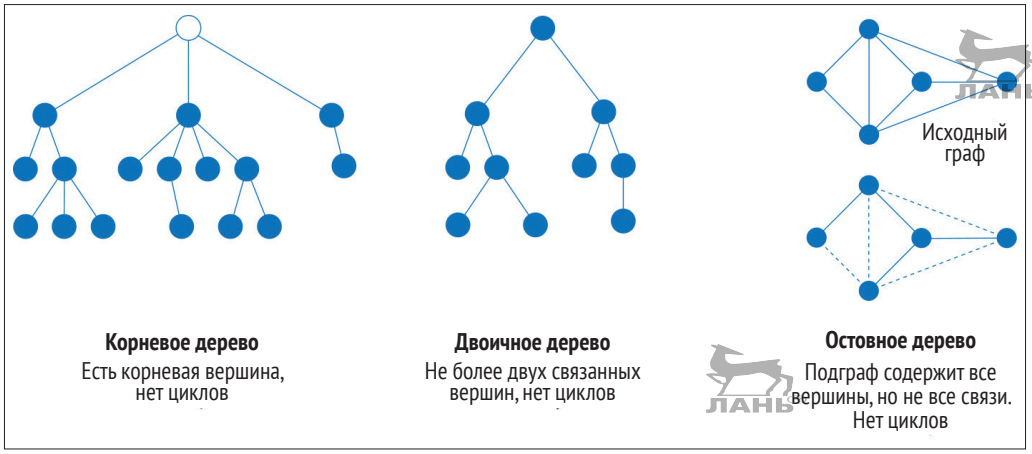


Рис. 2.9. В графовых алгоритмах чаще всего используются остовные деревья (каркасы)

Из упомянутых вариаций деревьев наиболее актуальными для нашей книги является *остовное дерево* (spannig tree), или *каркас*, – подграф, который включает в себя все вершины большого ациклического графа, но не обязательно все его ребра. *Минимальное остовное дерево* соединяет все узлы графа либо с наименьшим количеством переходов, либо с наименьшими взвешенными путями.

Разреженные и плотные графы

Разреженность графа основана на количестве ребер, которые он содержит по отношению к максимально возможному числу ребер, если бы все вершины соединялись друг с другом. Граф, в котором каждая вершина связана с любой другой вершиной, называется *полным графом* (complete graph) или *кликой* (clique). Например, если бы все ваши друзья дружили друг с другом, это была бы клика. Максимальная плотность графа – это число возможных ребер в полном графе. Она рассчитывается по формуле $\max D = \frac{N(N-1)}{2}$, где N – количество вершин. Для вычисления фактической плотности мы используем формулу $D = \frac{2(R)}{N(N-1)}$, где R – количество ребер. На рис. 2.10 показаны примеры трех показателей фактической плотности для ненаправленных графов.

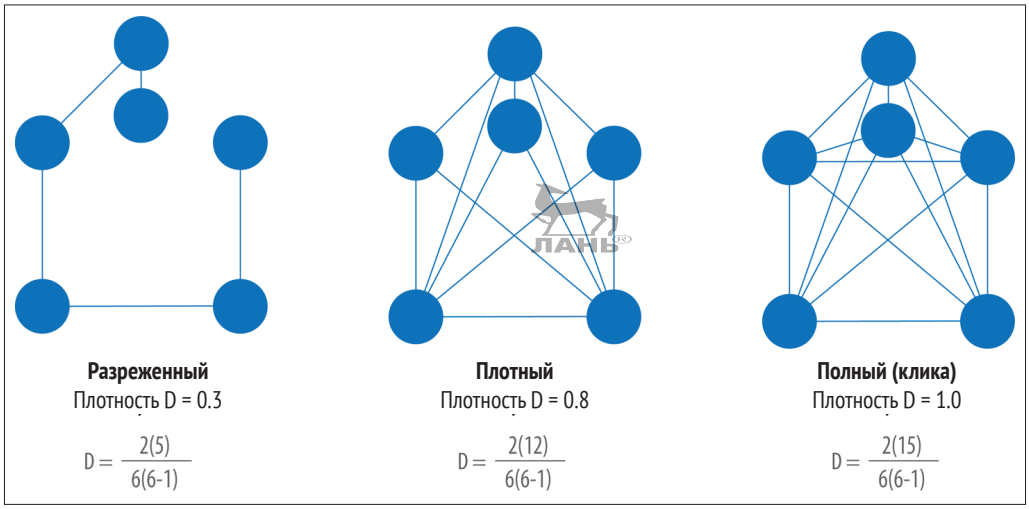


Рис. 2.10. Проверка плотности графа помогает избежать непредвиденных результатов

Хотя строгой границы не существует, любой граф с фактической плотностью, приближающейся к максимальной плотности, считается *плотным* (dense). Большинство графов, основанных на реальных сетях, имеют тенденцию к разреженности с приблизительно линейной корреляцией между количеством вершин и количеством ребер. Это особенно актуально, когда в игру вступают физические элементы, такие как практические ограничения на количество проводов, труб, дорог или дружеских отношений, к которым вы можете присоединиться в одной точке.

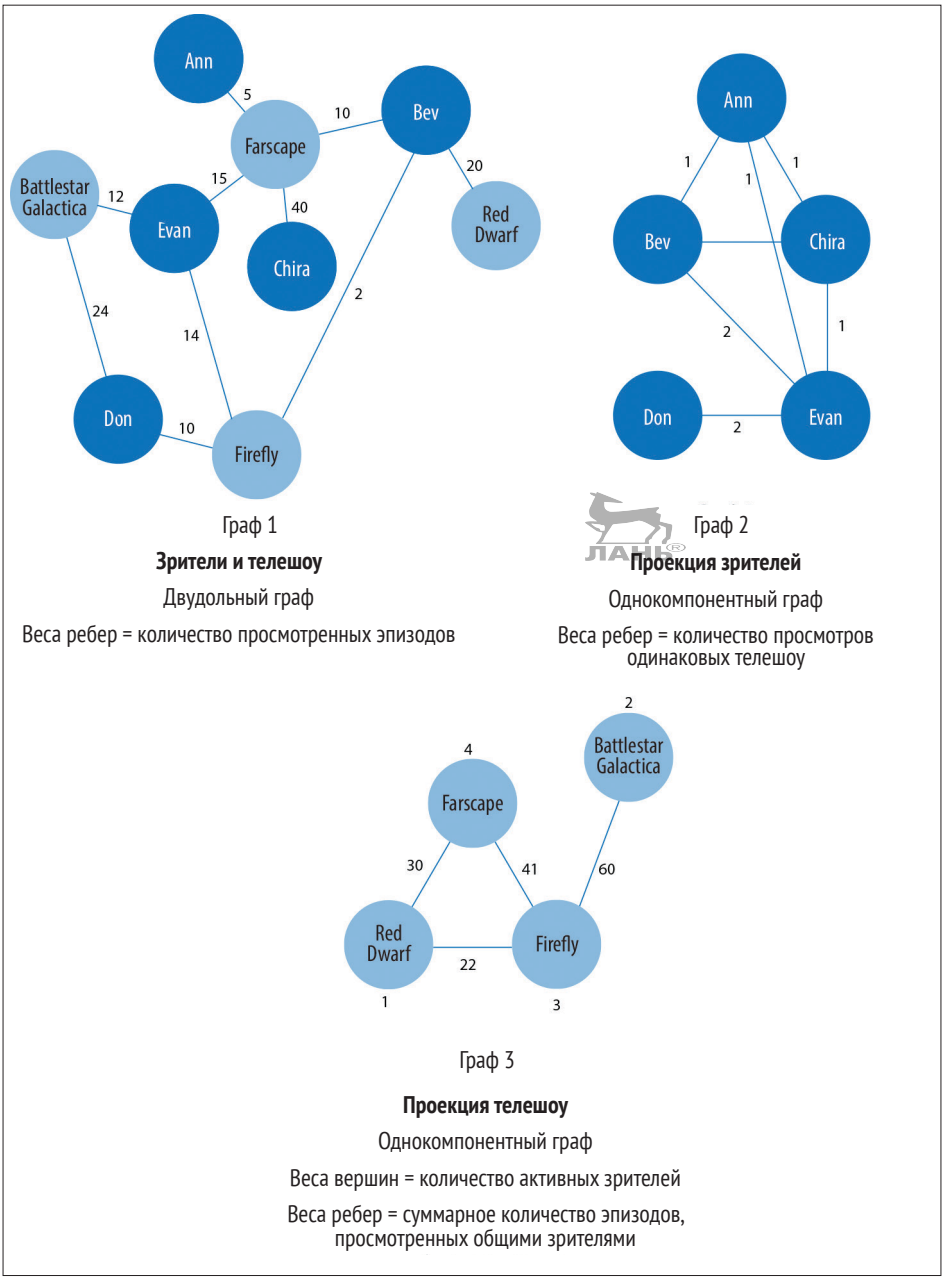
Некоторые алгоритмы при выполнении на очень разреженных или плотных графах будут возвращать бессмысленные результаты. Если граф слишком разреженный, то для вычисления полезных результатов алгоритму может не хватить связей. С другой стороны, вершины с очень высокой плотностью связей не добавляют много дополнительной информации. Высокая плотность может также исказить некоторые результаты или увеличить вычислительную сложность. В этих ситуациях хорошим практическим решением станет фильтрация соответствующего подграфа.

Однокомпонентные, двудольные и *k*-дольные графы

Большинство сетей фактически содержит данные с несколькими типами вершин и связей. Однако графовые алгоритмы часто учитывают только один тип вершины и один тип связи. Графы, отражающие один тип вершин и один тип связи, иногда называют *однокомпонентными* или *однодольными* (monopartite).

Двудольный (bipartite) граф – это граф, вершины которого можно разделить на два набора, так что связи соединяют только вершину из одного набора с вершиной из другого набора. На рис. 2.11 изображен пример та-

кого графа. У него есть два набора вершин – зрители и телешоу. Связи установлены только между наборами, а внутри наборов связей нет. Другими словами, на Графе 1 телешоу связано только со зрителями, но не с другим телешоу, и зрители также напрямую не связаны с другими зрителями.



Граф 1

Зрители и телешоу

Двудольный граф

Веса ребер = количество просмотренных эпизодов

Граф 2

Проекция зрителей

Однокомпонентный граф

Веса ребер = количество просмотров одинаковых телешоу

Граф 3

Проекция телешоу

Однокомпонентный граф

Веса вершин = количество активных зрителей

Веса ребер = суммарное количество эпизодов, просмотренных общими зрителями

Рис. 2.11. Двудольные графы часто проецируются в однокомпонентные графы для более детального анализа

Исходя из нашего двудольного графа зрителей и телешоу, мы создали две однокомпонентные *проекции* (projection) – Граф 2 показывает зрителей, которые смотрят одинаковые шоу, а Граф 3 отражает телевизионные шоу, связанные общими зрителями. Мы также можем фильтровать данные по типу отношений – например, составить проекции зрителей, которые просто посмотрели, поставили оценку или написали отзыв.

Построение проекций однокомпонентных графов с предполагаемыми связями является важной частью анализа графов. Такие проекции помогают выявить косвенные отношения и свойства. Например, на Графе 2 на рис. 2.11 зрители Бев и Энн смотрели только одно общее телешоу, тогда как у Бев и Эвана есть два общих шоу. На Графе 3 мы оценили взаимосвязь между телешоу с точки зрения общих зрителей. Эта или другие метрики, такие как сходство, могут использоваться для определения прочности связи между такими действиями, как просмотр шоу Battlestar Galactica и Firefly. Знание о наличии и прочности связи поможет нам создать рекомендации следующего просмотра для кого-то, похожего на Эвана, который на рис. 2.11 только что закончил смотреть последний эпизод Firefly.

В свою очередь, *k-дольные* (*k*-partite) графы содержат столько типов вершин, сколько их есть в данных (k). Например, если у нас есть три типа вершин, у нас будет трехдольный граф. Это просто расширение понятия однокомпонентных и двудольных графов на произвольное количество типов вершин. Многие графы реального мира, особенно графы знаний, имеют большое значение *k*, поскольку они объединяют много разных концепций и типов информации. Примером использования большого числа типов вершин является создание новых рецептов салата путем сопоставления набора рецептов с набором ингредиентов, а затем составление новых смесей на основе популярных связей. Мы также можем попытаться уменьшить количество типов вершин путем обобщения, например рассматривая такие типы, как «петрушка» и «руккола», как просто «листовую зелень».

Теперь, когда мы рассмотрели типы графов, с которыми вам, скорее всего, придется работать, давайте поговорим о типах графовых алгоритмов, которые вы будете применять к этим графам.

Типы графовых алгоритмов

Давайте рассмотрим три области анализа, которые лежат в основе графовых алгоритмов. Эти категории соответствуют следующим главам книги об алгоритмах поиска пути, вычисления центральности и обнаружения сообществ.

Поиск пути

Пути являются основополагающим понятием в анализе графов и графовых алгоритмов, поэтому мы начнем наши будущие главы с конкрет-

ных примеров алгоритмов. Поиск кратчайших путей, вероятно, является наиболее частой задачей, решаемой с помощью графовых алгоритмов, и является предшественником различных типов анализа. *Кратчайший путь* – это маршрут движения с наименьшим количеством переходов или самым низким суммарным весом ребер. Если граф ориентированный, то это самый короткий маршрут между двумя узлами с учетом разрешенных направлений.

Типы путей

Средний кратчайший путь (average shortest path) используется для оценки общей эффективности и отказоустойчивости сетей, например для понимания среднего расстояния между станциями метро. Иногда необходимо найти самый длинный оптимизированный маршрут для таких ситуаций, как выявление станций метро, расположенных наиболее далеко друг от друга или имеющих наибольшее количество остановок между ними, даже когда выбран лучший маршрут. В этом случае мы используем *диаметр* графа, чтобы найти самый длинный кратчайший путь между всеми парами вершин.

Определение центральности

Центральность – это представление о том, какие узлы важнее в сети. Но что мы подразумеваем под важностью? Существуют различные типы алгоритмов центральности, созданных для измерения разных свойств, таких как способность быстро распространять информацию. В этой книге мы сосредоточимся на том, как структурированы вершины и отношения.

Обнаружение сообщества

Связанность – это основная концепция теории графов, которая позволяет проводить сложный сетевой анализ, например поиск сообществ. Большинство реальных сетей имеют подструктуры (часто квазифрактальные) из более или менее независимых подграфов.

Для поиска сообществ и количественной оценки степени сгруппированности можно применить анализ существующих связей. Изучение различных типов сообществ в графе может выявить в нем наличие структур, таких как концентраторы и иерархии, а также склонность групп привлекать или отталкивать новых членов. Эти методы используются для изучения возникающих социальных явлений, таких как *эффект эхокамеры* в медийном пространстве или *эффект пузырькового фильтра*.

Заключение

Графы интуитивно понятны. Они соответствуют тому, как мы думаем и рисуем системы. Стоит лишь усвоить базовые термины и определения – и можно приступать к изучению основных методов работы с графами. В этой главе мы объяснили идеи и понятия, которые будем использовать на протяжении всей книги, и описали разновидности графов, с которыми вы столкнетесь.

Дополнительное чтение по теории графов

Если вы хотите узнать больше о самой теории графов, мы рекомендуем несколько вводных книг:

- «Введение в теорию графов»[3], *Ричард Дж. Трюдо*, – очень хорошо написанное, доступное и понятное введение;
- «Введение в теорию графов»[4], пятое издание, *Робин Дж. Уилсон*, – это серьезное введение с хорошими иллюстрациями;
- «Теория графов и ее приложения»[5], третье издание, *Джонатан Л. Гросс*, *Джей Йеллен* и *Марк Андерсон*, – эта книга предполагает более глубокие знания математики и содержит больше деталей и упражнений.

Далее мы рассмотрим виды обработки и анализа графов, а затем перейдем к реализации графовых алгоритмов в Apache Spark и Neo4j.

[3] Introduction to Graph Theory, by *Richard J. Trudeau* (Dover).

[4] Introduction to Graph Theory, Fifth Ed., by *Robin J. Wilson* (Pearson).

[5] Graph Theory and Its Applications, Third Ed., by *Jonathan L. Gross*, *Jay Yellen*, and *Mark Anderson* .(Chapman and Hall)

Глава 3

Графовые платформы и обработка

В этой главе мы бегло рассмотрим различные методы обработки графов и наиболее распространенные вычислительные графовые платформы. Затем более подробно остановимся на двух платформах, задействованных в этой книге (Apache Spark и Neo4j), и поясним, в каких ситуациях их следует применять. Благодаря наличию инструкций по установке вы полностью подготовитесь к работе с материалом следующих глав.

Графовые платформы и особенности обработки

Аналитической обработке графов присущи уникальные особенности, такие как вычисления, основанные на структуре, необходимость глобального охвата объекта анализа и трудность парсинга данных. Далее мы рассмотрим общие соображения относительно графовых платформ и обработки.

Подходы к выбору платформы

Не утихают споры о том, в каком направлении развивать обработку графов. Стоит ли использовать мощные локальные многоядерные машины с большой памятью и сосредоточиться на эффективных структурах данных и многопоточных алгоритмах? Или стоит инвестировать в структуры распределенной обработки и соответствующие алгоритмы?

Полезный подход к оценке – это метод «конфигурации, превосходящей однопоточную» (Configuration that Outperforms a Single Thread, COST), описанный в исследовательской работе[1] «Масштабируемость! Но какой ценой?» Ф. Макшерри, М. Изарда и Д. Мюррея. COST дает нам возможность сопоставить масштабируемость системы с накладными расходами, которые она вносит. Основная идея заключается в том, что хорошо сконфигурированная система, использующая оптимизированный алгоритм

[1] «Scalability! But at What COST?» by *F. McSherry*, *M. Isard*, and *D. Murray*. В названии статьи игра слов: COST по-английски означает «затраты». – *Прим. перев.*

и структуру данных, может превзойти текущие универсальные решения, основанные на горизонтальном масштабировании. Это метод измерения прироста производительности, достигаемого без затрат на системы, маскирующие свою неэффективность за счет распараллеливания. Разделение идей масштабируемости и эффективного использования ресурсов поможет нам создать платформу, специально настроенную для наших нужд.

Некоторые подходы к графовым платформам включают высоко интегрированные решения, которые оптимизируют и координируют алгоритмы, предварительную обработку и извлечение данных из памяти.

Подходы к обработке данных

Существуют разные подходы к обработке данных – например, потоковая или пакетная обработка, или парадигма распараллеливания с сокращением (map-reduce). Однако при обработке графовых данных можно также использовать специальные методы, основанные на зависимостях данных, присущих только графовым структурам:

ориентированные на вершины (node-centric).

Этот подход использует вершины в качестве модулей обработки, заставляя их накапливать и вычислять состояние и сообщать об изменениях состояния своим соседям. Обычно такая модель использует загружаемые функции преобразования для более простой реализации каждого алгоритма;

ориентированные на ребра (relationship-centric).

Этот подход имеет сходство с моделью, ориентированной на вершины, но может работать лучше с подграфами и при последовательном анализе;

графо-ориентированные (graph-centric).

Такие модели обрабатывают вершины внутри подграфа независимо от других подграфов, в то время как происходит минимальный обмен сообщениями между подграфами;

маршрутно-ориентированные (traversal-centric).

Эти модели используют накопление данных при обходе графа в качестве средства вычисления;

алгоритмически-ориентированные (algorithm-centric).

Это гибрид предыдущих моделей, в котором используют различные методы или их комбинацию для оптимальной реализации заданного алгоритма.

Pregel – это ориентированная на вершины графа отказоустойчивая система параллельной обработки, созданная Google для эффективного анализа больших графов. Pregel основана на модели *массово-синхронного параллелизма* (bulk synchronous parallel, BSP) и упрощает параллельное программирование благодаря наличию отдельных фаз вычисления и обмена данными.

Pregel накладывает ориентированную на вершины абстракцию поверх BSP. Таким образом, алгоритмы вычисляют значения из входящих сообщений от каждого соседа вершины. Эти вычисления выполняются один раз за итерацию и могут обновлять значения вершин и отправлять сообщения другим вершинам. Вершины также могут объединять передаваемые сообщения, что уменьшает количество переговоров в сети. Алгоритм завершается, когда больше нечего передавать или достигнут установленный лимит.

Большинство из этих специальных подходов требуют доступа ко всему графу для выполнения кросс-топологических операций. Это связано с тем, что разделение графа на части приводит к интенсивной передаче данных и частым переключениям между рабочими экземплярами. Подобный подход не совсем уместен для многих алгоритмов, которым требуется итеративно обрабатывать глобальную структуру графа.

Объединяющие платформы

Существуют несколько платформ, соответствующих особенностям алгоритмов обработки графов. Традиционно было разделение между графовыми вычислениями и графовыми базами данных, что вынуждало пользователей перемещать свои данные в зависимости от потребностей процесса:

графовые вычисления,

нетранзакционные механизмы только для чтения, которые сосредоточены на эффективном выполнении итеративного анализа и запросов всего графа. Механизмы вычисления графа поддерживают различные парадигмы описания и выполнения графовых алгоритмов, такие как ориентированные на вершины (например, Pregel, Gather-Apply-Scatter) или основанные на MapReduce (например, PACT). Примерами графовых вычислительных движков являются Giraph, GraphLab, Graph-Engine и Apache Spark;

графовые базы данных

основаны на транзакциях и сосредоточены на быстрых операциях записи и чтения с использованием небольших запросов, которые

обычно затрагивают небольшую часть графа. Их сильные стороны заключаются в надежности работы и высокой масштабируемости при конкурентном доступе многих пользователей.

Выбор платформы

Выбор рабочей платформы опирается на множество факторов, таких как тип выполняемого анализа, требования к производительности, существующая среда и предпочтения команды разработчиков. В этой книге для демонстрации графовых алгоритмов мы используем платформы Apache Spark и Neo4j, потому что они предлагают уникальные преимущества.

Spark – пример масштабируемого графового вычислительного движка, ориентированного на вершины. Его популярная вычислительная среда и библиотеки поддерживают различные процессы, применяемые в науке о данных. Spark является подходящей платформой, если:

- алгоритмы являются принципиально распараллеливаемыми или разделяемыми;
- для рабочих процессов алгоритма требуются «многоязычные» операции на нескольких инструментах и языках;
- анализ может работать автономно в пакетном режиме;
- анализ графа выполняется на данных, не преобразованных в графовый формат;
- команда нуждается в реализации собственных алгоритмов и обладает таким опытом;
- команда редко использует графовые алгоритмы;
- для хранения данных и анализа команда предпочитает использовать экосистему Hadoop.

Графовая платформа Neo4j является примером тесной интеграции графовой базы данных и алгоритмически-ориентированной обработки, оптимизированной для графов. Она популярна при разработке приложений на основе графов и включает в себя библиотеку графовых алгоритмов, настроенную для собственной графовой базы данных. Neo4j является подходящей платформой, если:

- алгоритмы более итеративны и требуют хорошей доступности памяти;
- алгоритмы и результаты чувствительны к производительности;
- анализ графа выполняется на сложных графовых данных и/или требует глубокого обхода;
- анализ или результаты связаны с высокими транзакционными нагрузками;

- результаты используются для обогащения существующего графа;
- команде нужна интеграция с инструментами графовой визуализации;
- команда предпочитает предварительно упакованные и поддерживаемые алгоритмы.

Наконец, некоторые организации одновременно используют Neo4j и Spark для обработки графов: Spark – для высокоуровневой фильтрации, предварительной обработки массивных наборов и объединения данных, а Neo4j – для более специфической обработки и интеграции с графовыми приложениями.

Apache Spark

Apache Spark (далее просто Spark) – это аналитический движок для крупномасштабной обработки данных. Он использует табличную абстракцию, называемую DataFrame, для представления и обработки данных в строках именованных и типизированных столбцов. Платформа интегрирует различные источники данных и поддерживает такие языки, как Scala, Python и R, а также различные аналитические библиотеки (рис. 3.1). Эта основанная на памяти система работает с использованием эффективно распределенных вычислительных графов.

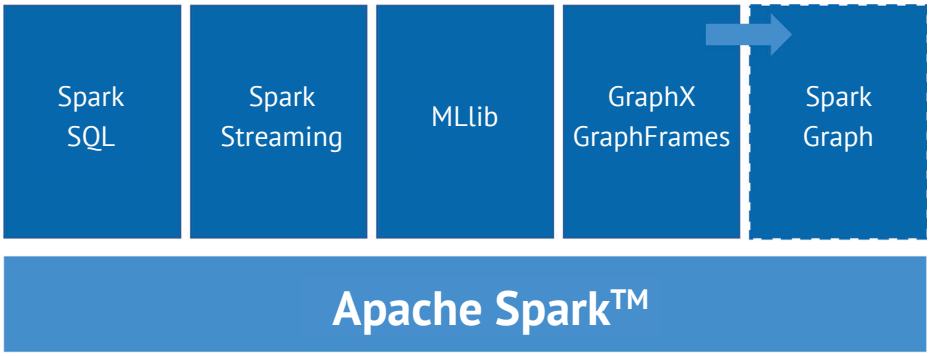


Рис. 3.1. Spark – это распределенная и универсальная инфраструктура кластерных вычислений с открытым исходным кодом. Она включает в себя несколько модулей для различных рабочих задач

GraphFrames – это библиотека обработки графов для Spark, сменившая GraphX в 2016 году, хотя она и отделена от ядра Apache Spark. Библиотека GraphFrames основана на GraphX, но использует объекты DataFrame в качестве базовой структуры данных. GraphFrames поддерживает языки программирования Java, Scala и Python. Весной 2019 года был принят рабочий документ[2] «Spark Graph: графы свойств, запросы Cypher и алгоритмы»

[2] «Spark Graph: Property Graphs, Cypher Queries, and Algorithms».

(см. врезку «Эволюция проекта Spark Graph»). Мы ожидаем, что реализация документа принесет в основной проект Spark новую функциональность с использованием инфраструктуры DataFrame и языка запросов Cypher. Тем не менее в этой книге наши примеры будут основаны на Python API (PySpark) из-за его текущей популярности среди аналитиков данных Spark.

Эволюция проекта Spark Graph

Проект Spark Graph – это совместная инициатива участников проекта Apache из Databricks и Neo4j по реализации поддержки DataFrame, Cypher и алгоритмов на основе DataFrame в основном проекте Apache Spark в рамках версии 3.0.

Cypher начинался как декларативный язык графовых запросов, реализованный в Neo4j, но благодаря проекту openCypher он теперь используется многими поставщиками баз данных и проектом с открытым исходным кодом Cypher для Apache Spark (CAPS).

В самое ближайшее время мы с нетерпением ожидаем использования CAPS для загрузки и проецирования графовых данных в качестве составной части платформы Spark. Мы опубликуем примеры использования Cypher после реализации проекта Spark Graph.

Новая разработка не повлияет на работоспособность алгоритмов, описанных в этой книге, но может добавить новые параметры для вызова процедур. Базовая модель данных, концепции и вычисления графовых алгоритмов останутся прежними.

Вершины и ребра графа представлены в виде DataFrame с уникальным идентификатором для каждой вершины и указанием начальной и конечной вершины для каждого отношения. Мы можем увидеть пример описания вершины DataFrame в табл. 3.1 и описания ребра DataFrame в табл. 3.2.

Таблица 3.1. Пример таблицы вершин

id	city	state
JFK	New York	NY
SEA	Seattle	WA

Таблица 3.2. Пример таблицы ребер

src	dst	delay	tripId
JFK	SEA	45	1058923

Граф GraphFrame, основанный на DataFrame из этого примера, будет иметь две вершины, JFK и SEA, и одно направленное ребро от JFK до SEA.

Таблица DataFrame для вершин должна иметь столбец `id` – значение в этом столбце используется для уникальной идентификации каждой вершины. Ребра в таблице DataFrame должны иметь столбцы `src` и `dst` – значения в этих столбцах описывают, какие вершины связаны ребром, и должны ссылаться на записи в столбце `id` таблицы вершин DataFrame.

Вершины и ребра DataFrame могут быть загружены с использованием любого из источников данных, совместимых с DataFrame, включая Parquet, JSON и CSV. Запросы описываются с использованием комбинации PySpark API и Spark SQL.

API GraphFrame предоставляет пользователям точку подключения[3] для реализации алгоритмов, которые не доступны из коробки.

Установка Spark

Вы можете скачать Spark с веб-сайта Apache Spark по адресу http://spark.apache.org/downloads.html. Сразу после установки Spark вам нужно установить следующие библиотеки для выполнения заданий Spark из Python:

```
pip install pyspark graphframes
```

Затем вы можете запустить pyspark REPL, выполнив следующую команду:

```
export SPARK_VERSION="spark-2.4.0-bin-hadoop2.7"
./${SPARK_VERSION}/bin/pyspark \
  --driver-memory 2g \
  --executor-memory 6g \
  --packages graphframes:graphframes:0.7.0-spark2.4-s_2.11
```

На момент написания этой книги последней выпущенной версией Spark была spark-2.4.0-bin-hadoop2.7, но она могла измениться к тому времени, как вы приступите к работе[4]. Если это так, обязательно измените переменную среды `SPARK_VERSION` соответствующим образом.

Хотя задания Spark должны выполняться на кластере машин, в демонстрационных целях мы собираемся выполнять задания только на одной машине. Вы можете узнать больше о работе Spark в реальных производственных условиях в книге Билла Чамберса и Матея Захарии «Spark: The Definitive Guide: Big Data Processing Made Simple» (O'Reilly).

На русском языке в 2015 году была опубликована книга «Изучаем Spark. Молниеносный анализ данных», *Матей Захария, Холден Карау и др. – Прим. перев.*

Теперь вы готовы научиться запускать графовые алгоритмы в Spark.

Графовая платформа Neo4j

Графовая платформа Neo4j поддерживает транзакционную и аналитическую обработку графовых данных. Она включает в себя хранение графов

[3] http://graphframes.github.io/graphframes/docs/_site/user-guide.html#message-passing-via-aggregatemessages.

[4] На момент подготовки перевода была выпущена версия 2.4.4 – *Прим. перев.*

и вычисления с помощью инструментов анализа и управления данными. Набор встроенных инструментов расположен поверх общего протокола, API и языка запросов Cypher, тем самым обеспечивая эффективный доступ для различных приложений, как показано на рис. 3.2.

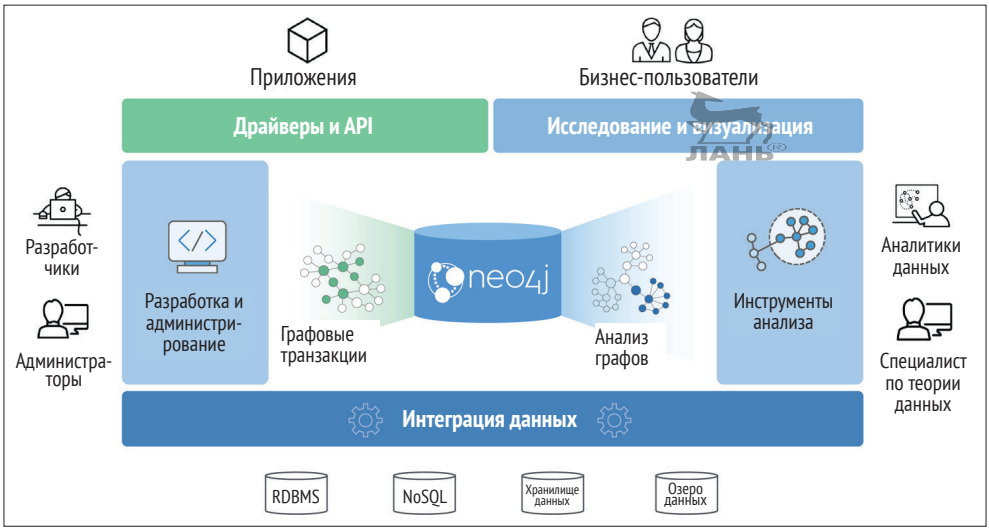


Рис. 3.2. Графовая платформа Neo4j построена на основе собственной графовой базы данных, которая поддерживает транзакционные приложения и анализ графов

В этой книге мы будем использовать библиотеку Neo4j Graph Algorithms. Библиотека устанавливается как плагин вместе с базой данных и предоставляет набор пользовательских процедур, которые можно выполнять с помощью языка запросов Cypher.

В этой книге мы также будем использовать библиотеку Neo4j под названием Awesome Procedures on Cypher (APOC). Библиотека APOC состоит из более чем 450 процедур и функций, предназначенных для решения общих задач, таких как интеграция данных, преобразование данных и рефакторинг модели.

Библиотека графовых алгоритмов содержит параллельные версии алгоритмов, реализующих анализ графов и рабочие процессы машинного обучения. Алгоритмы выполняются поверх задачно-ориентированной платформы параллельных вычислений и оптимизированы для платформы Neo4j. Внутренние представления графов разных размеров масштабируются до десятков миллиардов вершин и ребер.

Результаты передаются клиенту в виде потока кортежей, а результаты в табличной форме можно использовать в качестве исходной таблицы для

дальнейшей обработки. Результаты также могут быть записаны обратно в базу данных как свойства вершин или ребер.

Установка Neo4j

Neo4j Desktop – удобный инструмент для работы с локальными базами данных Neo4j. Его можно скачать с сайта Neo4j по адресу https://neo4j.com/download/. После установки и запуска Neo4j Desktop вы сможете установить подключаемые модули графовых алгоритмов и библиотеки APOC. В левом меню создайте проект и выберите его. Затем нажмите на кнопку **Manage** (Управление) в базе данных, в которую вы хотите установить плагины. На вкладке **Plugins** (Плагины) вы увидите опции для нескольких плагинов. Нажмите кнопку **Install** (Установить) для плагинов графовых алгоритмов и APOC, как показано на рис. 3.3 и 3.4.

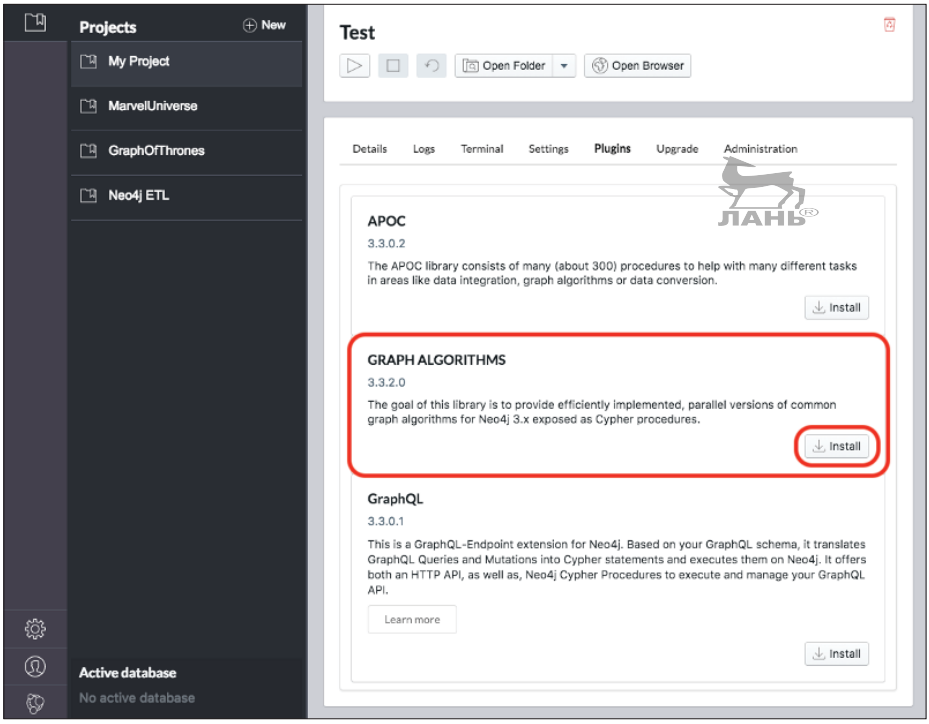


Рис. 3.3. Установка библиотеки графовых алгоритмов

За более подробными инструкциями вы можете обратиться к блогу Дженнифер Рейф «Исследуйте новые миры, добавляя плагины в Neo4j» по адресу https://bit.ly/2TU0Lj3. Теперь вы готовы к запуску графовых алгоритмов на платформе Neo4j.

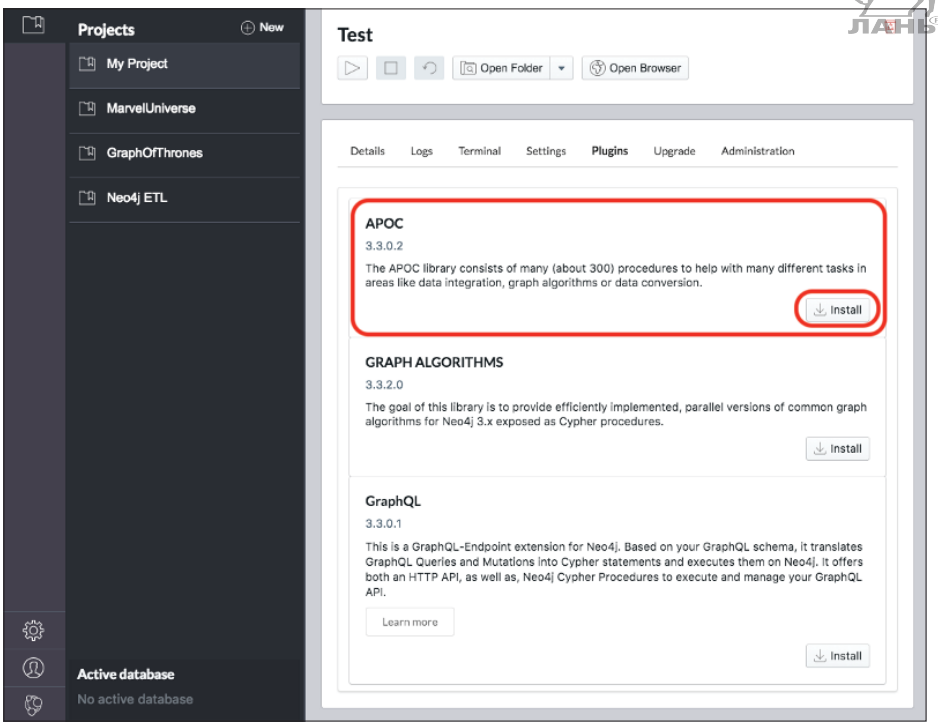


Рис. 3.4. Установка библиотеки APOC

Заключение

В предыдущих главах мы пояснили, почему анализ графов так важен для изучения реальных сетей, и рассмотрели фундаментальные концепции графов, анализ и обработку. Теперь у вас есть прочная основа для дальнейшего изучения графовых алгоритмов. В следующих главах вы узнаете, как запускать примеры графовых алгоритмов на платформах Spark и Neo4j.

Глава 4

Алгоритмы поиска по графу и поиска пути

Алгоритмы поиска по графу (graph search algorithm) исследуют граф либо для изучения его структуры, либо для поиска конкретной информации. Эти алгоритмы прокладывают пути через граф, но не стоит ожидать, что такие пути всегда будут вычислительно оптимальными. Первым делом вы познакомитесь с алгоритмами поиска в ширину и поиска в глубину, потому что они имеют основополагающее значение для обхода графа и часто являются обязательным первым шагом для многих других видов анализа.

Алгоритмы поиска пути (pathfinding algorithms) основываются на алгоритмах поиска по графу и исследуют маршруты между вершинами, начиная с одной вершины и проходя через ребра, пока не будет достигнут пункт назначения. Эти алгоритмы используются для определения оптимальных маршрутов в различных приложениях, таких как логистические планировщики, уменьшение стоимости звонков или оптимальная IP-маршрутизация, а также игровая симуляция.

В частности, в этой книге мы рассмотрим следующие алгоритмы поиска пути:

- *кратчайший путь* с двумя полезными вариантами (A* и Йен) – нахождение кратчайшего пути или путей между двумя выбранными вершинами;
- *кратчайший путь для всех пар* и *кратчайший путь из одного источника* – нахождение кратчайших путей между всеми парами или от выбранной вершины ко всем остальным;
- *минимальное остовное дерево* – нахождение связанной древовидной структуры с наименьшей стоимостью посещения всех вершин из заданной вершины;
- *случайное блуждание* – это полезный шаг предварительной обработки/выборки для процессов машинного обучения и других графовых алгоритмов.

В этой главе мы объясним, как работают упомянутые алгоритмы, и покажем примеры их реализации в Spark и Neo4j. В тех случаях, когда алгоритм доступен только на одной платформе, мы приведем только один пример или объясним, как вы можете настроить нашу реализацию под свои задачи.

На рис. 4.1 показаны ключевые различия между основными типами алгоритмов, а в табл. 4.1 приведена краткая справка о том, что делает каждый алгоритм, и примеры применения.

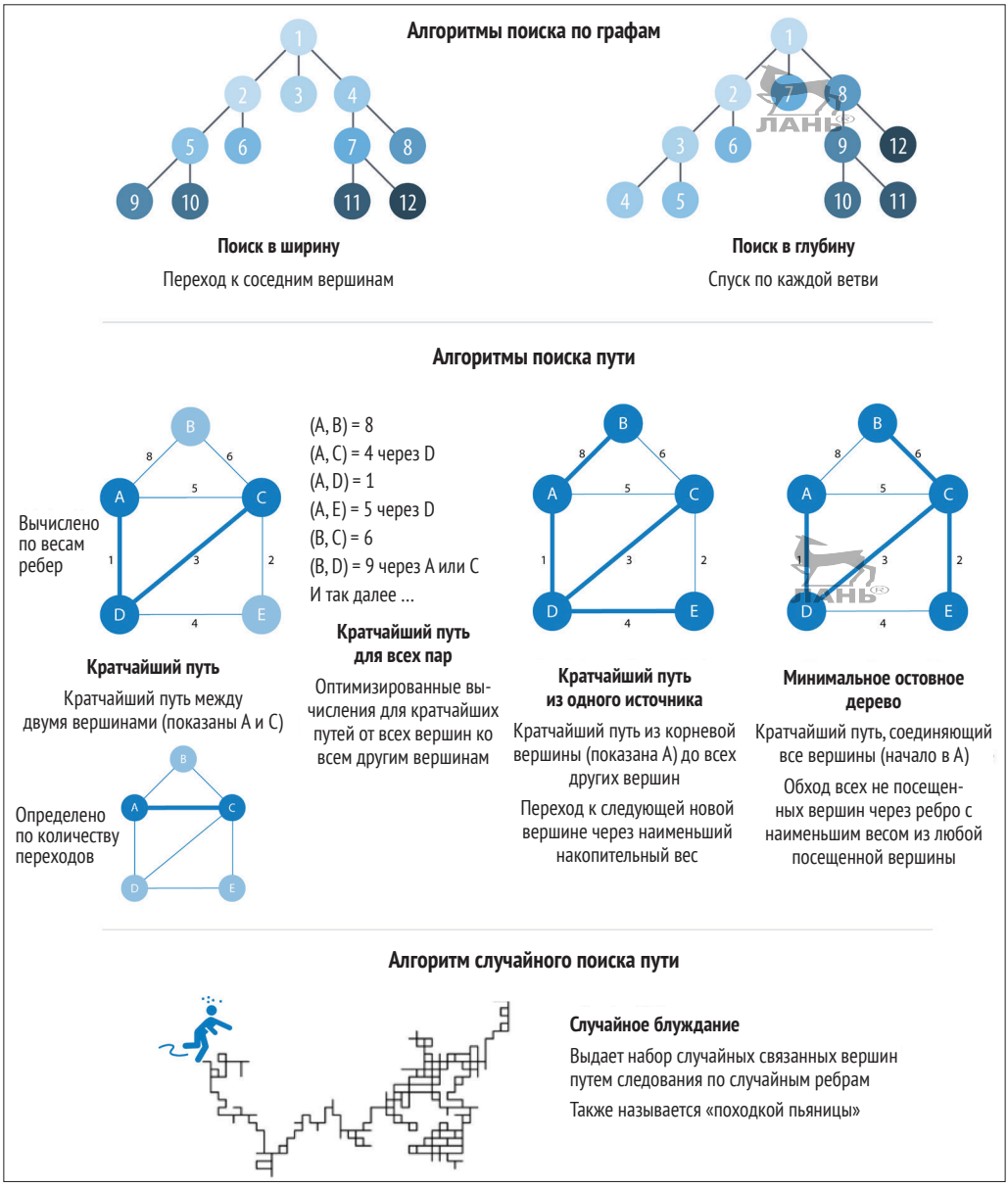


Рис. 4.1. Алгоритмы нахождения пути и поисковые алгоритмы

Таблица 4.1. Обзор алгоритмов нахождения пути и поисковых алгоритмов

Тип алгоритма	Что он делает	Пример применения	Пример Spark	Пример Neo4j
Поиск в ширину	Обходит древовидную структуру, разворачиваясь, чтобы исследовать ближайших соседей, а затем их соседей на следующем подуровне	Нахождение соседних вершин в навигаторах GPS для определения близлежащих достопримечательностей	Да	Нет
Поиск в глубину	Обходит древовидную структуру, спускаясь как можно дальше вниз по каждой ветви перед возвратом	Поиск оптимального пути решения в игровых симуляциях с иерархическим выбором	Нет	Нет
Вариации кратчайшего пути A* и Йен	Вычисляет кратчайший путь между парой вершин	Поиск направлений движения между двумя точками	Да	Да
Кратчайший путь между всеми парами	Вычисляет кратчайший путь между *всеми парами вершин* в графе	Оценка альтернативных маршрутов вокруг пробки	Да	Да
Кратчайший путь из одного источника	Вычисляет кратчайший путь между одной начальной вершиной и всеми остальными вершинами	Наименьшая стоимость маршрутизации телефонных звонков	Да	Да
Минимальное остовное дерево	Вычисляет путь в связанной древовидной структуре с наименьшей стоимостью посещения всех вершин	Оптимизация связанной маршрутизации, такой как прокладка кабеля или сборка мусора	Нет	Да
Случайное блуждание	Возвращает список вершин вдоль пути заданного размера путем случайного выбора ребер для прохождения	Дополнительный источник для машинного обучения или данные для графовых алгоритмов	Нет	Да

Сначала вы познакомитесь с набором данных для наших примеров и узнаете, как импортировать данные в Apache Spark и Neo4j. Главы разбиты на разделы, посвященные отдельным алгоритмам. Раздел начинается с краткого описания алгоритма и принципа его работы. Большинство разделов также содержат рекомендации о том, когда использовать соответствующие алгоритмы. Раздел завершается рабочим примером кода, использующим фрагмент реальных данных.

Давайте начнем!

Пример данных: транспортный граф

Все связанные данные содержат пути между вершинами, поэтому поиск данных и поиск пути являются отправными точками в изучении анализа графов. Наборы транспортных данных (табл. 4.2 и 4.3) иллюстрируют эти отношения в интуитивно понятном и доступном виде. Примеры в этой главе работают с графом, содержащим подмножество европейской дорожной сети. Вы можете получить файлы вершин и ребер в файловом архиве книги.

Таблица 4.2. Файл transport-nodes.csv (вершины)

id	latitude	longitude	population
Amsterdam	52.379189	4.899431	821752
Utrecht	52.092876	5.104480	334176
Den Haag	52.078663	4.288788	514861
Immingham	53.61239	–0.22219	9642
Doncaster	53.52285	–1.13116	302400
Hoek van Holland	51.9775	4.13333	9382
Felixstowe	51.96375	1.3511	23689
Ipswich	52.05917	1.15545	133384
Colchester	51.88921	0.90421	104390
London	51.509865	–0.118092	8787892
Rotterdam	51.9225	4.47917	623652
Gouda	52.01667	4.70833	70939

Таблица 4.3. Файл transport-relationships.csv (ребра)

src	dst	relationship	cost
Amsterdam	Utrecht	EROAD	46
Amsterdam	Den Haag	EROAD	59
Den Haag	Rotterdam	EROAD	26
Amsterdam	Immingham	EROAD	369
Immingham	Doncaster	EROAD	74
Doncaster	London	EROAD	277
Hoek van Holland	Den Haag	EROAD	27
Felixstowe	Hoek van Holland	EROAD	207
Ipswich	Felixstowe	EROAD	22
Colchester	Ipswich	EROAD	32
London	Colchester	EROAD	106
Gouda	Rotterdam	EROAD	25
Gouda	Utrecht	EROAD	35
Den Haag	Gouda	EROAD	32
Hoek van Holland	Rotterdam	EROAD	33

На рис. 4.2. изображен граф, который мы хотим смоделировать.

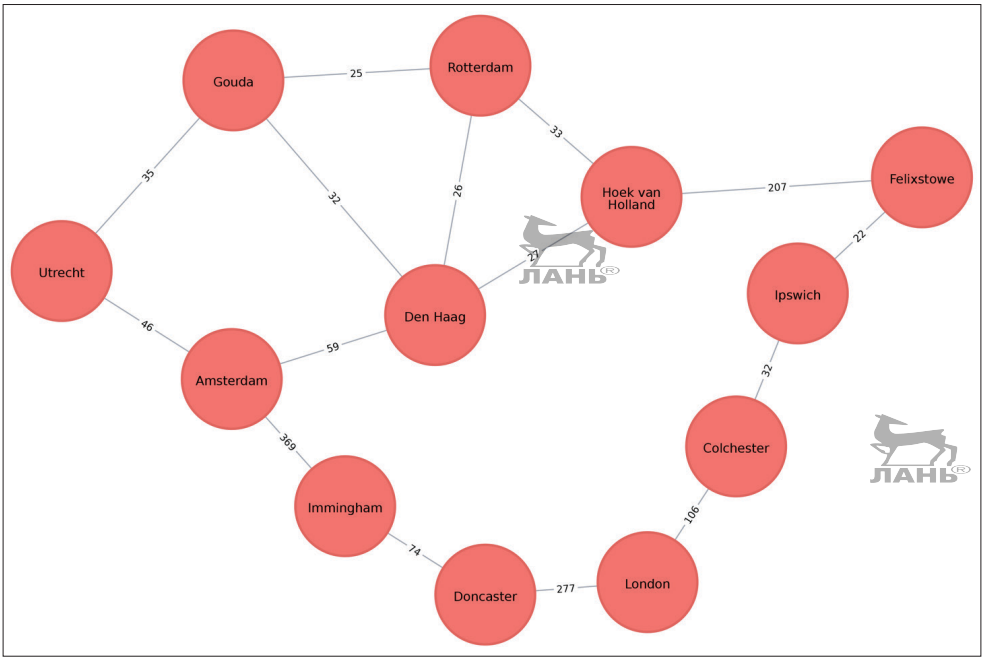


Рис. 4.2. Транспортный граф

Для простоты мы считаем граф на рис. 4.2 ненаправленным, поскольку большинство дорог между городами и в самом деле являются двунаправленными. Мы получили бы немного другие результаты, если бы сочли граф ориентированным из-за небольшого количества улиц с односторонним движением, но общий подход остается схожим. Тем не менее Spark и Neo4j работают только с ориентированными графами. Сейчас нам придется работать с ненаправленными графами (т. е. двунаправленными дорогами), поэтому применим простые хитрости:

- для Spark мы создадим две связи для каждой строки в *transport-relations.csv* – одну для перехода от `dst` к `src` и другую – из `src` в `dst`;
- для Neo4j мы создадим одну связь, а затем проигнорируем направление связи при запуске алгоритмов.

Теперь можем приступить к загрузке графов в Spark и Neo4j из файлов примеров CSV, только не забывайте про правильное представление ненаправленных связей.

Импорт данных в Apache Spark

Начиная работу с платформой Spark, сначала импортируем нужные нам пакеты из Spark и пакета GraphFrames:

```
from pyspark.sql.types import *
from graphframes import *
```

Следующая функция создает объект GraphFrame из CSV-файла примера:

```
def create_transport_graph():
    node_fields = [
        StructField("id", StringType(), True),
        StructField("latitude", FloatType(), True),
        StructField("longitude", FloatType(), True),
        StructField("population", IntegerType(), True)
    ]
    nodes = spark.read.csv("data/transport-nodes.csv", header=True,
                           schema=StructType(node_fields))

    rels = spark.read.csv("data/transport-relationships.csv", header=True)
    reversed_rels = (rels.withColumn("newSrc", rels.dst)
                     .withColumn("newDst", rels.src)
                     .drop("dst", "src")
                     .withColumnRenamed("newSrc", "src")
                     .withColumnRenamed("newDst", "dst")
                     .select("src", "dst", "relationship", "cost"))

    relationships = rels.union(reversed_rels)

    return GraphFrame(nodes, relationships)
```


Загрузка вершин проста, но для ребер придется выполнить небольшую предварительную обработку, чтобы мы могли создать каждое ребро дважды.

Теперь вызовем функцию, которую мы только что создали:

```
g = create_transport_graph()
```

Импорт данных в Neo4j

Если речь идет о Neo4j, мы начинаем с загрузки файла вершин:

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data" AS base
WITH base + "transport-nodes.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MERGE (place:Place {id:row.id})
SET place.latitude = toFloat(row.latitude),
    place.longitude = toFloat(row.latitude),
    place.population = toInteger(row.population)
```

А затем переходим к ребрам:

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
WITH base + "transport-relationships.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MATCH (origin:Place {id: row.src})
MATCH (destination:Place {id: row.dst})
MERGE (origin)-[:EROAD {distance: toInteger(row.cost)}]->(destination)
```

Хотя в данных представлены направления между вершинами, при выполнении алгоритма мы будем их игнорировать.

Поиск в ширину

Поиск в ширину (breadth first search, BFS) – один из фундаментальных алгоритмов обхода графа. Он начинается с выбранной вершины и исследует всех ее соседей на расстоянии одного перехода, а затем посещает всех соседей на расстоянии двух переходов и т. д.

Алгоритм был впервые опубликован в 1959 году Эдвардом Муром (Edward F. Moore), который использовал его, чтобы найти кратчайший путь из лабиринта. Затем на его основе в 1961 году разработали алгоритм проводной маршрутизации, предложенный в работе[1] С. Ли «Алгоритм связных путей и его приложения».

BFS чаще всего используется в качестве основы для других более целенаправленных алгоритмов. Например, алгоритмы кратчайшего пути, связанных компонентов и центральности применяют алгоритм BFS. Его также можно использовать для поиска кратчайшего пути между вершинами.

На рис. 4.3 показан порядок, в котором мы будем посещать вершины нашего транспортного графа, если бы мы выполняли поиск в ширину с началом в голландском городе Den Haag (на русском мы называем его Гаага). Цифры рядом с названием города указывают порядок посещения каждой вершины.

Сначала мы посещаем всех непосредственных соседей Гааги, затем их соседей и соседей их соседей, пока у нас не закончатся ребра, по которым можно переходить.

Поиск в ширину с помощью Apache Spark

Алгоритм поиска в ширину на платформе Spark находит кратчайший путь между двумя вершинами по количеству связей (т. е. переходов) между ними. Вы можете явно назвать вашу целевую вершину или задать критерии, которые должны быть выполнены.

Например, вы можете использовать функцию `bfs`, чтобы найти первый город среднего размера (по европейским стандартам) с населением

[1] «An Algorithm for Path Connections and Its Applications», *C. Y. Lee*, 1961.

от 100 000 до 300 000 человек. Давайте сначала проверим, в каких местах население соответствует этим критериям:

```
(g.vertices
 .filter("population > 100000 and population < 300000")
 .sort(«population»)
 .show())
```

Мы получим следующий вывод:

id	latitude	longitude	population
Colchester	51.88921	0.90421	104390
Ipswich	Ipswich	1.15545	133384

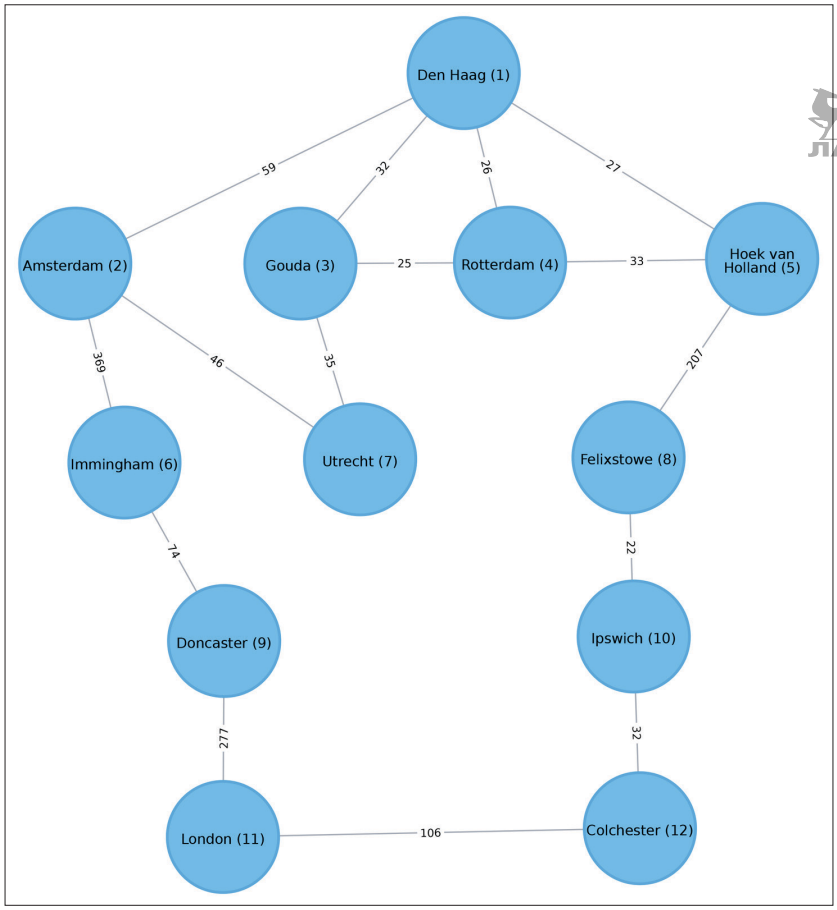


Рис. 4.3. Поиск в ширину, начиная с Гааги. Номера вершин означают порядок обхода

Как видите, есть только два места, которые соответствуют нашим критериям, – города Ипсвич (Ipswich) и Колчестер (Colchester), и можно ожидать, что на основании поиска в ширину мы сначала доберемся до Ипсвича.

Следующий код находит кратчайший путь от Гааги до города среднего размера:

```
from_expr = "id='Den Haag'"
to_expr = "population > 100000 and population < 300000 and id <> 'Den Haag'"
result = g.bfs(from_expr, to_expr)
```

Объект `result` содержит столбцы, которые описывают вершины и ребра между двумя городами. Мы можем запустить следующий код, чтобы просмотреть список возвращаемых столбцов:

```
print(result.columns)
```

и получим на выходе:

```
['from', 'e0', 'v1', 'e1', 'v2', 'e2', 'to']
```

Столбцы, начинающиеся с `e`, представляют ребра, а столбцы, начинающиеся с `v`, – вершины. Нас интересуют только вершины, поэтому давайте отфильтруем все столбцы, начинающиеся с `e`, из полученного объекта `DataFrame`:

```
columns = [column for column in result.columns if not column.startswith("e")]
result.select(columns).show()
```

Запустив код в pyspark, получим на выходе:

from	v1	v2	to
[Den Haag, 52.078…	[Hoek van Holland…	[Felixstowe, 51.9…	[Ipswich, 52.0591…

Как и ожидалось, алгоритм BFS возвращает Ипсвич! Помните, что эта функция завершается, когда она находит первое совпадение; и, как вы можете видеть на рис. 4.3, Ипсвич расположен ближе к Гааге, чем Колчестер.

Поиск в глубину

Поиск в глубину (depth first search, DFS) – это еще один фундаментальный алгоритм обхода графа. Он начинается с заданной вершины, выбирает одну из соседних вершин, а затем спускается по этому пути на максимальную глубину и возвращается.

Алгоритм DFS был изобретен французским математиком Шарлем Пьером Тремо (Charles Pierre Trémaux) как стратегия решения лабиринтов. Он является полезным инструментом для моделирования возможных путей развития сценариев.

На рис. 4.4 показан порядок, в котором мы бы посещали вершины нашего транспортного графа, если бы выполняли алгоритм DFS с началом в Гааге.

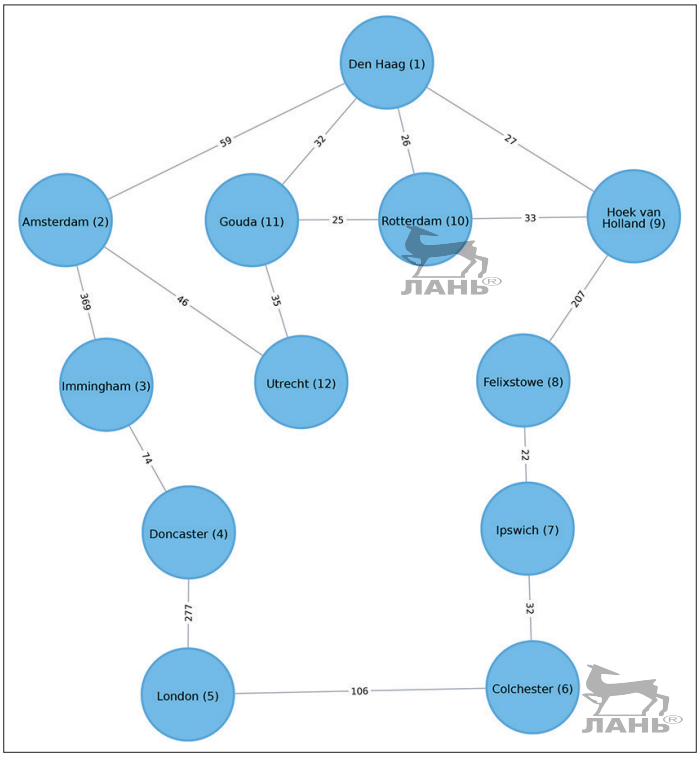


Рис. 4.4. Поиск в глубину, начиная с Гааги.
Номера вершин означают порядок обхода

Обратите внимание, насколько отличается порядок вершин по сравнению с BFS. В данном варианте DFS мы начинаем с перехода из Гааги в Амстердам, а затем можем добраться до всех остальных вершин на графе вообще без необходимости возврата назад!

Вы увидели, как алгоритмы поиска определяют основные принципы перемещения по графам. Теперь давайте рассмотрим алгоритмы поиска пути, которые находят самый дешевый путь с точки зрения количества переходов или веса. Вес может быть любым измеримым параметром, например таким как время, расстояние, емкость или стоимость.

Кратчайший путь

Алгоритм кратчайшего пути (shortest path algorithm) вычисляет кратчайший (взвешенный) путь между двумя вершинами. Он полезен для анализа взаимодействий между пользователями и для динамических рабочих процессов, потому что работает в режиме реального времени.

Поиск пути имеет историю, восходящую к XIX веку, и считается классической проблемой графов. Он приобрел известность в начале 1950-х годов в контексте альтернативной маршрутизации, т. е. для поиска второго

кратчайшего маршрута, если известный кратчайший маршрут заблокирован. В 1956 году Эдсгер Дейкстра (Edsger Dijkstra) создал самый известный из этих алгоритмов.

Два особых пути или цикла

В анализе графов есть два специальных пути, которые стоит упомянуть отдельно. Во-первых, *эйлеров путь*, или *эйлериан* (Eulerian path), – это тот путь, где каждое ребро посещается ровно один раз. Во-вторых, *гамильтонов путь*, или *гамильтониан* (Hamiltonian path), – это тот путь, где каждая вершина посещается ровно один раз. В принципе, путь может быть одновременно как эйлеровым, так и гамильтоновым, и, если вы начинаете и заканчиваете в одной и той же вершине, это считается *циклом* или *туром*. Визуальное сравнение различных специальных путей приведено на рис. 4.5.

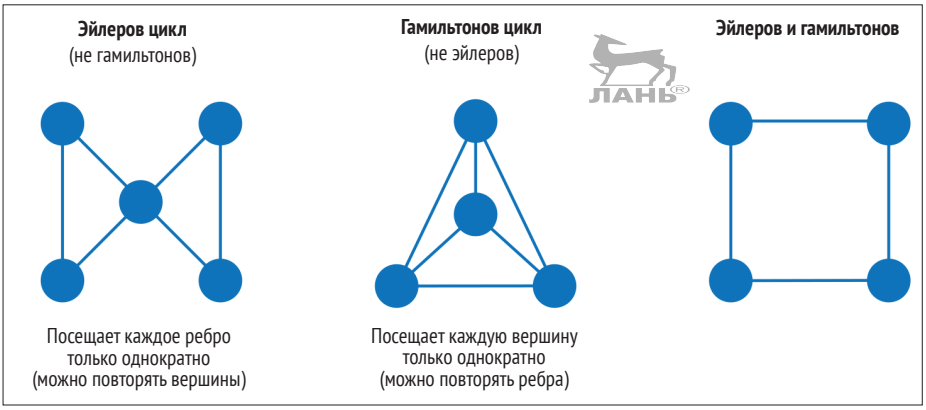


Рис. 4.5. Эйлеровы и гамильтоновы циклы имеют особое историческое значение

Задача Кенигсбергских мостов из главы 1 заключалась в поиске эйлерова цикла. Легко догадаться, что эйлеровы циклы применимы к сценариям оптимальной маршрутизации, таким как управление снегоуборочными машинами и доставка почты. Однако эйлеровы пути также используются другими алгоритмами при обработке данных в древовидных структурах и с математической точки зрения проще для изучения, чем другие циклы.

Гамильтонов цикл наиболее известен в виде *проблемы коммивояжера* (traveling salesman problem, TSP), которая гласит: «Каков должен быть кратчайший путь для продавца, чтобы посетить каждый из назначенных ему городов и вернуться в город отправления?» И хотя гамильтонов путь выглядит проще, чем цикл Эйлера, он сложнее в вычислительном отношении. Он используется в самых разных задачах планирования, логистики и оптимизации.

Алгоритм кратчайшего пути Дейкстры работает, сначала находя отношение наименьшего веса от начальной вершины к непосредственно связанным вершинам. Он отслеживает эти веса и перемещается к «ближайшему» узлу. Затем он выполняет те же вычисления, но теперь в виде совокупного итога от начальной вершины. Алгоритм продолжает делать это, оценивая «волну» совокупных весов и всегда выбирая наименьший взвешенный совокупный путь для продвижения вперед, пока не достигнет точки назначения.

В анализе графов при описании связей и путей используют термины *вес*, *стоимость*, *расстояние* и *переход*.

Вес (weight) – это числовое значение определенного свойства связи. *Стоимость* (cost) имеет аналогичное значение, но чаще применяется при рассмотрении общего веса пути. *Расстояние* (distance) часто используется в алгоритме в качестве названия свойства связи, указывающего стоимость перехода между парой вершин. Вовсе не обязательно, чтобы это был физический показатель расстояния. *Переход* (hop) или *прыжок* (jump) обычно используется для выражения количества промежуточных ребер между двумя вершинами. Иногда вы можете встретить сочетания упомянутых терминов, например: «Это расстояние в пять переходов до Лондона» или «Это самая низкая стоимость за расстояние».

Когда следует использовать алгоритм кратчайшего пути?

Используйте алгоритм кратчайшего пути, чтобы найти оптимальные маршруты между парами вершин на основе количества переходов или любого взвешенного параметра связи. Например, алгоритм может в режиме реального времени дать ответы о степени разнесенности объектов, кратчайшем расстоянии между точками или наименее дорогом маршруте. Вы также можете использовать этот алгоритм, чтобы просто исследовать связи между конкретными вершинами.

Примеры применения кратчайшего пути включают в себя:

- поиск направлений движения между локациями. Популярные картографические веб-приложения, такие как Google Maps, используют алгоритм кратчайшего пути или его варианты для построения маршрута движения;
- определение степени разделения людей в социальных сетях. Например, когда вы просматриваете чей-то профиль в LinkedIn, алгоритм показывает, сколько людей разделяют вас на графе, а также перечисляет ваши взаимные связи;

- определение степени связанности между произвольным актером и Кевином Бэйконом на основе фильмов, в которых они появлялись (*число Бэйкона*, Bacon number). Пример работы этого алгоритма можно увидеть на веб-сайте Oracle of Bacon (https://oracleofbacon.org/). Аналогичный проект, связанный с *числом Эрдёша* (Erdës number, https://www.oakland.edu/enp/), основан на анализе графа, отражающего связи современных ученых с Полом Эрдёшем, одним из самых плодовитых математиков XX века.

Алгоритм Дейкстры не поддерживает отрицательные веса. Алгоритм предполагает, что добавление новой связи к пути никогда не может сделать путь короче, – инвариант, который будет нарушен с отрицательными весами.

Реализация алгоритма кратчайшего пути с Neo4j

В библиотеке графовых алгоритмов Neo4j есть встроенная процедура, которую мы можем использовать для вычисления как невзвешенных, так и взвешенных кратчайших путей. Давайте сначала научимся вычислять невзвешенные кратчайшие пути.

Все алгоритмы кратчайшего пути Neo4j предполагают, что базовый граф является ненаправленным. Вы можете переопределить свойство, передавая параметр `direction: "OUTGOING"` (исходящий) или `direction: "INCOMING"` (входящий).

Чтобы алгоритм кратчайшего пути Neo4j игнорировал веса, нам нужно передать в качестве третьего параметра в процедуру значение `null`, указав тем самым, что мы не хотим учитывать свойство веса при выполнении алгоритма. По умолчанию алгоритм примет вес `1.0` для каждого ребра:

```
MATCH (source:Place {id: "Amsterdam"}),
      (destination:Place {id: "London"})
CALL algo.shortestPath.stream(source, destination, null)
YIELD nodeId, cost
RETURN algo.getNodeById(nodeId).id AS place, cost
```

Этот запрос вернет нам такой ответ:

place	cost
Amsterdam	0.0
Immingham	1.0
Doncaster	2.0
London	3.0

Здесь стоимость (`cost`) – это накопительная сумма ребер (или переходов). Это тот же путь, который мы получили, запустив поиск в ширину на платформе Spark.

Мы могли бы даже определить общее расстояние следования по этому пути, написав небольшую постобработку запроса Cypher. Следующая процедура находит кратчайший невзвешенный путь и затем вычисляет, какова будет фактическая стоимость этого пути:

```
MATCH (source:Place {id: "Amsterdam"}),
      (destination:Place {id: "London"})
CALL algo.shortestPath.stream(source, destination, null)
YIELD nodeId, cost

WITH collect(algo.getNodeById(nodeId)) AS path
UNWIND range(0, size(path)-1) AS index
WITH path[index] AS current, path[index+1] AS next
WITH current, next, [(current)-[r:EROAD]-(next) | r.distance][0] AS distance

WITH collect({current: current, next:next, distance: distance}) AS stops
UNWIND range(0, size(stops)-1) AS index
WITH stops[index] AS location, stops, index
RETURN location.current.id AS place,
       reduce(acc=0.0,
              distance in [stop in stops[0..index] | stop.distance] |
              acc + distance) AS cost
```

Если предыдущий код кажется немного громоздким, обратите внимание, что сложность заключается в извлечении накопительной стоимости из данных в течение всего пути. Это полезно иметь в виду, когда нам нужна совокупная стоимость пути.

Запрос возвращает следующий результат:

place	cost
Amsterdam	0.0
Immingham	369.0
Doncaster	443.0
London	720.0

На рис. 4.6 показан невзвешенный кратчайший путь из Амстердама в Лондон, который пролегает через наименьшее количество городов. Он имеет общую стоимость 720 км.

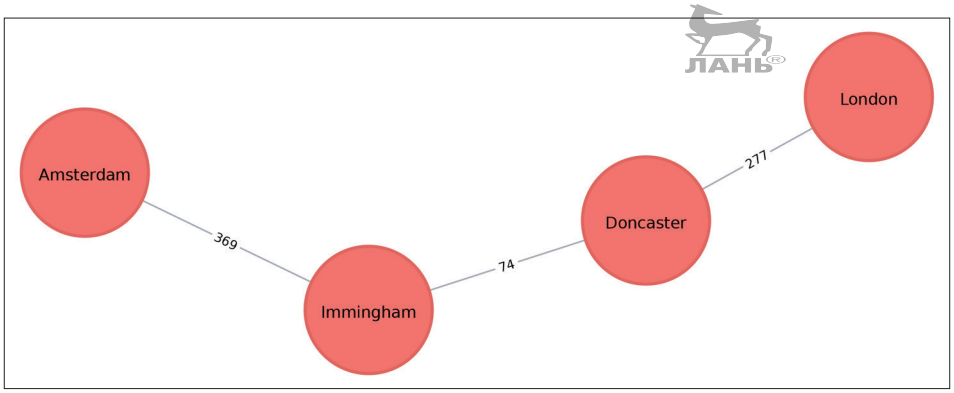


Рис. 4.6. Невзвешенный кратчайший путь между Амстердамом и Лондоном

Выбор маршрута с наименьшим количеством посещенных вершин может быть очень полезен в таких ситуациях, как построение маршрута в метро, где крайне желательно иметь меньше остановок или переходов между станциями. Однако в сценарии вождения грузовика нас, вероятно, больше интересует минимальный расход топлива с использованием кратчайшего взвешенного пути.

Поиск кратчайшего взвешенного пути с Neo4j

Для нахождения кратчайшего взвешенного пути между Амстердамом и Лондоном можно использовать следующую реализацию алгоритма:

```
MATCH (source:Place {id: "Amsterdam"}),
      (destination:Place {id: "London"})
CALL algo.shortestPath.stream(source, destination, "distance")
YIELD nodeId, cost
RETURN algo.getNodeById(nodeId).id AS place, cost
```

Мы передаем в алгоритм следующие параметры:

- `source` – вершина, где начинается наш поиск кратчайшего пути;
- `destination` – вершина, где заканчивается наш кратчайший путь;
- `distance` – имя свойства, которое определяет стоимость перехода между парой вершин.

В нашем случае стоимость – это количество километров между двумя точками. Запрос возвращает следующий результат:

place	cost
Amsterdam	0.0
Den Haag	59.0
Hoek van Holland	86.0
Felixstowe	293.0
Ipswich	315.0
Colchester	347.0
London	453.0

Как видите, самый короткий маршрут отличается от маршрута с наименьшим количеством переходов! Полная стоимость является накопительным итогом по мере продвижения по городам. Сначала мы переходим из Амстердама в Гаагу по цене 59. Затем идем из Гааги в Хук-ван-Холланд, совокупная стоимость составит 86 и т. д. Наконец, мы прибываем в Лондон из Колчестера, пройдя полную дистанцию длиной 453 км.

Вспомните, что невзвешенный кратчайший путь составлял 720 км. Учитывая вес ребер, мы смогли сэкономить 267 км при расчете кратчайшего пути.

Поиск кратчайшего взвешенного пути с Apache Spark

В разделе «Поиск в ширину с помощью Apache Spark» вы узнали, как найти кратчайший путь между двумя вершинами. Этот кратчайший путь основан на переходах и, следовательно, не совпадает с кратчайшим взвешенным путем, соответствующим кратчайшему совокупному расстоянию между городами.

Если мы хотим найти кратчайший взвешенный путь (в данном случае расстояние), нам нужно использовать свойство `cost`, которое характеризует вес ребра. Эта опция недоступна в комплекте поставки GraphFrames, поэтому нам нужно написать нашу собственную версию алгоритма кратчайшего взвешенного пути, используя фреймворк AggregateMessages. Большинство наших примеров алгоритмов для Spark основано на более простом вызове алгоритмов из библиотеки, но у нас есть возможность написания наших собственных функций. Более подробную информацию об AggregateMessages можно найти в разделе «Передача сообщений через AggregateMessages» в руководстве пользователя GraphFrames.

Перед созданием собственной функции следует импортировать нужные библиотеки:

```
from graphframes.lib import AggregateMessages as AM
from pyspark.sql import functions as F
```

Когда это возможно, мы рекомендуем использовать уже существующие протестированные библиотеки. Написание собственных функций, особенно для более сложных алгоритмов, требует углубленного понимания данных и вычислений.

Следующий пример следует рассматривать как базовую реализацию, и перед обработкой больших наборов данных его нужно оптимизировать. Если вы не интересуетесь написанием собственных функций, то можете пропустить этот пример.

Модуль AggregateMessages является частью библиотеки GraphFrames и содержит некоторые полезные вспомогательные функции.

Теперь давайте напишем нашу функцию. Сначала мы создадим пользовательскую функцию, которую будем использовать для построения путей между нашим источником и местом назначения:

```
add_path_udf = F.udf(lambda path, id: path + [id], ArrayType(StringType()))
```

А теперь займемся основной функцией, которая вычисляет кратчайший путь, начиная с источника, и завершает работу, как только достигнут пункт назначения:

```
def shortest_path(g, origin, destination, column_name="cost"):
    if g.vertices.filter(g.vertices.id == destination).count() == 0:
        return (spark.createDataFrame(sc.emptyRDD(), g.vertices.schema)
                .withColumn("path", F.array()))

    vertices = (g.vertices.withColumn("visited", F.lit(False))
                .withColumn("distance", F.when(g.vertices["id"] == origin, 0)
                            .otherwise(float("inf")))
                .withColumn("path", F.array()))
    cached_vertices = AM.getCachedDataFrame(vertices)
    g2 = GraphFrame(cached_vertices, g.edges)

    while g2.vertices.filter('visited == False').first():
        current_node_id = g2.vertices.filter('visited == False').sort
                                            ("distance").first().id

        msg_distance = AM.edge[column_name] + AM.src['distance']
        msg_path = add_path_udf(AM.src["path"], AM.src["id"])
        msg_for_dst = F.when(AM.src['id'] == current_node_id,
                             F.struct(msg_distance, msg_path))
```

```
        new_distances = g2.aggregateMessages(F.min(AM.msg).alias("aggMess"),
                                             sendToDst=msg_for_dst)

        new_visited_col = F.when(
            g2.vertices.visited | (g2.vertices.id == current_node_id),
                                                 True).otherwise(False)
        new_distance_col = F.when(new_distances["aggMess"].isNotNull() &
                                  (new_distances.aggMess["col1"]
                                  < g2.vertices.distance),
                                  new_distances.aggMess["col1"])
                                  .otherwise(g2.vertices.distance)
        new_path_col = F.when(new_distances["aggMess"].isNotNull() &
                              (new_distances.aggMess["col1"]
                              < g2.vertices.distance), new_distances.aggMess["col2"]
                              .cast("array<string>")).otherwise(g2.vertices.path)

        new_vertices = (g2.vertices.join(new_distances, on="id",
                                         how="left_outer")
                        .drop(new_distances["id"])
                        .withColumn("visited", new_visited_col)
                        .withColumn("newDistance", new_distance_col)
                        .withColumn("newPath", new_path_col)
                        .drop("aggMess", "distance", "path")
                        .withColumnRenamed('newDistance', 'distance')
                        .withColumnRenamed('newPath', 'path'))
        cached_new_vertices = AM.getCachedDataFrame(new_vertices)
        g2 = GraphFrame(cached_new_vertices, g2.edges)
        if g2.vertices.filter(g2.vertices.id == destination).first().visited:
            return (g2.vertices.filter(g2.vertices.id == destination)
                    .withColumn("newPath", add_path_udf("path", "id"))
                    .drop("visited", "path")
                    .withColumnRenamed("newPath", "path"))
    return (spark.createDataFrame(sc.emptyRDD(), g.vertices.schema)
            .withColumn("path", F.array()))
```

Если мы храним в наших функциях ссылки на какие-либо объекты DataFrames, нужно их кешировать с помощью функции `AM.getCachedDataFrame`, иначе мы столкнемся с утечкой памяти во время выполнения. В функции `shorttest_path` мы используем эту функцию для кеширования объектов `verticles` и `new_vertices`.

Если мы хотим найти кратчайший путь между Амстердамом и Колчестером, то можем вызвать созданную нами функцию:

```
result = shortest_path(g, "Amsterdam", "Colchester", "cost")
result.select("id", "distance", "path").show(truncate=False)
```

и получим следующий результат:

id	distance	path
Colchester	347.0	[Amsterdam, Den Haag, Hoek van Holland, Felixstowe, Ipswich, Colchester]

Общая длина кратчайшего пути между Амстердамом и Колчестером составляет 347 км, маршрут проходит через Гаагу, Хук-ван-Холланд, Феликстоу и Ипсвич. Напротив, кратчайший путь с точки зрения количества переходов между городами, который мы разработали с помощью алгоритма поиска в ширину (рис. 4.4), пролегал через Иммингем, Донкастер и Лондон.

Вариант алгоритма кратчайшего пути: A*

*Алгоритм кратчайшего пути A** (читается «А-звезда» от англ. *A-star*) улучшает работу алгоритма Дейкстры за счет более быстрого поиска кратчайших путей. Это достигается за счет включения дополнительной информации, которую алгоритм может использовать при эвристическом определении путей, подлежащих дальнейшему изучению.

Алгоритм был изобретен Питером Хартом, Нильсом Нильссоном и Бертрамом Рафаэлем и описан в их статье[2] 1968 года «Формальная основа эвристического определения путей минимальной стоимости».

Алгоритм A* работает, определяя, какой из частичных путей раскрывается на каждой итерации основного цикла. Это делается на основе эвристической оценки стоимости, оставшейся до достижения пункта назначения.

Будьте внимательны по отношению к эвристике, используемой для оценки стоимости пути. Недооценка стоимости пути может привести к включению лишних путей, без которых можно обойтись, но результаты все равно останутся приемлемыми. Однако, если эвристика переоценивает стоимость пути, она может пропустить фактически более короткие пути (неправильно оцененные как более длинные), что может привести к неточным результатам.

Алгоритм A* выбирает путь, который дает минимальное значение функции:

[2] *Peter Hart*, *Nils Nilsson*, and *Bertram Raphael*, «A Formal Basis for the Heuristic Determination of Minimum Cost Paths», 1968, https://bit.ly/2JAaV3s.

```
f(n) = g(n) + h(n)
```

где

- `g(n)` – стоимость пути от точки старта до вершины `n`;
- `h(n)` – эвристически предсказанная стоимость пути от вершины `n` до пункта назначения.

В реализации Neo4j в качестве эвристического параметра используется геопространственное расстояние. В нашем примере набора транспортных данных для вычисления эвристической функции мы используем широту и долготу каждого местоположения.

Реализация алгоритма A при помощи Neo4j*

Следующий запрос выполняет алгоритм A*, чтобы найти кратчайший путь между Гаагой и Лондоном:

```
MATCH (source:Place {id: "Den Haag"}),
      (destination:Place {id: "London"})
CALL algo.shortestPath.astar.stream(source,
          destination, "distance", "latitude", "longitude")
YIELD nodeId, cost
RETURN algo.getNodeById(nodeId).id AS place, cost
```

В алгоритм передаются следующие параметры:

- `source` – вершина, с которой начинается наш кратчайший путь;
- `destination` – вершина, в которой заканчивается наш кратчайший путь;
- `distance` – имя свойства, которое означает стоимость перехода между парой вершин. В нашем случае стоимость – это количество километров между двумя вершинами;
- `latitude` – имя свойства вершины, используемого для представления широты каждой вершины как компонента эвристических вычислений;
- `longtitude` – имя свойства вершины, используемого для представления долготы каждой вершины как компонента эвристических вычислений.

После запуска процедуры мы получим такой результат:

place	cost
Den Haag	0.0
Hoek van Holland	27.0
Felixstowe	234.0
Ipswich	256.0
Colchester	288.0
London	394.0

Мы получили бы тот же результат, используя алгоритм кратчайшего взвешенного пути, но на более сложных наборах данных алгоритм A* работает быстрее, поскольку он оценивает меньшее количество путей.

Вариант алгоритма кратчайшего пути: *k*-кратчайший путь Йена

Алгоритм k-кратчайшего пути Йена аналогичен алгоритму кратчайшего пути, но, вместо того чтобы находить только кратчайший путь между двумя парами вершин, он также вычисляет второй кратчайший путь, третий кратчайший путь и т. д., вплоть до (k–1)-й вариации кратчайшего пути.

Джин И. Йен изобрел алгоритм в 1971 году и описал его в статье[3] «Поиск *K* самых коротких незамкнутых путей в сети». Этот алгоритм полезен для получения альтернативных путей, когда поиск абсолютного кратчайшего пути – не единственная наша цель. Это особенно важно, когда вам нужно иметь под рукой более одного резервного маршрута!

Реализация алгоритма Йена с Neo4j

Следующий запрос выполняет алгоритм Йена, чтобы найти кратчайшие пути между Гаудой и Феликстоу:

```
MATCH (start:Place {id:"Gouda"}),
      (end:Place {id:"Felixstowe"})
CALL algo.kShortestPaths.stream(start, end, 5, "distance")
YIELD index, nodeIds, path, costs
RETURN index,
       [node in algo.getNodesById(nodeIds[1..-1]) | node.id] AS via,
       reduce(acc=0.0, cost in costs | acc + cost) AS totalCost
```

В алгоритм передаются следующие параметры:

- `start` – вершина, с которой начинается наш кратчайший путь;
- `end` – вершина, в которой заканчивается наш кратчайший путь;
- `5` – максимальное число исследуемых кратчайших путей;

[3] *Jin Y. Yen*, «Finding the *K* Shortest Loopless Paths in a Network», 1971, https://bit.ly/2HS0eXB.

- `distance` – имя свойства, которое указывает стоимость перехода между парой вершин. В нашем случае стоимость – это количество километров между двумя вершинами.

После того как получены кратчайшие пути, мы ищем соответствующую вершину для каждого идентификатора, а затем накладываем на коллекцию фильтр по начальной и конечной вершине.

Выполнение этой процедуры дает следующий результат:

index	via	totalCost
0	[Rotterdam, Hoek van Holland]	265.0
1	[Den Haag, Hoek van Holland]	266.0
2	[Rotterdam, Den Haag, Hoek van Holland]	285.0
3	[Den Haag, Rotterdam, Hoek van Holland]	298.0
4	[Utrecht, Amsterdam, Den Haag, Hoek van Holland]	374.0

На рис. 4.7 изображен кратчайший путь между Гаудой и Феликстоу.

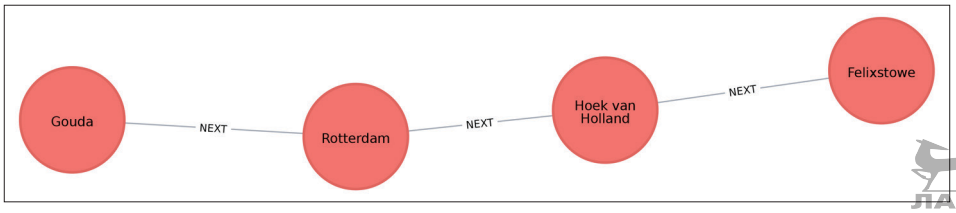


Рис. 4.7. Кратчайший путь между Гаудой и Феликстоу

Кратчайший путь на рис. 4.7 интересен в сравнении с результатами, упорядоченными по совокупной стоимости. Он показывает, что иногда вы можете рассмотреть несколько кратчайших путей или исходить из других параметров. В нашем примере второй кратчайший маршрут всего на 1 км длиннее самого короткого. Если вам нравится любоваться пейзажами, вы можете выбрать немного более длинный маршрут.

Алгоритм кратчайшего пути между всеми парами вершин

Алгоритм кратчайшего пути между всеми парами (all pairs shortest path, APSP) вычисляет кратчайший (взвешенный) путь между всеми парами вершин. Это более эффективно, чем последовательный запуск алгоритма кратчайшего пути с одним источником для каждой пары вершин в графе.

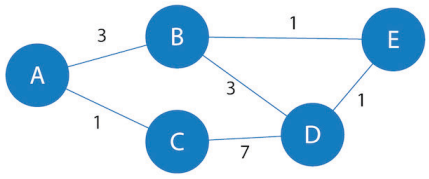
APSP оптимизирует операции, отслеживая рассчитанные расстояния и обрабатывая вершины параллельно. Полученные известные расстояния могут затем использоваться повторно при расчете кратчайшего пути к

незнакомой вершине. Изучите пример кода в следующем разделе, чтобы лучше понять, как работает алгоритм.

Некоторые пары вершин могут быть недоступны друг для друга, что означает, что между этими вершинами нет кратчайшего пути. Алгоритм не возвращает расстояния для этих пар вершин.

Подробный разбор алгоритма APSP

Вычисления, заложенные в основу APSP, легче всего понять, если изучить последовательность операций. Диаграмма на рис. 4.8 иллюстрирует вычислительные шаги для вершины *A*.



Для всех вершин задаем начальное расстояние ∞, затем подставляем 0 для первой вершины			На каждом этапе сохраняется или обновляется наименьшее значение, найденное ранее Показаны переходы только от вершины A ко всем остальным вершинам				
			1-й путь от A	2-й от A до C и далее	3-й от A до B и далее	4-й от A до E и далее	5-й от A до D и далее
A	∞	0	0	0	0	0	0
B	∞	∞	3	3	3	3	3
C	∞	∞	1	1	1	1	1
D	∞	∞	∞	8	6	5	5
E	∞	∞	∞	∞	4	4	4

Рис. 4.8. Шаги для вычисления кратчайшего пути от вершины *A* ко всем остальным вершинам (обновляемые значения выделены окрашенным фоном ячеек)

Первоначально алгоритм предполагает бесконечное расстояние до всех вершин. Когда выбрана начальная вершина, то расстояние до этой вершины устанавливается равным 0. Затем выполняются следующие операции:

1. Исходя из начальной вершины *A*, мы оцениваем стоимость перехода к вершинам, которые можем достичь на этом шаге, и обновляем

соответствующие значения. Сейчас в поиске наименьшего значения у нас есть выбор *B* (стоимость 3) или *C* (стоимость 1). Поэтому мы выбираем *C* для следующего этапа обхода.

2. Теперь, исходя из вершины *C*, алгоритм обновляет совокупные расстояния от *A* до вершин, которые могут быть достигнуты непосредственно из *C*. Значения обновляются только тогда, когда найдена более низкая стоимость:

```
A=0, B=3, C=1, D=8, E=∞
```

   Далее вершина *B* назначается в качестве следующей ближайшей вершины, которая еще не была посещена. Она имеет связи с вершинами *A*, *D* и *E*. Алгоритм вычисляет расстояние до этих вершин путем суммирования расстояния от *A* до *B* с расстоянием от *B* до каждой из этих вершин. Обратите внимание, что наименьшая стоимость от начальной вершины A до текущей вершины всегда сохраняется как *расходы прошлого периода* (sunk cost). Результаты расчета расстояния (*d*):

```
d(A,A) = d(A,B) + d(B, A) = 3 + 3 = 6
d(A,D) = d(A,B) + d(B, D) = 3 + 3 = 6
d(A,E) = d(A,B) + d(B, E) = 3 + 1 = 4
```

   - на этом шаге расстояние от вершины *A* до *B* и обратно до *A*, обозначенное как *d*(*A*,*A*) = 6, больше, чем ранее вычисленное самое короткое расстояние (0), поэтому его значение не обновляется;
   - расстояния для вершин *D* (6) и *E* (4) меньше предварительно рассчитанных расстояний, поэтому их значения обновляются.

3. Теперь выбрана вершина *E*. На этом этапе уменьшается только совокупная дистанция для достижения вершины *D* (5), и поэтому обновляется только она.
4. Когда, наконец, мы дошли до вершины *D*, то новых минимальных весов пути не существует – обновлять нечего, и алгоритм завершается.

Несмотря на то что алгоритм APSP оптимизирован для параллельных вычислений, он все равно может надолго «застрять» на больших и разветвленных графах. Попробуйте использовать подграф, если вам нужно только оценить пути между подкатегориями вершин.

Когда следует использовать APSP?

Алгоритм APSP обычно используется для альтернативной маршрутизации, когда кратчайший маршрут заблокирован или становится неоптимальным. Например, этот алгоритм используется при планировании логических маршрутов, чтобы обеспечить наличие нескольких путей и диверсифицировать маршрутизацию. Используйте поиск кратчайшего пути для всех пар, когда вам нужно рассмотреть все возможные маршруты между всеми или большинством вершин вашего графа.

Примеры использования APSP включают в себя:

- оптимизацию расположения городских объектов и распределения товаров. Одним из примеров является определение ожидаемой транспортной нагрузки на разных сегментах транспортной сети. Для получения дополнительной информации см. книгу[4] Р. К. Ларсона и А. Р. Одони «Наука городского управления»;
- поиск сети с максимальной пропускной способностью и минимальной задержкой как часть алгоритма проектирования центра обработки данных. Более подробно об этом подходе можно прочитать в статье[5] «REWIRE: основанная на оптимизации платформа для проектирования сетей центров обработки данных», автором которой являются Р. Р. Куртис и соавт.

Реализация APSP на платформе Apache Spark

Функция Spark `shorttestPaths` предназначена для поиска кратчайших путей от всех вершин до набора вершин, называемых *ориентирами* (landmark). Если бы мы хотели найти кратчайший путь из каждого места в Колчестер, Иммингем и Хук-ван-Холланд, мы бы написали следующий запрос:

```
result = g.shortestPaths(["Colchester", "Immingham", "Hoek van Holland"])
result.sort(["id"]).select("id", "distances").show(truncate=False)
```

Запустив этот код в pyspark, мы получим такой результат:

id	distances
Amsterdam	[Immingham → 1, Hoek van Holland → 2, Colchester → 4]
Colchester	[Colchester → 0, Hoek van Holland → 3, Immingham → 3]
Den Haag	[Hoek van Holland → 1, Immingham → 2, Colchester → 4]
Doncaster	[Immingham → 1, Colchester → 2, Hoek van Holland → 4]

[4] *R. C. Larson* and *A. R. Odoni*, «Urban Operations Research», Prentice-Hall.

[5] «REWIRE: An Optimization-Based Framework for Data Center Network Design», *A. R. Curtis et al*.

Felixstowe	[Hoek van Holland → 1, Colchester → 2, Immingham → 4]
Gouda	[Hoek van Holland → 2, Immingham → 3, Colchester → 5]
Hoek van Holland	[Hoek van Holland → 0, Immingham → 3, Colchester → 3]
Immingham	[Immingham → 0, Colchester → 3, Hoek van Holland → 3]
Ipswich	[Colchester → 1, Hoek van Holland → 2, Immingham → 4]
London	[Colchester → 1, Immingham → 2, Hoek van Holland → 4]
Rotterdam	[Hoek van Holland → 1, Immingham → 3, Colchester → 4]
Utrecht	[Immingham → 2, Hoek van Holland → 3, Colchester → 5]

Число рядом с каждым местоположением в столбце `distances` – это количество связей (дорог) между городами, которые мы должны пройти, чтобы добраться туда от исходной вершины. В нашем примере Колчестер – один из городов назначения (ориентир), и вы можете видеть, что у него есть 0 вершин, которые нужно пройти, чтобы добраться до самого себя, и три перехода от Иммингема и Хук-ван-Холланда. Если бы мы планировали поездку, мы могли бы использовать эту информацию, чтобы максимизировать наше свободное время в выбранных пунктах назначения.

Реализация APSP на платформе Neo4j

Neo4j предлагает параллельную реализацию алгоритма APSP, который возвращает расстояние между каждой парой вершин.

Первый параметр данной процедуры – это свойство, используемое для определения кратчайшего взвешенного пути. Если мы установим это значение в `null`, то алгоритм будет вычислять невзвешенные кратчайшие пути между всеми парами вершин.

Следующий запрос выполняет поиск путей:

```
CALL algo.allShortestPaths.stream(null)
YIELD sourceNodeId, targetNodeId, distance
WHERE sourceNodeId < targetNodeId
RETURN algo.getNodeById(sourceNodeId).id AS source,
       algo.getNodeById(targetNodeId).id AS target,
       distance
ORDER BY distance DESC
LIMIT 10
```

Этот алгоритм возвращает кратчайший путь между каждой парой вершин дважды – один раз с каждой из вершин в качестве исходной точки. Это было бы полезно, если бы вы оценивали ориентированный граф улиц с односторонним движением. Однако нам не нужно просматривать каждый путь дважды, поэтому мы фильтруем результаты, чтобы сохранить только один из них, используя предикат `sourceNodeId<targetNodeId`.

Запрос возвращает следующие результаты:

source	target	distance
Colchester	Utrecht	5.0
London	Rotterdam	5.0
London	Gouda	5.0
Ipswich	Utrecht	5.0
Colchester	Gouda	5.0
Colchester	Den Haag	4.0
London	Utrecht	4.0
London	Den Haag	4.0
Colchester	Amsterdam	4.0
Ipswich	Gouda	4.0

Эти выходные данные показывают 10 пар местоположений, которые имеют наибольшее количество связей между ними, потому что мы запросили результаты в порядке убывания (`DESC`).

Если мы хотим вычислить кратчайшие взвешенные пути, а не передавать `null` в качестве первого параметра, можем передать имя свойства, обозначающего стоимость, которое будет использоваться при вычислении кратчайшего пути. Это свойство будет затем оцениваться для определения кратчайшего взвешенного пути между каждой парой вершин.

Следующий запрос выполняет взвешенный поиск:

```
CALL algo.allShortestPaths.stream("distance")
YIELD sourceNodeId, targetNodeId, distance
WHERE sourceNodeId < targetNodeId
RETURN algo.getNodeById(sourceNodeId).id AS source,
       algo.getNodeById(targetNodeId).id AS target,
       distance
ORDER BY distance DESC
LIMIT 10
```

Запрос вернет такой результат:

source	target	distance
Doncaster	Hoek van Holland	529.0
Rotterdam	Doncaster	528.0
Gouda	Doncaster	524.0
Felixstowe	Immingham	511.0

Den Haag	Doncaster	502.0
Ipswich	Immingham	489.0
Utrecht	Doncaster	489.0
London	Utrecht	460.0
Colchester	Immingham	457.0
Immingham	Hoek van Holland	455.0

Теперь мы видим 10 пар местоположений, наиболее удаленных друг от друга с точки зрения общего расстояния между ними. Обратите внимание, что английский Донкастер часто появляется вместе с несколькими городами в Нидерландах. Похоже, мы провели бы много времени за рулем, если бы захотели отправиться в путешествие между этими районами.

Кратчайший путь из одного источника

Алгоритм кратчайшего пути из одного источника (single source shortest path, SSSP), появившийся на свет примерно в то же время, что и алгоритм кратчайшего пути Дейкстры, применяется для решения обеих проблем.

Алгоритм SSSP вычисляет кратчайший (взвешенный) путь от корневой вершины ко всем остальным вершинам на графе, как показано на рис. 4.9.

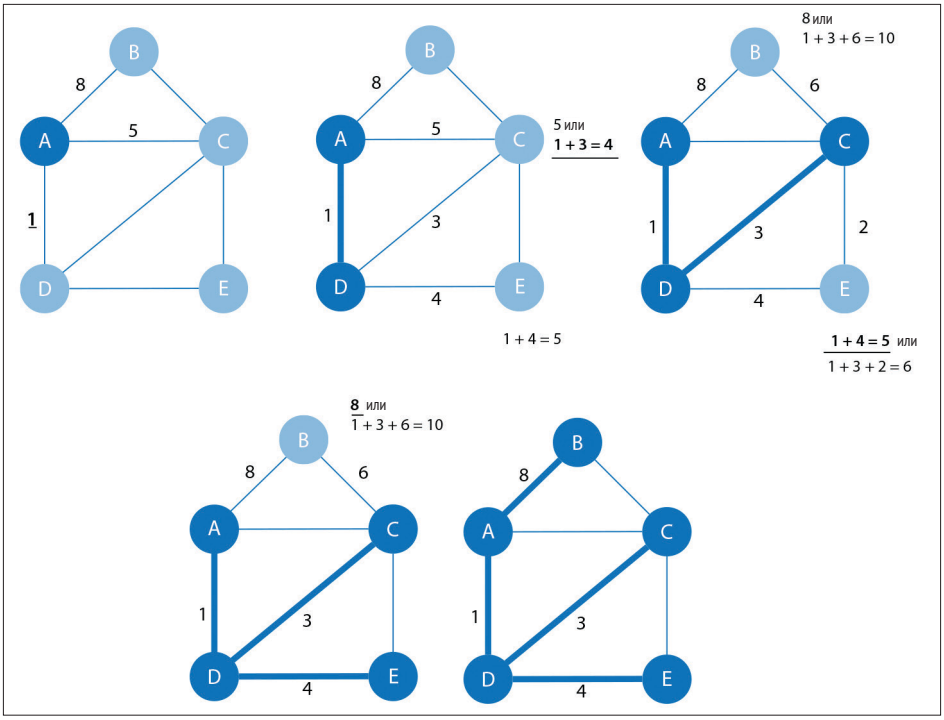


Рис. 4.9. Шаги алгоритма кратчайшего пути из одного источника

Алгоритм работает следующим образом:

1. Поиск начинается с корневой вершины, от которой будут измеряться все пути. На рис. 4.9 мы выбрали вершину *A* в качестве корневой.
2. Выбираем ребро с наименьшим весом, исходящее из этой корневой вершины, и добавляем его в дерево вместе с его связанной вершиной. В данном случае это $d(A,D) = 1$.
3. Выбираем следующее ребро с наименьшим совокупным весом от нашей корневой вершины до любой непосещенной вершины и добавляем в дерево аналогичным образом. На рис. 4.9 мы выбираем из трех вариантов: $d(A,B) = 8$, $d(A,C) = 5$ напрямую или 4 через *A*-*D*-*C* и $d(A,E) = 5$. Итак, выбираем маршрут *A*-*D*-*C* и добавляем вершину *C* в наше дерево.
4. Процесс продолжается, до тех пор пока не останется больше вершин для добавления, и у нас будет кратчайший путь из одного источника до каждой вершины.

Когда следует использовать алгоритм SSSP?

Используйте кратчайший путь из одного источника, когда необходимо оценить оптимальный маршрут от фиксированной начальной точки ко всем другим пунктам назначения. Поскольку маршрут выбирается на основе совокупного веса пути от корня, он полезен для поиска наилучшего пути к каждому пункту, но не обязательно оптимален, когда необходимо посетить *все* пункты за одну поездку.

Например, SSSP полезен для определения основных маршрутов движения экстренных служб, когда нужно посетить только одно место с одним инцидентом, но не подходит для поиска единого маршрута для сбора мусора, где вам за одну поездку нужно посетить каждый дом. В последнем случае вы будете использовать алгоритм минимального остовного дерева, описанный ниже.

Примеры использования SSSP включают в себя:

- обнаружение изменений в топологии, таких как сбои соединения, и предложение новой структуры маршрутизации за считанные секунды;
- использование алгоритма Дейкстры в качестве протокола IP-маршрутизации в автономных системах, таких как локальная сеть (LAN).

Реализация алгоритма SSSP на платформе Apache Spark

Мы можем адаптировать написанную нами функцию `shorttest_path` для вычисления кратчайшего пути между двумя местоположениями, чтобы вместо этого возвращать кратчайший путь из одного местоположения во

все остальные. Обратите внимание, что мы снова используем библиотеку Spark aggregateMessages.

Сначала импортируем те же библиотеки, что и раньше:

```
from graphframes.lib import AggregateMessages as AM
from pyspark.sql import functions as F
```

Позаимствуем пользовательскую функцию для построения путей из предыдущего примера:

```
add_path_udf = F.udf(lambda path, id: path + [id], ArrayType(StringType()))
```

Теперь займемся основной функцией, которая вычисляет кратчайший путь, начиная с источника:

```
def sssp(g, origin, column_name="cost"):
    vertices = g.vertices \
        .withColumn("visited", F.lit(False)) \
        .withColumn("distance",
            F.when(g.vertices["id"] == origin, 0).otherwise(float("inf"))) \
        .withColumn("path", F.array())
    cached_vertices = AM.getCachedDataFrame(vertices)
    g2 = GraphFrame(cached_vertices, g.edges)

    while g2.vertices.filter('visited == False').first():
    current_node_id = g2.vertices.filter('visited == False')
                        .sort("distance").first().id
    msg_path = add_path_udf(AM.src["path"], AM.src["id"])
    msg_for_dst = F.when(AM.src['id'] == current_node_id,
                  F.struct(msg_distance, msg_path))
    new_distances = g2.aggregateMessages(
        F.min(AM.msg).alias("aggMess"), sendToDst=msg_for_dst)

    new_visited_col = F.when(
        g2.vertices.visited | (g2.vertices.id == current_node_id),
                               True).otherwise(False)
    new_distance_col = F.when(new_distances["aggMess"].isNotNull() &
                              (new_distances.aggMess["col1"] <
                              g2.vertices.distance),
                              new_distances.aggMess["col1"]) \
                              .otherwise(g2.vertices.distance)
    new_path_col = F.when(new_distances["aggMess"].isNotNull() &
                          (new_distances.aggMess["col1"] <
                          g2.vertices.distance),
                          new_distances.aggMess["col2"]
                          .cast("array<string>")) \
```

```
                        .otherwise(g2.vertices.path)

    new_vertices = g2.vertices.join(new_distances, on="id",
                                    how="left_outer") \
        .drop(new_distances["id"]) \
        .withColumn("visited", new_visited_col) \
        .withColumn("newDistance", new_distance_col) \
        .withColumn("newPath", new_path_col) \
        .drop("aggMess", "distance", "path") \
        .withColumnRenamed('newDistance', 'distance') \
        .withColumnRenamed('newPath', 'path')
    cached_new_vertices = AM.getCachedDataFrame(new_vertices)
    g2 = GraphFrame(cached_new_vertices, g2.edges)

return g2.vertices \
            .withColumn("newPath", add_path_udf("path", "id")) \
            .drop("visited", "path") \
            .withColumnRenamed("newPath", "path")
```

Если мы захотим найти кратчайший путь от Амстердама до остальных городов, то можем вызвать функцию следующим образом:

```
via_udf = F.udf(lambda path: path[1:-1], ArrayType(StringType()))

result = sssp(g, "Amsterdam", "cost")
(result
 .withColumn("via", via_udf("path"))
 .select("id", "distance", "via")
 .sort("distance")
 .show(truncate=False))
```

Мы объявляем другую пользовательскую функцию для выделения начальной и конечной вершины из полученного пути. Запустив код примера, получим следующий результат:

id	distance	via
Amsterdam	0.0	[]
Utrecht	46.0	[]
Den Haag	59.0	[]
Gouda	81.0	[Utrecht]
Rotterdam	85.0	[Den Haag]
Hoek van Holland	86.0	[Den Haag]
Felixstowe	293.0	[Den Haag, Hoek van Holland]

Ipswich	315.0	[Den Haag, Hoek van Holland, Felixstowe]
Colchester	347.0	[Den Haag, Hoek van Holland, Felixstowe, Ipswich]
Immingham	369.0	[]
Doncaster	443.0	[Immingham]
London	453.0	[Den Haag, Hoek van Holland, Felixstowe, Ipswich, Colchester]

В этих результатах мы видим физические расстояния в километрах от корневой вершины (Амстердам) до всех других городов на графе, упорядоченные по нарастанию.

Реализация алгоритма SSSP на платформе Neo4j

Neo4j реализует разновидность SSSP, называемую алгоритмом *дельта-шагания* или *дельта-степпинга* (Δ-stepping, delta-stepping), которая разделяет алгоритм Дейкстры на несколько этапов, выполняемых параллельно.

Следующий запрос реализует алгоритм Delta-Stepping:

```
MATCH (n:Place {id:"London"})
CALL algo.shortestPath.deltaStepping.stream(n, "distance", 1.0)
YIELD nodeId, distance
WHERE algo.isFinite(distance)
RETURN algo.getNodeById(nodeId).id AS destination, distance
ORDER BY distance
```

Запрос возвращает следующий результат:

destination	distance
London	0.0
Colchester	106.0
Ipswich	138.0
Felixstowe	160.0
Doncaster	277.0
Immingham	351.0
Hoek van Holland	367.0
Den Haag	394.0
Rotterdam	400.0
Gouda	425.0
Amsterdam	453.0
Utrecht	460.0

Здесь мы видим физические расстояния в километрах от корневой вершины (Лондон) до всех других городов на графе, упорядоченные по нарастанию.

Минимальное остовное дерево

Алгоритм минимального остовного дерева (minimum spanning tree) начинается с заданной вершины и находит все достижимые вершины и набор ребер, которые связывают вершины вместе с минимально возможным весом. Алгоритм переходит к следующей не посещенной вершине с наименьшим весом из всех посещенных вершин, избегая циклов.

Первый известный алгоритм остовного дерева с минимальным весом был разработан чешским ученым Отакаром Боровкой (Otakar Borůvka) в 1926 году. *Алгоритм Прима*, изобретенный в 1957 году, является самым простым и известным.

Алгоритм Прима похож на алгоритм кратчайшего пути Дейкстры, но, вместо того чтобы минимизировать общую длину пути, заканчивающегося в каждой взаимосвязи, он минимизирует длину каждой взаимосвязи в отдельности. В отличие от алгоритма Дейкстры он допускает отношения с отрицательным весом.

Алгоритм минимального остовного дерева работает, как показано на рис. 4.10.

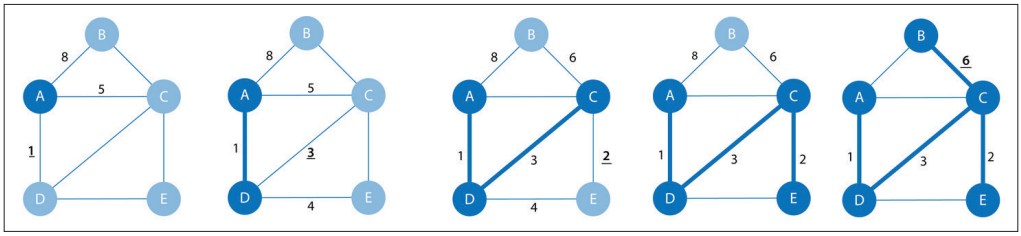


Рис. 4.10. Шаги алгоритма минимального остовного дерева

Алгоритм остовного дерева выполняет следующие шаги:

1. Алгоритм начинается с дерева, содержащего только одну вершину. На рис. 4.10 мы начинаем с вершины *A*.
2. Выбираем ребро с наименьшим весом, исходящее из этой вершины и добавляем в дерево вместе с его связанной вершиной. В нашем случае это *A*-*D*.
3. Повторяем процесс, всегда выбирая ребро с минимальным весом, ведущее к любой вершине, которой еще нет в дереве. Если вы сравните наш пример с примером SSSP на рис. 4.9, вы заметите, что на четвертом этапе пути становятся другими. Это связано с тем, что SSSP оценивает кратчайший путь на основе совокупного веса от кор-

ня, в то время как минимальное остовное дерево оценивает только стоимость следующего перехода.

4. Если больше нет вершин для добавления, то минимальное остовное дерево построено.

Существуют также варианты этого алгоритма, которые находят остовное дерево с максимальным весом (дерево с наибольшей стоимостью) и *k*-остовное дерево (размер дерева ограничен).

Когда следует использовать минимальное остовное дерево?

Используйте минимальное остовное дерево, когда вам нужен лучший маршрут для посещения всех вершин. Поскольку маршрут выбирается исходя из стоимости каждого следующего шага, алгоритм полезен, когда вы должны посетить все вершины за один проход. (Если вам не нужен путь для одной поездки, воспользуйтесь алгоритмом из предыдущего раздела.)

Вы можете применять этот алгоритм для оптимальной прокладки коммуникаций в таких системах, как водопроводные трубы и электропроводка. Он также используется для аппроксимации некоторых проблем с неизвестным временем вычислений, таких как задача коммивояжера и некоторые типы проблем округления. Хотя этот алгоритм не всегда может найти абсолютно оптимальное решение, он делает потенциально сложный и вычислительно нагруженный анализ гораздо более доступным.

Примеры использования минимального остовного дерева включают в себя:

- минимизацию дорожных расходов для изучения страны. В статье[6] «Применение минимальных остовных деревьев к планированию поездок» описано, как алгоритм с этой целью проанализировал авиационные и морские маршруты;
- визуализацию корреляций между поступлениями валютной выручки. Это применение описано в статье[7] «Применение минимального остовного дерева на валютном рынке»;
- отслеживание путей передачи инфекции при эпидемии. Для получения дополнительной информации см. публикацию[8] «Использование модели минимального остовного дерева для молекулярно-эпидемиологического исследования внутрибольничной вспышки вирусной инфекции гепатита С».

---

[6] «An Application of Minimum Spanning Trees to Travel Planning», *Lakoa Fitina, John Imbal et al.*, https://bit.ly/2CQBs6Q.

[7] «Minimum Spanning Tree Application in the Currency Market», *Marcel Rešovský, Denis Horváth et al.*, https://bit.ly/2HFbGGG.

[8] «Use of the Minimum Spanning Tree Model for Molecular Epidemiological Investigation of a Nosocomial Outbreak of Hepatitis C Virus Infection», *Enea Spada, Luciano Sagliocca et al.*, https://bit.ly/2U7SR4Y.

Алгоритм минимального остовного дерева дает значимые результаты только при запуске на графе, где ребра имеют разные веса. Если граф не имеет весов или все отношения имеют одинаковый вес, то любое остовное дерево является минимальным.

Реализация минимального остовного дерева на платформе Neo4j

Давайте взглянем на алгоритм минимального остовного дерева в действии. Следующий запрос находит остовное дерево, начинающееся с Амстердама:

```
MATCH (n:Place {id:"Amsterdam"})
CALL algo.spanningTree.minimum("Place", "EROAD", "distance", id(n),
  {write:true, writeProperty:"MINST"})
YIELD loadMillis, computeMillis, writeMillis, effectiveNodeCount
RETURN loadMillis, computeMillis, writeMillis, effectiveNodeCount
```

Алгоритм ожидает следующие параметры:

- `Place` – метки вершины, которые необходимо учитывать при вычислении связующего дерева;
- `EROAD` – типы отношений, которые следует учитывать при вычислении остовного дерева;
- `distance` – имя свойства, которое обозначает стоимость обхода между парой вершин;
- `id(n)` – внутренний идентификатор вершины, с которой должно начинаться остовное дерево.

Этот запрос сохраняет результаты в графе. Если вы хотите вернуть минимальные веса остовного дерева, то можете выполнить следующий запрос:

```
MATCH path = (n:Place {id:"Amsterdam"})-[:MINST*]-()
WITH relationships(path) AS rels
UNWIND rels AS rel
WITH DISTINCT rel AS rel
RETURN startNode(rel).id AS source, endNode(rel).id AS destination,
                                    rel.distance AS cost
```

Запрос возвращает следующий вывод:

source	destination	cost
Amsterdam	Utrecht	46.0
Utrecht	Gouda	35.0
Gouda	Rotterdam	25.0
Rotterdam	Den Haag	26.0
Den Haag	Hoek van Holland	27.0
Hoek van Holland	Felixstowe	207.0
Felixstowe	Ipswich	22.0
Ipswich	Colchester	32.0
Colchester	London	106.0
London	Doncaster	277.0
Doncaster	Immingham	74.0

Если вы находитесь в Амстердаме и хотите посетить все остальные города из нашего перечня за одну поездку, то можете воспользоваться кратчайшим непрерывным маршрутом, показанным на рис. 4.11.

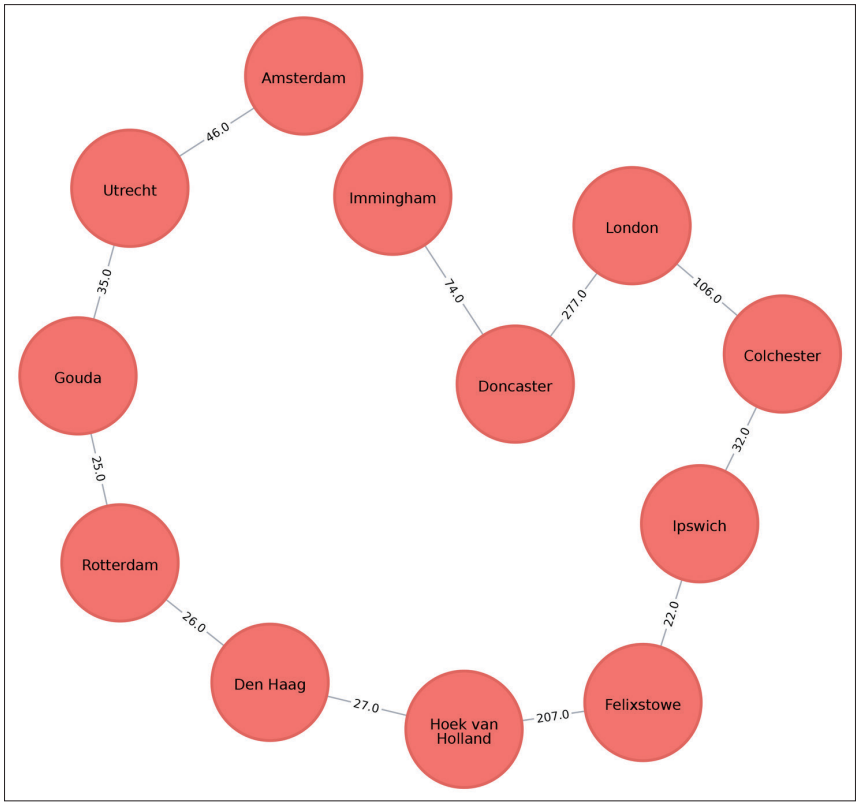


Рис. 4.11. Минимальные веса остовного дерева с корневой вершиной в Амстердаме

Алгоритм случайного блуждания

Алгоритм случайного блуждания (random walk) возвращает набор вершин на случайном пути в графе. Термин был впервые упомянут Карлом Пирсоном в 1905 году в письме[9] в журнал Nature под названием «Проблема случайного блуждания». Хотя концепция возникла еще раньше, алгоритм случайного блуждания лишь недавно нашел применение в сетевой науке.

Случайное блуждание в целом иногда описывается как походка пьяного человека, пересекающего город. Он знает, какое направление ему требуется или какую конечную точку он хочет достичь, но может пойти по очень замысловатому окольному пути.

Алгоритм начинается в одной вершине и *в некоторой степени случайно* переходит по одному из ребер вперед или назад к соседней вершине. Затем он делает то же самое из этой вершины и т. д., пока не достигнет заданной длины пути. (Мы говорим *в некоторой степени случайно*, потому что количество ребер, которые связывают вершину и ее соседей, влияет на вероятность прохождения вершины).

Когда следует использовать алгоритм случайного блуждания?

Используйте алгоритм случайного блуждания как часть других алгоритмов или конвейеров данных, когда вам нужно сгенерировать в основном случайный набор связанных вершин.

Примеры использования включают в себя:

- часть алгоритмов node2vec и graph2vec, которые создают векторные представления вершин. Эти представления могут затем использоваться в качестве входных данных для нейронной сети;
- часть алгоритмов обнаружения сообщества Walktrap и Infomap. Если при случайном обходе несколько раз возвращается небольшой набор вершин, это означает, что данный набор может иметь структуру сообщества;
- часть процесса машинного обучения. Более подробно это описано в статье[10] Дэвида Мака «Обзорное прогнозирование с помощью Neo4j и TensorFlow».

Вы можете прочитать о других случаях применения алгоритма в статье[11] Н. Масуды, М. А. Портера и Р. Ламбиотт «Случайное блуждание и диффузия в сетях».

---

[9] Karl Pearson, «The Problem of the Random Walk», Nature, 1905, https://go.nature.com/2Fy15em.

[10] David Mack, «Review Prediction with Neo4j and TensorFlow», https://bit.ly/2Cx14ph.

[11] N. Masuda, M. A. Porter, and R. Lambiotte, «Random Walks and Diffusion on Networks», https://bit.ly/2JDvlJ0.

Реализация алгоритма случайного блуждания на платформе Neo4j

Neo4j имеет встроенную реализацию алгоритма случайного блуждания. Она поддерживает два режима выбора следующего ребра для перехода на каждом этапе алгоритма:

- `random` – случайным образом выбирает ребро, по которому нужно следовать;
- `node2vec` – выбирает ребро, по которому нужно следовать, основываясь на вычислении распределения вероятности предыдущих соседей.

Следующий запрос запускает алгоритм случайного блуждания:

```
MATCH (source:Place {id: "London"})
CALL algo.randomWalk.stream(id(source), 5, 1)
YIELD nodeIds
UNWIND algo.getNodesById(nodeIds) AS place
RETURN place.id AS place
```

Параметры, передаваемые этому алгоритму:

- `id (source)` – внутренний идентификатор начальной вершины для случайного обхода;
- `5` – количество переходов, которое должен пройти случайный обход;
- `1` – количество случайных обходов, которые мы хотим вычислить.

Запрос возвращает следующий результат (один из возможных):

place
London
Doncaster
Immingham
Amsterdam
Utrecht
Amsterdam

На каждом этапе случайного блуждания следующее ребро выбирается случайным образом. Это означает, что, если мы повторно запустим алгоритм даже с теми же параметрами, мы, вероятно, не получим тот же результат. Можно даже прогуляться, вернувшись в исходную точку, как видно на рис. 4.12, где путь лежит из Амстердама в Утрехт и обратно.

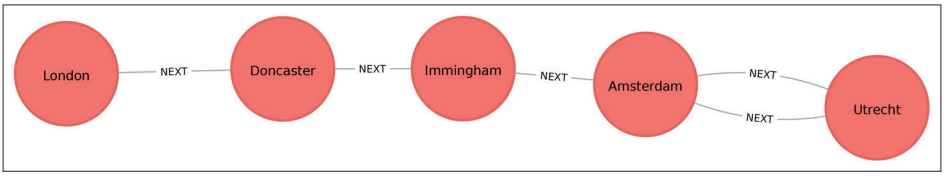


Рис. 4.12. Случайное блуждание с началом в Лондоне

Заключение

Алгоритмы поиска пути полезны для понимания того, как связаны наши данные. В этой главе мы начали с фундаментальных алгоритмов поиска в ширину и глубину, а затем перешли к алгоритму Дейкстры и другим алгоритмам кратчайшего пути. Мы также рассмотрели варианты алгоритмов кратчайшего пути, оптимизированные для поиска кратчайшего пути от одной вершины ко всем остальным вершинам или между всеми парами вершин в графе. Мы закончили главу знакомством с алгоритмом случайного блуждания, который можно использовать для поиска произвольных наборов путей.

Далее вы узнаете об алгоритмах вычисления центральности, которые можно использовать для поиска влиятельных вершин в графе.

Дополнительное чтение

Существует много книг про алгоритмы, но одна из них выделяется тем, что охватывает основные понятия и графовые алгоритмы: «Алгоритмы. Руководство по разработке», Стивен С. Скиена. Мы настоятельно рекомендуем этот учебник тем, кто ищет исчерпывающий источник информации по классическим алгоритмам и методам проектирования или просто хочет глубже изучить работу различных алгоритмов.

Глава 5

Алгоритмы вычисления центральности

Алгоритмы вычисления центральности (centrality algorithm, далее просто *алгоритмы центральности*) используются для понимания роли отдельных узлов в сети и оценки уровня их влияния на сеть. Они полезны, потому что выявляют наиболее важные узлы и помогают нам понять групповую динамику, такую как достоверность, доступность, скорость обмена информацией, и мосты между группами. Хотя многие из этих алгоритмов были изобретены для анализа социальных сетей, с тех пор они нашли применение в других областях.

Мы рассмотрим следующие алгоритмы:

- центральность по степени – как базовая метрика связности;
- центральность по близости для измерения того, насколько центральным является узел в группе, включая два варианта для разрозненных групп;
- центральность через посредника для поиска контрольных точек, включая альтернативу для аппроксимации;
- PageRank как критерий общего влияния, а также популярный способ персонализации результатов.

Различные алгоритмы центральности могут давать ощутимо разные результаты в зависимости от того, что они должны измерять. Когда вы видите неожиданные или неоптимальные результаты, лучше убедиться, что алгоритм, который вы использовали, соответствует его назначению.

Мы объясним, как работают эти алгоритмы, и приведем примеры реализации в Spark и Neo4j. Там, где алгоритм недоступен на одной из платформ или где различия не важны, мы приведем только один пример платформы.

На рис. 5.1 показаны различия между типами вопросов, на которые могут ответить алгоритмы центральности, а в табл. 5.1 приведен сводный обзор алгоритмов и примеры их применения.

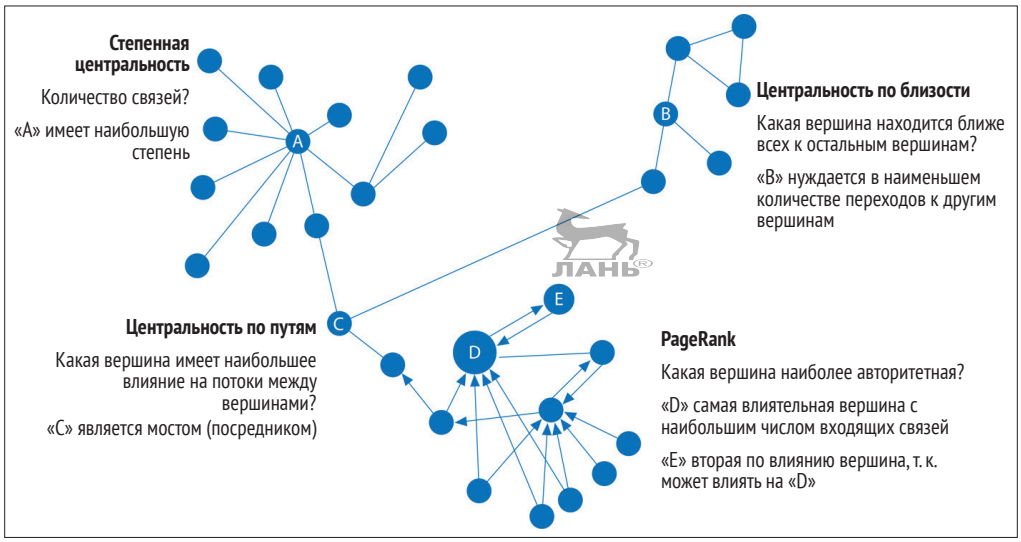


Рис. 5.1. Основные алгоритмы центральности и типы вопросов, на которые они отвечают

Таблица 5.1. Обзор алгоритмов центральности

Тип метрики в алгоритме	Что делает алгоритм	Пример использования	Пример Spark	Пример Neo4j
Степенная центральность	Измеряет количество отношений, которые имеет узел	Оценка популярности человека путем рассмотрения его степени и использования его степени для оценки общительности	Да	Нет
Центральность по близости, алгоритм Вассермана–Фауст, гармоническая центральность	Вычисляет, какие узлы имеют кратчайшие пути ко всем остальным узлам	Поиск оптимального местоположения новых общественных услуг для максимальной доступности	Да	Да
Центральность по посредничеству, алгоритм Брандеса	Измеряет количество кратчайших путей, проходящих через узел	Улучшение таргетинга лекарств путем поиска контрольных генов для конкретных заболеваний	Нет	Да

Тип метрики в алгоритме	Что делает алгоритм	Пример использования	Пример Spark	Пример Neo4j
PageRank, персонализованный PageRank	Оценивает важность текущего узла по его связанным соседям и соседям соседей (популярен благодаря Google)	Поиск наиболее влиятельных признаков для использования в машинном обучении и ранжирования текста по релевантности сущностей при обработке естественного языка	Да	Да

Некоторые алгоритмы центральности вычисляют кратчайшие пути между каждой парой узлов. Это хорошо работает для графов малого и среднего размера, но для больших графов может оказаться запредельным в вычислительном отношении. Чтобы сократить время выполнения на больших графах, некоторые алгоритмы (например, центральность через посредника) имеют аппроксимирующие версии.

Сначала вы познакомитесь с новым набором данных для наших примеров и научитесь импортировать эти данные в Apache Spark и Neo4j. Алгоритмы рассмотрены в соответствии с порядком перечисления в табл. 5.1. Изучение алгоритма начинается с краткого описания и – при необходимости – информации о том, как он работает. Упоминания уже рассмотренных алгоритмов будут включать меньше деталей. Большинство разделов также содержат советы о том, когда лучше использовать соответствующий алгоритм. В конце каждого раздела мы приводим пример кода, оперирующего демонстрационным набором данных.

Давайте начнем!

Пример графовых данных – социальный граф

Алгоритмы центральности применимы к любым графам, но социальные сети являются наиболее ярким примером динамически меняющегося влияния и потоков информации. Примеры в этой главе рассматривают небольшой граф в стиле Твиттера. Вы можете получить файлы вершин и связей, которые мы будем использовать для создания нашего графа, из файлового архива этой книги.

Таблица 5.2. Файл вершин social-nodes.csv

id
Alice
Bridget
Charles
Doug
Mark
Michael
David
Amy
James

Таблица 5.3. Файл связей social-relationships.csv

src	dst	relationship
Alice	Bridget	FOLLOWS
Alice	Charles	FOLLOWS
Mark	Doug	FOLLOWS
Bridget	Michael	FOLLOWS
Doug	Mark	FOLLOWS
Michael	Alice	FOLLOWS
Alice	Michael	FOLLOWS
Bridget	Alice	FOLLOWS
Michael	Bridget	FOLLOWS
Charles	Doug	FOLLOWS
Bridget	Doug	FOLLOWS
Michael	Doug	FOLLOWS
Alice	Doug	FOLLOWS
Mark	Alice	FOLLOWS
David	Amy	FOLLOWS
James	David	FOLLOWS

На рис. 5.2. представлена графовая модель социальных отношений, которую можно построить на основе имеющихся данных.

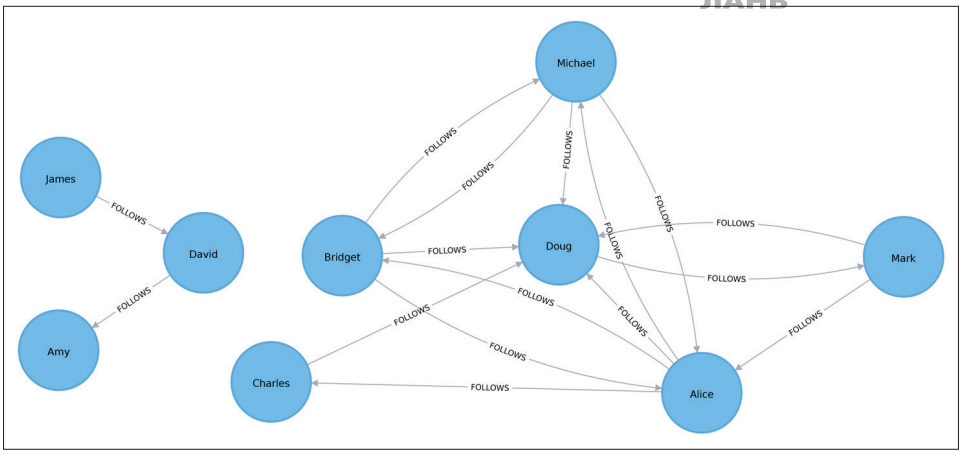


Рис. 5.2. Графовая модель социальных отношений

У нас есть одна большая группа пользователей со связями между ними и меньшая группа, не имеющая связей с большой группой.

Давайте создадим графы в Spark и Neo4j на основе содержимого этих CSV-файлов.

Импорт данных в Apache Spark

Сначала мы импортируем необходимые пакеты из Spark и пакета `GraphFrames`:

```
from graphframes import *
from pyspark import SparkContext
```

Вы можете воспользоваться следующим кодом для создания объекта `GraphFrame` на основе содержимого CSV-файлов:

```
v = spark.read.csv("data/social-nodes.csv", header=True)
e = spark.read.csv("data/social-relationships.csv", header=True)
g = GraphFrame(v, e)
```

Импорт данных в Neo4j

Далее мы загрузим данные для Neo4j. Следующий запрос импортирует узлы:

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
WITH base + "social-nodes.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MERGE (:User {id: row.id})
```

Этот запрос импортирует отношения:

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
WITH base + "social-relationships.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MATCH (source:User {id: row.src})
MATCH (destination:User {id: row.dst})
MERGE (source)-[:FOLLOWS]->(destination)
```

Итак, наш граф импортирован, можно заняться алгоритмами!

Степенная центральность

Степенна́я центральность (degree centrality) – это самая простая метрика, и, соответственно, самый простой из алгоритмов центральности, которые мы рассмотрим в этой книге. Он подсчитывает количество всех входящих и исходящих связей вершины и используется для поиска популярных вершин. Степенная центральность была предложена Линтоном К. Фрименом

в его статье[1] 1979 года «Центральность в социальных сетях: концептуальное разъяснение».

Охват вершины

Понимание *охвата* (reach) вершины является справедливой мерой важности. Скольких других вершин эта вершина может коснуться прямо сейчас? *Степень* (degree) вершины – это общее число прямых входящих и исходящих связей, которыми она обладает. Вы можете интерпретировать степень как меру непосредственной доступности узла. Например, человек с высокой степенью в социальной сети будет иметь много непосредственных контактов и с большей вероятностью окажется и в вашей сети тоже.

Средняя степень (average degree) сети – это просто общее количество связей, деленное на общее количество вершин; это значение может быть сильно искажено наличием вершин с высокой степенью. *Распределение степеней* (degree distribution) – это вероятность того, что случайно выбранная вершина будет иметь определенное число отношений.

Рисунок 5.3 иллюстрирует разницу с точки зрения фактического распределения связей между темами *сабреддитов* (subreddit) на социально-развлекательном сайте reddit.com[2]. Если просто взять среднее значение, вы бы предположили, что большинство тем имеют 10 связей, тогда как на самом деле большинство тем имеют только 2 связи.

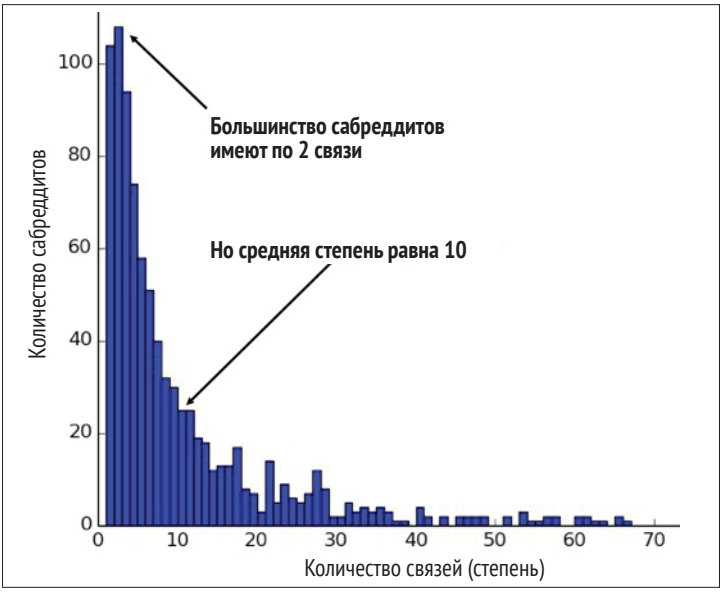


Рис. 5.3. Это отображение распределения степеней по сабреддиту, выполненное Джейкобом Сильтеррапа, демонстрирует пример того, как среднее значение не отражает фактическое распределение связей в сетях. CC BY-SA 3.0

[1] «Centrality in Social Networks: Conceptual Clarification», *Linton C. Freeman*, 1979.

[2] Российским аналогом reddit.com является сайт pikabu.ru. – *Прим. перев.*

Эти меры используются для классификации типов сетей, которые обсуждались в главе 2. Они также помогают быстро оценить вероятность распространения чего-либо по всей сети – например, сплетни или инфекции.

Когда следует использовать степенную центральность?

Используйте степенную центральность, если вы пытаетесь проанализировать влияние, глядя на количество входящих и исходящих связей, или ищете «популярность» отдельных вершин. Этот подход хорошо работает, когда вы заинтересованы в анализе непосредственных связей или краткосрочных вероятностей. Однако степенная центральность также применяется и в глобальном анализе, когда вы хотите оценить минимальную степень, максимальную степень, среднюю степень и среднеквадратичное отклонение по всему графу.

Примеры использования алгоритма степенной центральности:

- выявление влиятельных людей через их отношения, такие как связи в социальной сети. Например, в «Самых влиятельных мужчинах и женщинах в Твиттере 2017» от BrandWatch у пяти влиятельных людей в каждой категории более 40 млн подписчиков;
- отделение мошенников от благонадежных пользователей сайта онлайн-аукциона. Взвешенная центральность мошенников, как правило, значительно выше из-за сговора, направленного на искусственное повышение цен. Подробнее читайте в статье[3] П. Банчароенша и соавт. «Двухшаговое обучение с частичным подкреплением на основе графов для выявления мошенничества на аукционах».

Реализация алгоритма степенной центральности с Apache Spark

Следующий код позволяет применить алгоритм степенной центральности к данным из нашего примера социальных связей:

```
total_degree = g.degrees
in_degree = g.inDegrees
out_degree = g.outDegrees
(total_degree.join(in_degree, "id", how="left")
.join(out_degree, "id", how="left")
.fillna(0)
.sort("inDegree", ascending=False)
.show())
```

[3] «Two Step Graph-Based Semi-Supervised Learning for Online Auction Fraud Detection», *P. Bangcharoensap et al.*

Сначала мы рассчитываем количество входящих (`in_degree`) и исходящих (`out_degree`) связей. Затем объединяем эти объекты DataFrames, используя *левое соединение* (left join), чтобы сохранить любые вершины, которые не имеют входящих или исходящих связей. Если вершины не имеют связей, мы устанавливаем это значение в 0, используя функцию `fillna`.

Вот результат запуска кода в pyspark:

id	degree	inDegree	outDegree
Doug	6	5	1
Alice	7	3	4
Michael	5	2	3
Bridget	5	2	3
Charles	2	1	1
Mark	3	1	2
David	2	1	1
Amy	1	1	0
James	1	0	1

На рис. 5.4 легко заметить, что Дуг (Doug) – самый популярный пользователь в нашем графе Твиттера с пятью подписчиками (входящие связи). Все остальные пользователи в этой части графа подписаны на него, а он – только на одного человека. В реальной сети Твиттер знаменитости имеют большое количество подписчиков, но, как правило, сами подписаны лишь на немногих. Поэтому мы можем считать Дуга знаменитостью!

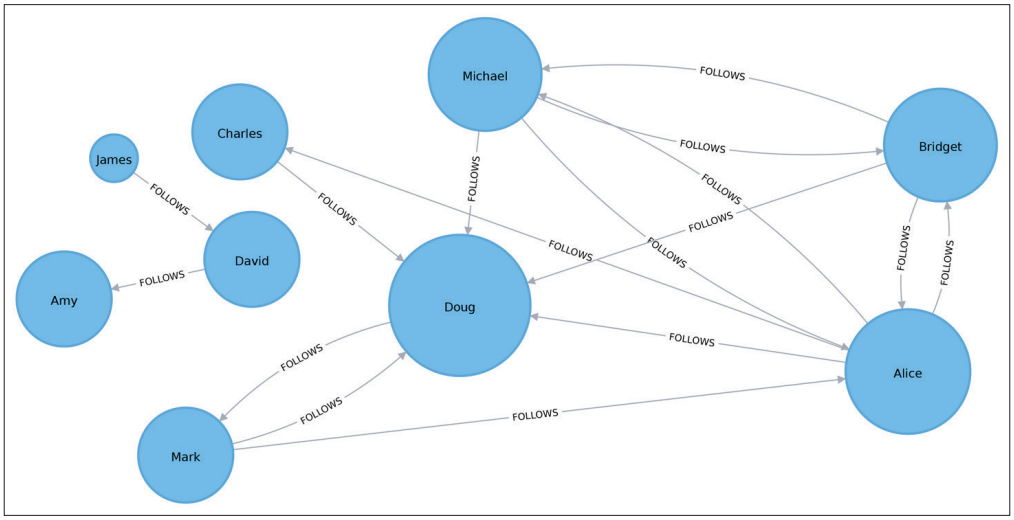


Рис. 5.4. Визуализация степенной центральности

Если бы мы создавали страницу, показывающую самых популярных пользователей сети, или хотели бы подсказать людям, на кого подписаться, мы могли бы использовать этот алгоритм для выявления таких пользователей.

Некоторые данные могут содержать очень плотные вершины с множеством связей. Они не добавляют много дополнительной информации и могут искажать некоторые результаты или увеличивать вычислительную сложность. Вы можете отфильтровать эти плотные вершины, используя подграф, или использовать проекцию графа, чтобы суммировать связи в виде весов.

Центральность по близости

Центральность по близости – это способ обнаружения вершин, которые могут эффективно распространять информацию через подграф.

Мера центральности вершины по близости – это ее *средняя дальность* (обратное расстояние) по отношению ко всем остальным вершинам. Вершины с высоким показателем близости имеют самые короткие расстояния от всех других вершин.

Алгоритм центральности по близости вычисляет сумму расстояний от вершины до всех других вершин на основе вычисления кратчайших путей между всеми парами вершин. Полученная сумма затем инвертируется, чтобы получить показатель центральности по близости для этой вершины.

Центральность по близости вычисляется по формуле:

$$C(u) = \frac{1}{\sum_{V=1}^{n-1} d(u,v)},$$

где

u – вершина;

n – количество вершин в графе;

$d(u,v)$ – длина кратчайшего пути между вершиной u и другой вершиной v.

Формула нормализованной центральности выглядит так:

$$C_{\text{norm}}(u) = \frac{n-1}{\sum_{V=1}^{n-1} d(u,v)}.$$

Когда следует использовать центральность по близости?

Применяйте центральность по близости, если хотите узнать, какие вершины распространяют информацию быстрее всего. Использование взвешенных связей может быть особенно полезно при оценке скорости взаимодействия в коммуникациях и поведенческом анализе.

Примеры использования алгоритма центральности по близости:

- выявление лиц, занимающих наиболее благоприятные позиции в отношении руководства и доступа к важной информации и ресурсов внутри организации. Одним из таких исследований[4] является «Картирование сетей террористических ячеек», В. Е. Кребс;
- эвристическая оценка времени прибытия в телекоммуникациях и доставке пакетов, когда контент перемещается по кратчайшим путям к заранее определенной цели. Он также используется для изучения распространения по всем кратчайшим путям одновременно, например при распространении инфекции через местное сообщество. Более подробную информацию можно найти в статье[5] С. П. Боргатти «Центральность и сетевой поток»;
- оценка важности слов в документе при помощи извлечения ключевой фразы на основе графа. Этот процесс описал Ф. Боден в статье[6] «Сравнение мер центральности для извлечения ключевой фразы на основе графа».

Центральность по близости лучше всего работает на связанных графах. Когда исходная формула применяется к несвязанному графу, мы получаем бесконечное расстояние между двумя вершинами, если между ними нет связи. Это означает, что мы получим бесконечную оценку центральности при суммировании всех расстояний от этой вершины. Ниже будет показано, как избежать этой проблемы путем изменения исходной формулы.

Реализация алгоритма центральности по близости с Apache Spark

Apache Spark не имеет встроенного решения для центральности по близости, но мы можем написать свой собственный код, используя библиотеку AggregateMessages, с которой вы познакомились в предыдущей главе.

[4] «Mapping Networks of Terrorist Cells», *V. E. Krebs*, http://bit.ly/2WjFdsM.

[5] «Centrality and Network Flow», *S. P. Borgatti*, http://bit.ly/2Op5bbH.

[6] «A Comparison of Centrality Measures for Graph-Based Keyphrase Extraction», *F. Boudin*, https://bit.ly/2WkDByX.

Прежде чем мы создадим нашу функцию, необходимо импортировать некоторые библиотеки:

```
from graphframes.lib import AggregateMessages as AM
from pyspark.sql import functions as F
from pyspark.sql.types import *
from operator import itemgetter
```

Мы также создадим несколько пользовательских функций, которые понадобятся позже:

```
def collect_paths(paths):
    return F.collect_set(paths)

collect_paths_udf = F.udf(collect_paths, ArrayType(StringType()))

paths_type = ArrayType(
    StructType([StructField("id", StringType()), StructField("distance",

def flatten(ids):
    flat_list = [item for sublist in ids for item in sublist]
    return list(dict(sorted(flat_list, key=itemgetter(0))).items())

flatten_udf = F.udf(flatten, paths_type)

def new_paths(paths, id):
    paths = [{"id": col1, "distance": col2 + 1} for col1,
                            col2 in paths if col1 != id]
    paths.append({"id": id, "distance": 1})
    return paths

new_paths_udf = F.udf(new_paths, paths_type)

def merge_paths(ids, new_ids, id):
    joined_ids = ids + (new_ids if new_ids else [])
    merged_ids = [(col1, col2) for col1, col2 in joined_ids if col1 != id]
    best_ids = dict(sorted(merged_ids, key=itemgetter(1), reverse=True))
    return [{"id": col1, "distance": col2} for col1, col2 in best_ids.items()]

merge_paths_udf = F.udf(merge_paths, paths_type)

def calculate_closeness(ids):
    nodes = len(ids)
    total_distance = sum([col2 for col1, col2 in ids])
    return 0 if total_distance == 0 else nodes * 1.0 / total_distance

closeness_udf = F.udf(calculate_closeness, DoubleType())
```

А теперь примемся за основной код, вычисляющий центральность по близости для каждой вершины:

```
vertices = g.vertices.withColumn("ids", F.array())
cached_vertices = AM.getCachedDataFrame(vertices)
g2 = GraphFrame(cached_vertices, g.edges)

for i in range(0, g2.vertices.count()):
    msg_dst = new_paths_udf(AM.src["ids"], AM.src["id"])
    msg_src = new_paths_udf(AM.dst["ids"], AM.dst["id"])
    agg = g2.aggregateMessages(F.collect_set(AM.msg).alias("agg"),
                               sendToSrc=msg_src, sendToDst=msg_dst)
    res = agg.withColumn("newIds", flatten_udf("agg")).drop("agg")
    new_vertices = (g2.vertices.join(res, on="id", how="left_outer")
                    .withColumn("mergedIds", merge_paths_udf("ids", "newIds",
                    "id")).drop("ids", "newIds")
                    .withColumnRenamed("mergedIds", "ids"))
    cached_new_vertices = AM.getCachedDataFrame(new_vertices)
    g2 = GraphFrame(cached_new_vertices, g2.edges)

(g2.vertices
 .withColumn("closeness", closeness_udf("ids"))
 .sort("closeness", ascending=False)
 .show(truncate=False))
```

После запуска кода мы получим следующий вывод:

id	ids	doseness
Doug	[[Charles,1], [Mark,1], [Alice,1], [Bridget,1], [Michael,1]]	1.0
Alice	[[Charles,1], [Mark,1], [Bridget,1], [Doug,1], [Michael,1]]	1.0
David	[[James,1], [Amy,1]]	1.0
Bridget	[[Charles,2], [Mark,2], [Alice,1], [Doug,1], [Michael,1]]	0.7142857142857143
Michael	[[Charles,2], [Mark,2], [Alice,1], [Doug,1], [Bridget,1]]	0.7142857142857143
James	[[Amy,2], [David,1]]	0.6666666666666666
Amy	[[James,2], [David,1]]	0.6666666666666666
Mark	[[Bridget,2], [Charles,2], [Michael,2], [Doug,1], [Alice,1]]	0.625
Charles	[[Bridget,2], [Mark,2], [Michael,2], [Doug,1], [Alice,1]]	0.625

Алиса, Дуг и Дэвид представляют наиболее плотно связанные вершины графа с оценкой `1.0`, означающей, что каждая из них напрямую соединяется со всеми вершинами в своей части графа. Рисунок 5.5 показывает, что, хотя у Дэвида есть лишь несколько связей, для его группы друзей это

важно. Другими словами, эта оценка представляет близость каждого пользователя к другим в пределах его подграфа, но не по всему графу.

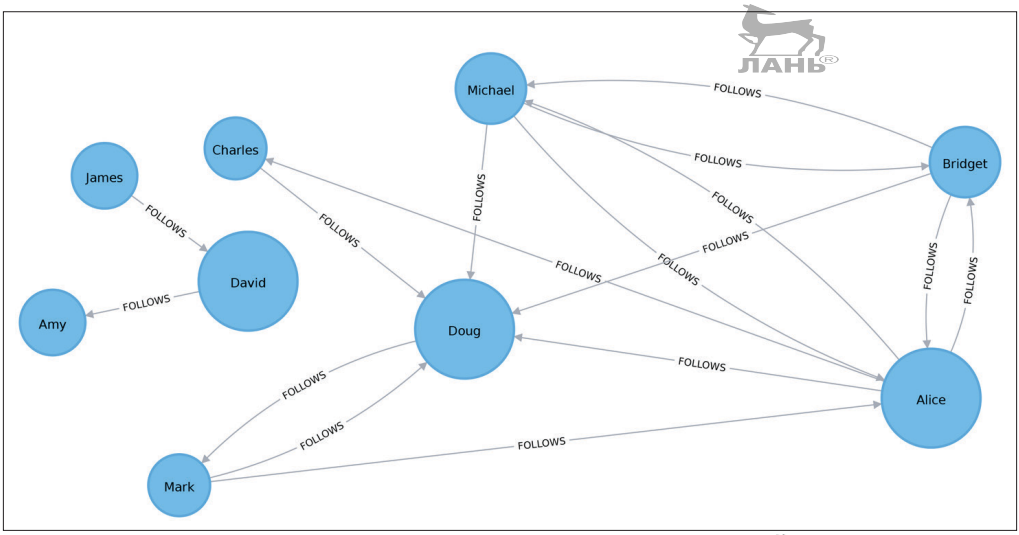


Рис. 5.5. Визуализация центральности по близости

Реализация алгоритма центральности по близости с Neo4j

Реализация центральности по близости в Neo4j основана на следующей формуле:

$$C(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(u,v)},$$

где

u – это вершина;
n – количество вершин, таких как u, в одном компоненте (подграфе или группе);
$d(u, v)$ – длина кратчайшего пути между другой вершиной v и n.

Вызов следующей процедуры вычислит центральность по близости для каждой из вершин в нашем графе:

```
CALL algo.closeness.stream("User", "FOLLOWS")
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id, centrality
ORDER BY centrality DESC
```

Процедура вернет следующий вывод:

user	centrality
Alice	1.0
Doug	1.0
David	1.0
Bridget	0.7142857142857143
Michael	0.7142857142857143
Amy	0.6666666666666666
James	0.6666666666666666
Charles	0.625
Mark	0.625

Мы получаем те же результаты, что и на платформе Spark, но, как и раньше, оценка центральности представляет близость вершин в пределах их подграфа, но не по всему графу.

В строгой интерпретации алгоритма центральности по близости все узлы в нашем графе будут иметь оценку ∞, потому что для каждой вершины в наших данных найдется хотя бы одна вершина, которую она не может достичь. Однако, как правило, более полезно реализовать оценку по каждому компоненту.

В идеале мы хотели бы получить представление о близости по всему графу, и в следующих двух разделах вы узнаете о нескольких вариациях алгоритма центральности по близости, которые делают это.

Вариант центральности по близости: Вассерман и Фауст

Стэнли Вассерман (Stanley Wasserman) и Катрин Фауст (Katherine Faust) придумали улучшенную формулу для вычисления близости в графах с несколькими подграфами без связей между этими группами. Подробная информация о формуле содержится в их книге[7] «Анализ социальных сетей: методы и приложения». Результатом этой формулы является отношение доли вершин в группе, которые достижимы, к среднему расстоянию от достижимых вершин.

Формула выглядит следующим образом:

$$C_{WF}(u) = \frac{n-1}{N-1}\left(\frac{n-1}{\sum_{v=1}^{n-1} d(u,v)}\right),$$

[7] *Stanley Wasserman and Katherine Faust*, «Social Network Analysis: Methods and Applications».

где

u – вершина;
N – общее количество вершин;
n – число вершин в том же компоненте, что и u;
$d(u, v)$ – длина кратчайшего пути между другой вершиной v и n.

Мы можем указать процедуре вычисления центральности по близости использовать эту формулу, передав параметр `improved: true`.

Следующий запрос вычисляет центральность по близости, используя формулу Вассермана–Фауст:


```
CALL algo.closeness.stream("User", "FOLLOWS", {improved: true})
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id AS user, centrality
ORDER BY centrality DESC
```

Запрос возвращает следующий результат:

user	centrality
Alice	0.5
Doug	0.5
Bridget	0.35714285714285715
Michael	0.35714285714285715
Charles	0.3125
Mark	0.3125
David	0.125
Amy	0.08333333333333333
James	0.08333333333333333

Как показано на рис. 5.6, результаты теперь более полно отражают близость вершин ко всему графу. Баллы членов меньшего подграфа (Дэвид, Эми и Джеймс) были уменьшены, и теперь у них самые низкие оценки среди всех пользователей. Это имеет смысл, поскольку они являются наиболее изолированными вершинами. Эта формула более уместна для определения важности вершины по всему графу, а не внутри его собственного подграфа.

В следующем разделе мы обсудим алгоритм гармонической центральности, который достигает аналогичных результатов, используя другую формулу для вычисления близости.

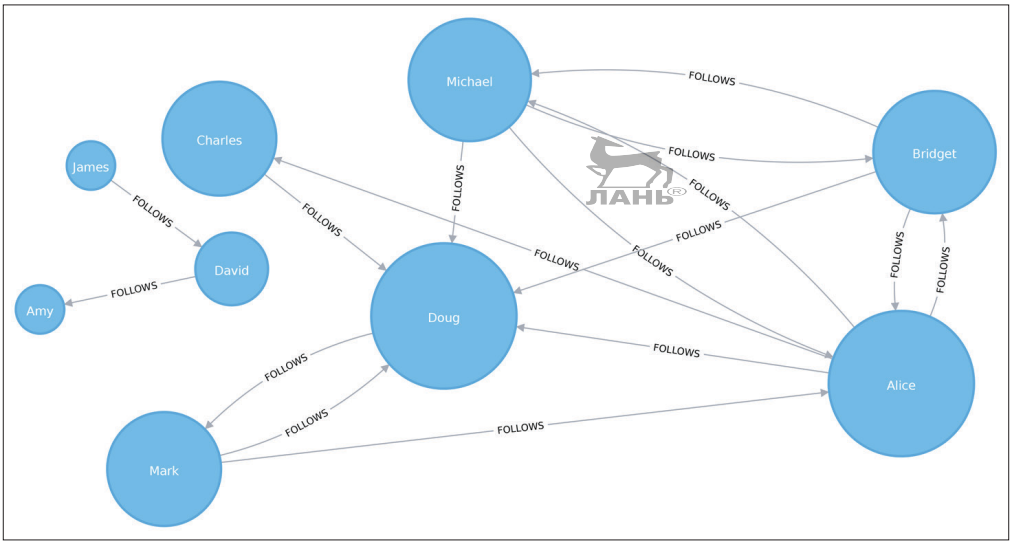


Рис. 5.6. Визуализация центральности по близости с улучшенной формулой

Вариант центральности по близости: гармоническая центральность

Гармоническая центральность (harmonic centrality, также известная как *значимая центральность*) – это вариант центральности по близости, изобретенный для решения исходной проблемы с несвязанными графами. В публикации[8] «Гармония в маленьком мире» М. Маркиори и В. Латора предложили эту концепцию в качестве практического представления среднего кратчайшего пути.

При вычислении оценки близости вместо суммирования расстояний между данной вершиной и всеми другими вершинами суммируется обратное значение этих расстояний. Это означает, что больше не появляются бесконечные значения.

Исходная гармоническая центральность вершины рассчитывается по следующей формуле:

$$H(u) = \sum_{v=1}^{n-1} \frac{1}{d(u,v)},$$

где

u – вершина;
n – количество вершин в графе;
$d(u, v)$ – длина кратчайшего пути между вершиной u и другой вершиной v.

[8] *M. Marchiori and V. Latora*, «Harmony in a Small World», https://arxiv.org/pdf/cond-mat/0008357.pdf.

Как и в случае центральности по близости, мы также можем вычислить нормализованную гармоническую центральность по следующей формуле:

$$H_{\text{norm}}(u) = \frac{\sum_{v=1}^{n-1} \frac{1}{d(u,v)}}{n-1}.$$

В этой формуле бесконечные значения обрабатываются корректно.

Реализация гармонической центральности с Neo4j

Следующий запрос запускает выполнение алгоритма гармонической центральности:

```
CALL algo.closeness.harmonic.stream("User", "FOLLOWS")
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id AS user, centrality
ORDER BY centrality DESC
```

Выполнение запроса дает следующий результат:

user	centrality
Alice	0.625
Doug	0.625
Bridget	0.5
Michael	0.5
Charles	0.4375
Mark	0.4375
David	0.25
Amy	0.1875
James	0.1875

Результаты этого алгоритма отличаются от результатов оригинального алгоритма центральности, но аналогичны результатам вычислений по улучшенной формуле Вассермана и Фауст. Оба алгоритма могут применяться при работе с графами, содержащими более одного подключенного компонента.

Центральность по посредничеству

Иногда наиболее важным винтиком в системе является не тот, кто имеет наибольшую мощность или самый высокий статус. Иногда это посредник, связывающий группы, или брокер, который в наибольшей степени кон-

тролирует перемещение ресурсов или поток информации. *Центральность по посредничеству* (betweenness centrality) – это способ определения степени влияния вершины на потоки информации или ресурсов в графе. Обычно эта метрика используется для поиска вершин, которые служат *мостом* (bridge) от одной части графа к другой.

Алгоритм центральности по посредничеству сначала вычисляет кратчайший (взвешенный) путь между каждой парой вершин в связанном графе. Каждая вершина получает оценку, основанную на количестве этих кратчайших путей, проходящих через вершину. Чем больше кратчайших путей соединено с вершиной, тем выше ее оценка.

Центральность по посредничеству считалась одной из «трех различных интуитивных концепций центральности», введенных Линтоном К. Фрименом в его статье[9] 1971 года «Набор мер центральности, основанных на посредничестве».

Мосты и контрольные точки

Мостом графа может служить вершина или ребро. На очень простом графе вы можете найти их, выявляя вершину или ребро, удаление которых приведет к отключению части графа. Однако такой подход непрактичен по отношению к реальным сложным графам, поэтому мы используем алгоритм центральности по посредничеству. Мы также можем измерить посредничество кластера, рассматривая группу как вершину.

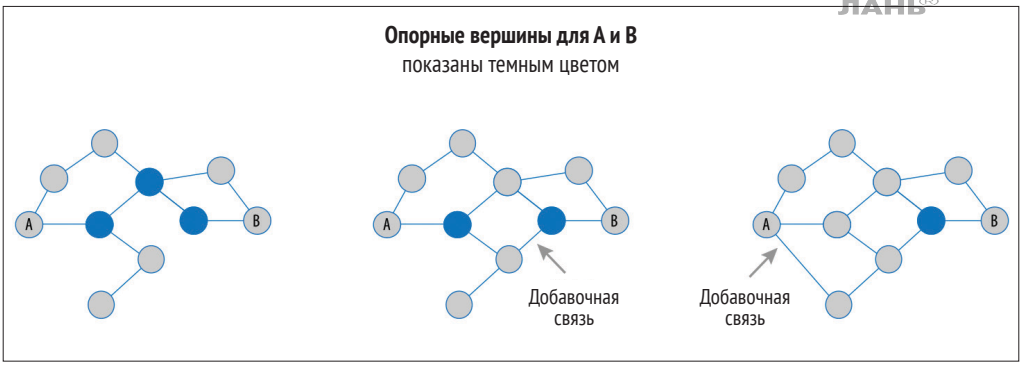


Рис. 5.7. Опорные вершины лежат на *каждом* кратчайшем пути между двумя другими вершинами. Нахождение более коротких путей может уменьшить количество опорных вершин в таких применениях, как снижение риска отказа

Вершина считается *опорной* или *основной* (pivotal) для двух других вершин, если она расположена на *каждом* кратчайшем пути между этими вершинами, как показано на рис. 5.7.

Опорные вершины играют важную роль в соединении других вершин – если вы удалите опорную вершину, новый кратчайший путь для упомя-

[9] *C. Freeman*, «A Set of Measures of Centrality Based on Betweenness», 1971, http://moreno.ss.uci.edu/23.pdf.

нутых пар вершин будет длиннее или дороже. Это может быть исходным соображением для поиска отдельных точек уязвимости.

Расчет центральности по посредничеству

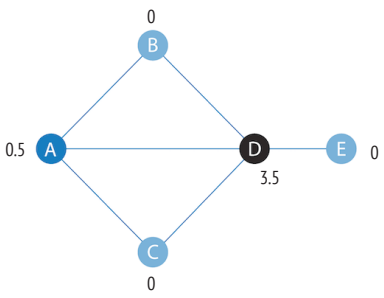
Центральность по посредничеству вычисляется путем сложения результатов следующей формулы для всех кратчайших путей:

$$B(u) = \sum_{s \neq u \neq t} \frac{p(u)}{p},$$

где

u – вершина;
p – общее количество кратчайших путей между вершинами s и t;
$p(u)$ – количество кратчайших путей между вершинами s и t, которые проходят через вершину u.

Рисунок 5.8. иллюстрирует методику вычисления центральности по посредничеству.



Вычисления для вершины D

Пары с кратчайшей связью через D	Полное число кратчайших путей этой пары	% от полного числа через D (1/полное число)
A, E	1	1
B, E	1	1
C, E	1	1
B, C	2 (через D и A)	0.5
Оценка вершины		3.5

Рис. 5.8. Методика вычисления центральности по посредничеству

Подробнее методику вычисления можно пояснить так:

1. Для каждой вершины находим кратчайшие пути, которые проходят через нее. В нашем случае вершины *B*, *C*, *E* не лежат ни на одном из кратчайших путей и им присваивается значение 0.
2. Для каждого кратчайшего пути на шаге 1 вычисляем его процент от общего числа возможных кратчайших путей для этой пары.
3. Суммируем все значения, полученные на шаге 2, чтобы найти оценку центральности. Таблица на рис. 5.8 иллюстрирует шаги 2 и 3 для узла *D*.
4. Повторяем процесс для каждой вершины.

Когда следует использовать центральность по посредничеству?

Данный алгоритм применим к широкому кругу проблем в реальных сетях. Мы используем его для поиска узких мест, контрольных точек и уязвимостей.

Примеры использования включают в себя:

- выявление влиятельных лиц в различных организациях. Влиятельные люди не обязательно находятся на руководящих должностях, но их можно найти на «брокерских должностях». Удаление таких влиятельных лиц может серьезно дестабилизировать организацию. Это может быть как успехом правоохранительных органов, если организация является преступной, так и катастрофой, если предприятие теряет ключевых сотрудников, которых оно недооценивает. Более подробную информацию можно найти в статье[10] «Оценка посредничества в активных взаимодействиях» К. Морселли и Дж. Роя;
- выявление ключевых точек передачи в сетях, таких как электрические сети. И наоборот, удаление определенных мостов может фактически *улучшить* общую надежность путем «наложения» помех. Подробности исследования представлены в статье[11] «Робастность европейских энергосистем при преднамеренной атаке», Р. Соле и соавт.;
- помощь микроблогерам в распространении информации в Твиттере с механизмом рекомендаций по таргетингу на влиятельных людей. Этот подход описан в статье[12] С. Ву и др. «Создание рекомендаций в микроблоге для повышения включенности фокусного пользователя».

Реализация центральности по посредничеству с Neo4j

В Spark нет встроенной реализации алгоритма центральности по посредничеству, поэтому мы продемонстрируем этот алгоритм с использованием Neo4j. Вызов следующей процедуры вычислит центральность по посредничеству для каждой вершины в нашем графе:

```
CALL algo.betweenness.stream("User", "FOLLOWS")
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id AS user, centrality
ORDER BY centrality DESC
```

[10] *C. Morselli and J. Roy*, «Brokerage Qualifications in Ringing Operations», https://bit.ly/2WKKPg0.

[11] *R. Sole, et al.*, «Robustness of the European Power Grids Under Intentional Attack», https://bit.ly/2Wtqyvp.

[12] *S. Wu et al.*, «Making Recommendations in a Microblog to Improve the Impact of a Focal User», https://bit.ly/2Ft58aN.

Центральность по посредничеству предполагает, что все взаимодействия между вершинами происходят по кратчайшему пути и с одинаковой частотой, что не всегда имеет место в реальной жизни. Следовательно, данный подход не дает нам идеального знания о наиболее влиятельных вершинах графа, а скорее предоставляет хороший обзор. Марк Ньюман объясняет это более подробно на с. 186 книги «Введение в сети» (издательство Оксфордского университета). (*Mark Newman*, «Networks: An Introduction», 2010, http://bit.ly/2UaM9v0.)

Выполнение этой процедуры дает нам следующий результат:

user	centrality
Alice	10.0
Doug	7.0
Mark	7.0
David	1.0
Bridget	0.0
Charles	0.0
Michael	0.0
Amy	0.0
James	0.0

Как видно на рис. 5.9, Алиса является основным посредником в этой сети, но Марк и Дуг не сильно отстают. В меньшем подграфе все кратчайшие пути проходят через Дэвида, поэтому он важен для обмена информацией между этими узлами.

В случае больших графов точное вычисление центральности непрактично. Самый быстрый известный алгоритм для точного вычисления центральности всех вершин имеет время выполнения, пропорциональное произведению числа вершин и числа ребер.

Мы можем сначала выделить интересующий нас подграф или использовать описанный в следующем разделе алгоритм, который работает с подмножеством вершин.

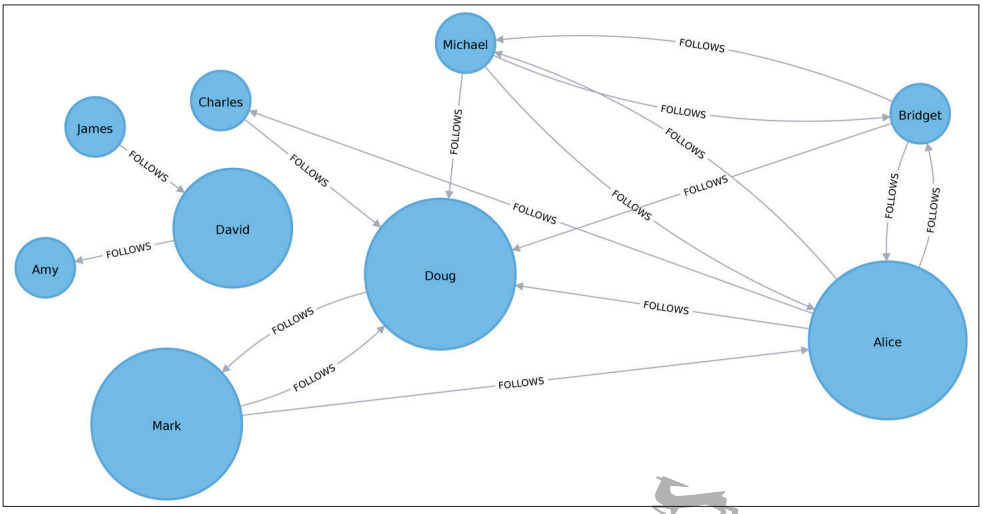


Рис. 5.9. Визуализация промежуточной центральности

Мы можем объединить два разомкнутых компонента нашего графа, добавив нового пользователя по имени Джейсон (Jason), на которого подписаны люди из обеих групп пользователей:

```
WITH ["James", "Michael", "Alice", "Doug", "Amy"] AS existingUsers

MATCH (existing:User) WHERE existing.id IN existingUsers
MERGE (newUser:User {id: "Jason"})

MERGE (newUser)<-[:FOLLOWS]-(existing)
MERGE (newUser)-[:FOLLOWS]->(existing)
```

Повторно запустив процедуру, получим новый результат:

user	centrality
Jason	44.33333333333333
Doug	18.333333333333332
Alice	16.666666666666664
Amy	8.0
James	8.0
Michael	4.0
Mark	2.1666666666666665
David	0.5
Bridget	0.0
Charles	0.0

У Джейсона самая высокая оценка, потому что связь между двумя наборами пользователей будет проходить через него. Можно сказать, что Джейсон действует как *локальный мост* между двумя группами пользователей, как показано на рис. 5.10.

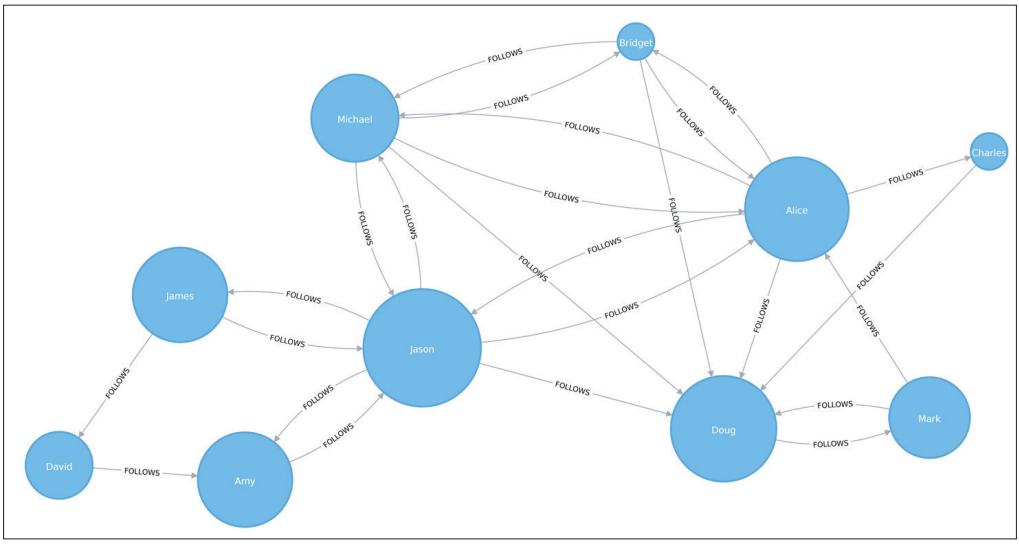


Рис. 5.10. Визуализация центральности по посредничеству при наличии связи через Джейсона

Прежде чем мы перейдем к следующему разделу, давайте вернем наш граф в исходное состояние, удалив Джейсона и его отношения:

```
MATCH (user:User {id: "Jason"})
DETACH DELETE user
```

Вариант центральности по посредничеству: алгоритм Брандеса

Напомним, что вычисление точной центральности в случае крупных графов может быть очень дорогим. В таких случаях желательно использовать аппроксимирующий алгоритм, который работает намного быстрее, но все же предоставляет полезную (хотя и не самую точную) информацию.

Алгоритм рандомизированного приближения Брандеса (randomized-approximate Brandes), который далее для краткости будем называть просто алгоритмом Брандеса, является наиболее известным алгоритмом для вычисления приблизительного значения центральности по посредничеству. Вместо вычисления кратчайшего пути между каждой парой вершин алгоритм Брандеса рассматривает только подмножество вершин. Существуют две общие стратегии для выбора подмножества вершин.

Случайная

Вершины выбираются равномерно, случайным образом, с определенной вероятностью выбора. Вероятность по умолчанию следующая: $\frac{\log 10(N)}{e^2}$. Если вероятность равна 1, алгоритм работает так же, как обычный алгоритм центральности по посредничеству, где задействованы все вершины.

Степенная

Вершины выбираются случайным образом, но те, чья степень ниже среднего, автоматически исключаются (т. е. только вершины с большим количеством связей имеют шанс на внимание алгоритма).

В качестве дальнейшей оптимизации вы можете ограничить глубину, используемую алгоритмом кратчайшего пути, который затем предоставит подмножество всех кратчайших путей.

Реализация алгоритма Брандеса на платформе Neo4j

Следующий запрос выполняет алгоритм Брандеса с использованием стратегии случайного выбора:

```
CALL algo.betweenness.sampled.stream("User", "FOLLOWS", {strategy:"degree"})
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id AS user, centrality
ORDER BY centrality DESC
```

Выполнение процедуры дает следующий результат:

user	centrality
Alice	9.0
Mark	9.0
Doug	4.5
David	2.25
Bridget	0.0
Charles	0.0
Michael	0.0
Amy	0.0
James	0.0

Наш список влиятельных лиц остался прежним, хотя у Марка сейчас более высокий рейтинг, чем у Дуга.

Из-за случайного характера этого алгоритма при каждом его запуске мы можем видеть разные результаты. Однако на больших графах эта случай-

ность будет иметь меньшее влияние, чем на нашем небольшом выборочном графе.

PageRank

PageRank является самым известным из алгоритмов центральности. Он измеряет транзитивное (или направленное) влияние вершин. Все остальные алгоритмы центральности, которые мы обсуждаем, измеряют прямое влияние вершины, тогда как PageRank учитывает влияние соседей вершины и соседей их соседей. Например, наличие нескольких очень влиятельных друзей может сделать вас более влиятельным, чем наличие множества заурядных приятелей. PageRank вычисляется либо путем итеративного распределения рейтинга текущей вершины по соседям, либо путем случайного обхода графа и подсчета частоты, с которой каждый узел попадается во время этих обходов.

PageRank назван в честь соучредителя Google Ларри Пейджа, который создал алгоритм для ранжирования сайтов в результатах поиска Google. Основным предположением является то, что страница, на которую ссылаются чаще всего (особенно если на нее ссылаются другие влиятельные страницы) является наиболее авторитетным источником информации. PageRank измеряет количество и качество входящих связей вершины, чтобы определить, насколько важна эта вершина. Предполагается, что сайты с большим влиянием на сеть имеют больше входящих ссылок от других влиятельных сайтов.

Имейте в виду, что показатели центральности представляют важность вершины по сравнению с другими вершинами. Центральность – это ранжирование возможного влияния вершин, а не мера фактического влияния. Например, вы можете обнаружить двух людей с наивысшей центральностью в сети, но, возможно, в некоторой стране существуют политические или культурные нормы, которые фактически отдают влияние другим людям. Количественная оценка фактического влияния является активной областью исследований для разработки дополнительных показателей влияния.

Влияние

Идея, лежащая в основе *влияния* (influence), заключается в том, что наличие связей с более важными вершинами дает больший вклад во влияние рассматриваемой вершины, чем эквивалентные связи с менее важными вершинами. Измерение влияния обычно включает в себя оценку вершин,

часто со взвешенными связями, а затем обновление оценок в течение многих итераций. Иногда оцениваются все вершины, а иногда в качестве репрезентативного распределения используется случайная выборка.

Формула алгоритма PageRank

В оригинальной документации Google алгоритм PageRank определен следующим образом:

$$PR(u) = (1-d) + d\left(\frac{PR(T_1)}{C(T_1)} + \ldots + \frac{PR(T_n)}{C(T_n)}\right),$$

где

мы подразумеваем, что страница *u* содержит цитаты со страниц с T_1 по T_n;

d – коэффициент затухания[13] (damping factor), который устанавливается между 0 и 1. Обычно он устанавливается на 0,85. Вы можете ассоциировать его с вероятностью того, что пользователь продолжит переход. Это помогает минимизировать снижение рейтинга, объясненное в следующем разделе;

(1 – *d*) – вероятность того, что страница достигнута напрямую, без каких-либо связей;

$C(T_n)$ определяется как степень по исходящим связям страницы *T*.

На рис. 5.11 показан небольшой пример того, как PageRank будет продолжать обновлять оценку страницы, пока не достигнет схождения или заданного числа итераций.

Итерация, случайные пользователи и ранжирование

PageRank – это итеративный алгоритм, который запускается до тех пор, пока результаты не сойдутся или пока не будет достигнуто заданное количество итераций.

Концептуально PageRank предполагает, что пользователь посещает страницы по ссылкам или по случайному URL. Коэффициент демпфирования `_d_` определяет вероятность перехода по следующей ссылке. Иными словами, коэффициент затухания можно рассматривать как вероятность того, что посетителю станет скучно и он перейдет на какую-то случайную страницу или вообще закроет браузер. *Оценка* (score) PageRank представляет вероятность того, что страница посещается по входящей ссылке, а не случайно.

[13] В последнее время в IT-публикациях *damping factor* часто переводят буквально как *коэффициент демпфирования*, хотя в других областях науки и техники (радиотехника, теория систем и т. д.) применяется именно коэффициент затухания. – *Прим. перев.*

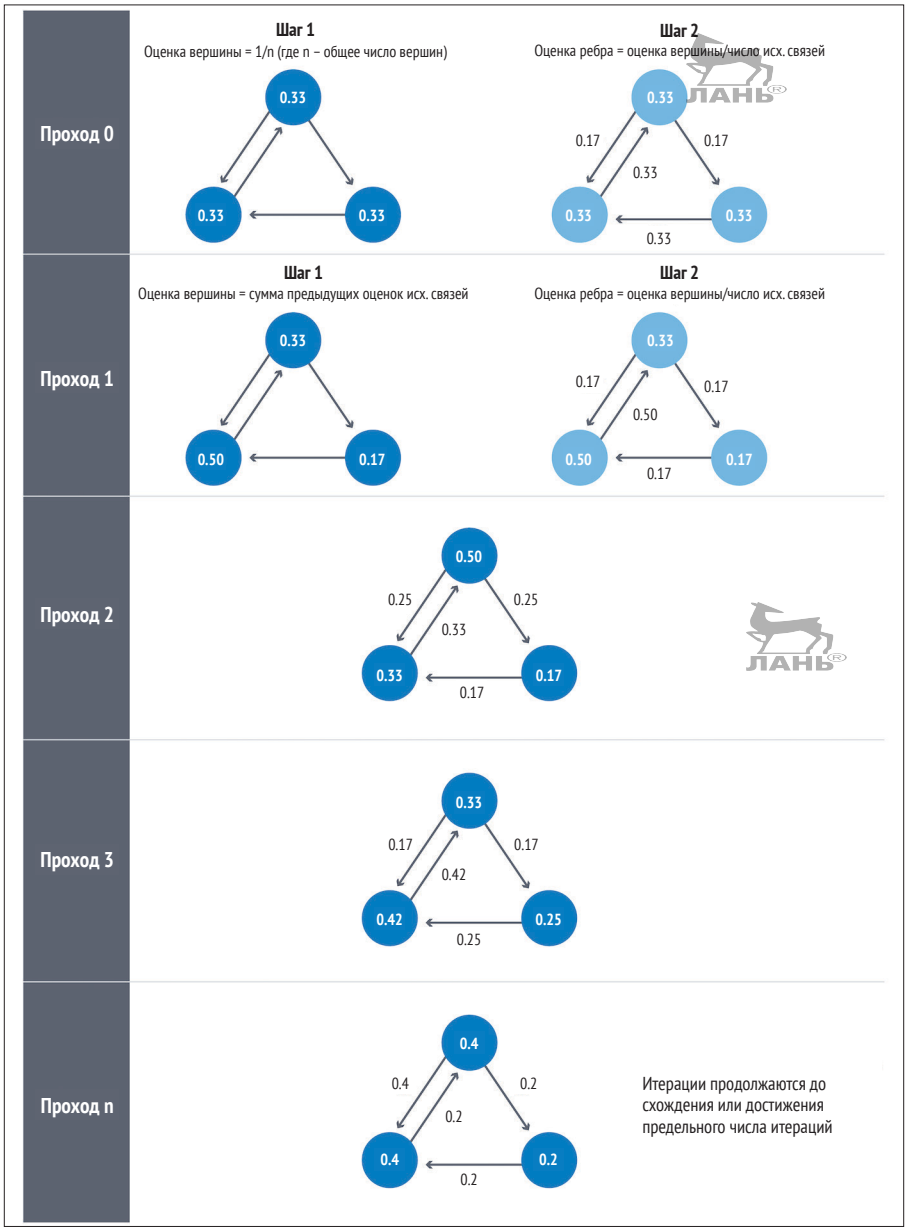


Рис. 5.11. Каждая итерация PageRank имеет два шага вычисления: один для обновления значений вершин и один для обновления значений ребер

Узлы или группа узлов сети без исходящих связей, также называемых *висячим* или *тупиковым узлом* (dangling node), могут монополизировать оценку PageRank, отказавшись предоставить ссылки для дальнейшего перехода. Это явление известно как *оценочная яма* (rank sink). По сути, это означает, что пользователь застревает на странице или подмножестве

страниц, не имея выхода. Другая проблема создается страницами, которые указывают только друг на друга внутри группы. Циркулярные ссылки вызывают увеличение рейтинга включенных страниц (и, соответственно, снижение рейтинга других страниц), поскольку посетитель перемещается по ограниченному набору страниц. Эти ситуации изображены на рис. 5.12.

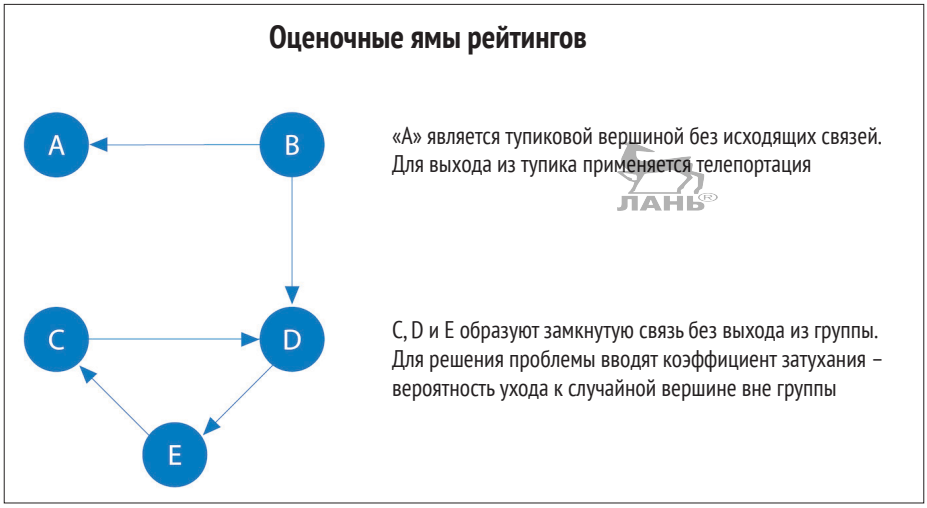


Рис. 5.12. Поглощение ранга вызывается узлом или группой узлов без исходящих отношений

Для противодействия утечке рейтинга применяются две стратегии. Во-первых, когда достигается страница, которая не имеет исходящих ссылок, PageRank предполагает наличие связей с другими страницами. Переход по этим невидимым ссылкам иногда называют *телепортацией* (teleportation). Во-вторых, коэффициент затухания предоставляет еще одну возможность избежать оценочной ямы, вводя вероятность перехода по прямой ссылке в противовес случайному посещению страницы. Когда вы устанавливаете `d = 0.85`, это соответствует посещению полностью случайной страницы в 15% случаев.

Хотя базовая формула содержит коэффициент затухания 0,85, первоначальное ранжирование страниц в интернете подчинялось степенному распределению (на большинство страниц ссылаются очень редко, а на несколько страниц – очень часто). Понижение коэффициента затухания уменьшает вероятность следования по длинным траекториям связей, прежде чем произойдет случайный переход. В свою очередь, это увеличивает вклад непосредственных предшественников страницы в ее оценку и рейтинг.

Если, применяя алгоритм PageRank, вы получаете неожиданные результаты, следует провести предварительный анализ графа и выяснить, не является ли причиной одна из упомянутых выше проблем. Для более деталь-

ного изучения алгоритма прочитайте статью[14] Яна Роджерса «Алгоритм Google PageRank, и как он работает».

Когда следует использовать PageRank?

Алгоритм PageRank подходит для множества применений кроме индексации веб-страниц. Используйте этот алгоритм всякий раз, когда вы ищете источник обширного влияния на сеть. Например, если вы ищете целевой ген, который оказывает наибольшее общее воздействие на биологическую функцию, он может и не обладать наибольшей связностью. Фактически это может быть ген, имеющий наибольшую связь с другими, более значимыми функциями (и в силу этого приобретающий собственную значимость).

Примеры использования алгоритма PageRank включают в себя:

- предоставление пользователям рекомендаций других учетных записей, на которые они могли бы подписаться (для этого Twitter использует персонализированный PageRank). Алгоритм запускается на графе, который содержит общие интересы и общие связи. Этот подход более подробно описан в статье[15] П. Гупта и др. «Механизм рекомендательного сервиса в Твиттере»;
- прогнозирование транспортных потоков и движения людей в общественных местах или на улицах. Алгоритм выполняется на графе пересечений дорог, где показатель PageRank отражает тенденцию людей парковаться или заканчивать свое путешествие на определенной улице. Это применение более подробно описано в статье[16] Б. Цзян, С. Чжао и Дж. Инь «Самоорганизованные естественные пути и прогнозирование транспортного потока: исследование зависимости»;
- часть систем обнаружения аномалий и мошенничества в сфере здравоохранения и страхования. PageRank помогает выявить врачей или клиентов, которые ведут себя необычным образом, и результаты затем вводятся в алгоритм машинного обучения.
- Дэвид Глайх описывает множество других применений алгоритма в своей статье[17] «PageRank за пределами Web».

Реализация алгоритма PageRank с Apache Spark

Теперь мы готовы к применению алгоритма PageRank. Библиотека GraphFrames поддерживает две реализации PageRank:

---

[14] *Ian Rogers*, «The Google PageRank Algorithm and How It Works», http://bit.ly/2TYSaeQ.

[15] *P. Gupta et al.*, «WTF: The Who to Follow Service at Twitter», https://stanford.io/2ux00wZ.

[16] *B. Jiang, S. Zhao, and J. Yin*, «Self-Organized Natural Roads for Predicting Traffic Flow: A Sensitivity Study», https://bit.ly/2usHENZ.

[17] *David Gleich*, «PageRank Beyond the Web», https://bit.ly/2JCYi80.

- первая реализация запускает PageRank для фиксированного числа итераций. Это можно сделать, установив параметр `maxIter`;
- вторая реализация запускает PageRank до достижения конвергенции. Это можно сделать, установив параметр `tol`.

PageRank с фиксированным числом итераций

Пример реализации алгоритма с фиксированным числом итераций выглядит очень просто:

```
results = g.pageRank(resetProbability=0.15, maxIter=20)
results.vertices.sort("pagerank", ascending=False).show()
```

Обратите внимание, что в Spark коэффициент демпфирования более интуитивно понятно называется вероятностью сброса (reset probability) и имеет обратное значение. Другими словами, `resetProbability=0.15` в этом примере эквивалентно `dampingFactor:0.85` в Neo4j.

Запустив код в pyspark, мы получим следующий результат:

user	pageRank
Doug	2.2865372087512252
Mark	2.1424484186137263
Alice	1.520330830262095
Michael	0.7274429252585624
Bridget	0.7274429252585624
Charles	0.5213852310709753
Amy	0.5097143486157744
David	0.36655842368870073
James	0.1981396884803788

Как и следовало ожидать, у Дуга самый высокий рейтинг PageRank, потому что на его твиты подписаны все остальные пользователи в его подграфе. Хотя у Марка есть только один подписчик, этим подписчиком является Дуг, поэтому Марк также считается важной персоной в этом графе. Имеет значение не только количество подписчиков, но и важность этих подписчиков!

Ребра на графе, на котором мы запускали алгоритм PageRank, не имеют весов, поэтому каждое ребро считается равным. Веса ребер добавляются путем указания столбца весов в наборе данных (DataFrame).

PageRank с конвергенцией

А теперь давайте попробуем реализовать конвергенцию, которая будет запускать PageRank до тех пор, пока решение не сойдется в пределах установленного допуска:

```
results = g.pageRank(resetProbability=0.15, tol=0.01)
results.vertices.sort("pagerank", ascending=False).show()
```

Запустив код в pyspark мы получим следующий результат:

user	pageRank
Doug	2.2233188859989745
Mark	2.090451188336932
Alice	1.5056291439101062
Michael	0.733738785109624
Bridget	0.733738785109624
Amy	0.559446807245026
Charles	0.5338811076334145
David	0.40232326274180685
James	0.21747203391449021

В данном случае точные значения рейтингов отличаются от результатов алгоритма с фиксированным числом итераций, но, как и следовало ожидать, порядок пользователей в списке остался прежним.

Хотя сходимость идеальных решений выглядит великолепно, в некоторых случаях PageRank не может математически сходиться, а для больших графов выполнение алгоритма может длиться слишком долго. Предел допуска помогает установить приемлемый диапазон точности для сходящегося результата, но многие предпочитают вместо этого использовать параметр максимального числа итераций (или комбинировать оба подхода). Как правило, чем больше итераций, тем больше точность. Независимо от того, какой вариант вы выберете, вам может потребоваться протестировать несколько различных настроек, чтобы найти, что работает лучше для вашего набора данных. Большие графы обычно требуют большего количества итераций или меньшего допуска, чем графы среднего размера.

Реализация алгоритма PageRank с Neo4j

Мы также можем запустить PageRank в Neo4j. Вызов следующей процедуры вычислит PageRank для каждой вершины в нашем графе:

```
CALL algo.pageRank.stream('User', 'FOLLOWS', {iterations:20,
dampingFactor:0.85})
YIELD nodeId, score
RETURN algo.getNodeById(nodeId).id AS page, score
ORDER BY score DESC
```

Результат выполнения процедуры:

user	pageRank
Doug	1.6704119999999998
Mark	1.5610085
Alice	1.1106700000000003
Bridget	0.535373
Michael	0.535373
Amy	0.385875
Charles	0.3844895
David	0.2775
James	0.150000000000000002

Как и в примере с Spark, Дуг – самый влиятельный пользователь, и Марк следует за ним как единственный, на кого подписался Дуг. Важность вершин относительно друг друга проиллюстрирована на рис. 5.13.

Реализации PageRank варьируются, поэтому они могут производить различную оценку, даже если порядок одинаков. Neo4j инициализирует вершины, используя значение 1 минус коэффициент демпфирования, тогда как Spark использует значение 1. В этом случае относительное ранжирование (цель PageRank) идентично, но базовые значения оценки, используемые для достижения этих результатов, отличаются.

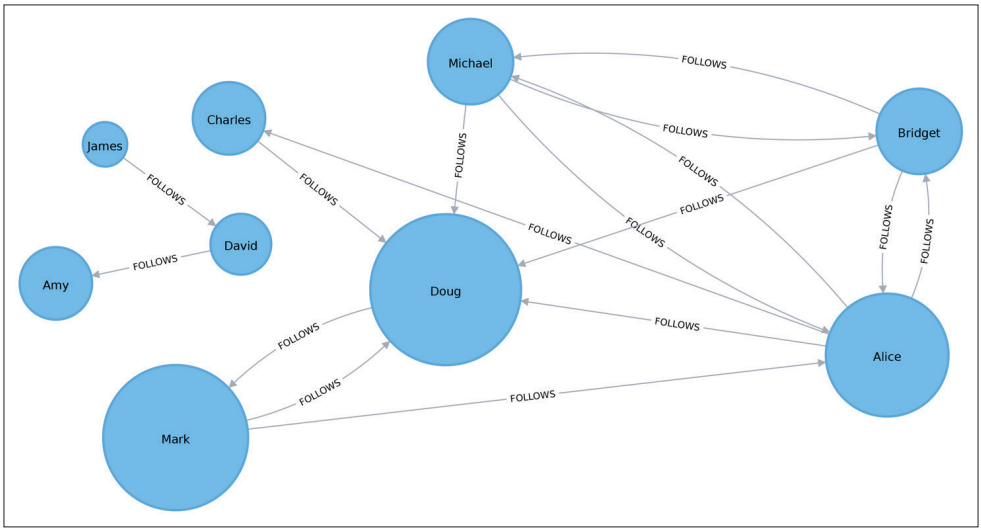


Рис. 5.13. Визуализация алгоритма PageRank

Как и в нашем примере использования Spark, ребра графа, на котором мы запускали алгоритм PageRank, не имеют весов, поэтому каждая связь считается равноценной. Веса могут учитываться путем включения свойства `weightProperty` в конфигурацию, передаваемую процедуре PageRank. Например, если ребра графа имеют свойство `weight`, содержащее веса, мы бы передали следующую конфигурацию процедуре: `weightProperty: "weight"`.

Вариант алгоритма PageRank: персонализированный PageRank

Персонализированный PageRank (personalized PageRank, PPR) – это вариант алгоритма PageRank, который вычисляет важность вершин в графе с точки зрения конкретной вершины. При использовании PPR случайные переходы направлены к ограниченному набору начальных страниц, причем вероятность выбора зависит непосредственно от «заинтересованной» вершины. Подобная ограниченность и предвзятость результатов делает PPR полезным для целенаправленных рекомендаций.

Персонализированный PageRank с Apache Spark

Мы можем вычислить персонализированную оценку PageRank для определенной вершины, передав параметр `sourceId`. Следующий код вычисляет PPR для Дуга:

```
me = "Doug"
results = g.pageRank(resetProbability=0.15, maxIter=20, sourceId=me)
people_to_follow = results.vertices.sort("pagerank", ascending=False)

already_follows = list(g.edges.filter(f"src = '{me}'").toPandas()["dst"])
people_to_exclude = already_follows + [me]

people_to_follow[~people_to_follow.id.isin(people_to_exclude)].show()
```

Результаты этого запроса можно использовать при составлении списка людей, на которых мы рекомендуем подписаться Дугу. Обратите внимание, что мы также делаем все возможное, чтобы исключить из нашего списка людей, за которыми Дуг уже следует, а также и его самого.

Если мы запустим этот код в pyspark, то получим следующий список:

user	pageRank
Alice	0.1650183746272782
Michael	0.048842467744891996
Bridget	0.048842467744891996
Charles	0.03497796119878669
David	0.0
James	0.0
Amy	0.0

Алиса – лучший кандидат, на которого мы советуем подписаться Дугу, но также можно предложить Майкла и Бриджит.

Заключение

Алгоритмы центральности являются отличным инструментом для выявления факторов влияния в сети. В этой главе вы узнали о различных метриках алгоритмов центральности: по степени, по близости, по посредничеству и PageRank. Мы также рассмотрели несколько вариантов решения таких проблем, как длительное время выполнения и изолированные компоненты, а также варианты альтернативного использования.

Существует множество способов использования алгоритмов центральности, и мы поощряем эксперименты по их применению для различных анализов. Вы можете применить полученные знания для поиска оптимальных маршрутов распространения информации, обнаружения скрытых посредников, которые контролируют поток ресурсов, и выявления влиятельных личностей, скрывающихся в тени.

Далее мы обратимся к алгоритмам обнаружения сообщества, которые работают с группами и разделами.

Глава 6

Алгоритмы выделения сообществ

Возникновение сообществ свойственно всем типам сетей, и их идентификация необходима для оценки группового поведения и возникающих в сети явлений. Общий принцип поиска сообщества основан на том, что у его членов будет больше связей внутри группы, чем с внешними узлами[1]. Анализ этих взаимосвязей выявляет кластеры узлов или изолированные группы и дает понимание структуры сети. Эта информация помогает определить аналогичное поведение или предпочтения групп сверстников, оценить устойчивость социальной группы, найти вложенные отношения и подготовить данные для других анализов. *Алгоритмы выделения сообществ* также обычно применяются при обзорной визуализации сети.

В этой главе мы подробно расскажем о наиболее характерных алгоритмах выделения сообществ, основанных на следующих подходах:

- *подсчет треугольников* и вычисление *коэффициента кластеризации* на основе общей плотности связей;
- *сильно связанные компоненты* и *связанные компоненты* для поиска связанных кластеров;
- *распространение меток* для быстрого определения групп на основе меток вершин;
- *Лувенская модульность* для оценки качества группировки и иерархий.

Мы расскажем, как работают алгоритмы, и приведем примеры реализации в Apache Spark и Neo4j. В тех случаях, когда алгоритм доступен только на одной платформе, приведем только один пример. Для этих алгоритмов мы используем графы со взвешенными ребрами, потому что они обычно используются для определения значимости различных отношений.

[1] Здесь и далее когда мы говорим о *сетях*, то используем термины *узел* и *связь*. Если же мы говорим о представляющем эту сеть *графе*, то стараемся пользоваться терминами *вершина* и *ребро*. – *Прим. перев.*

На рис. 6.1 приведен наглядный обзор различий между алгоритмами выделения сообществ, описанными в этой книге, а в табл. 6.1 кратко сказано, что вычисляет каждый алгоритм, и приведены примеры использования.

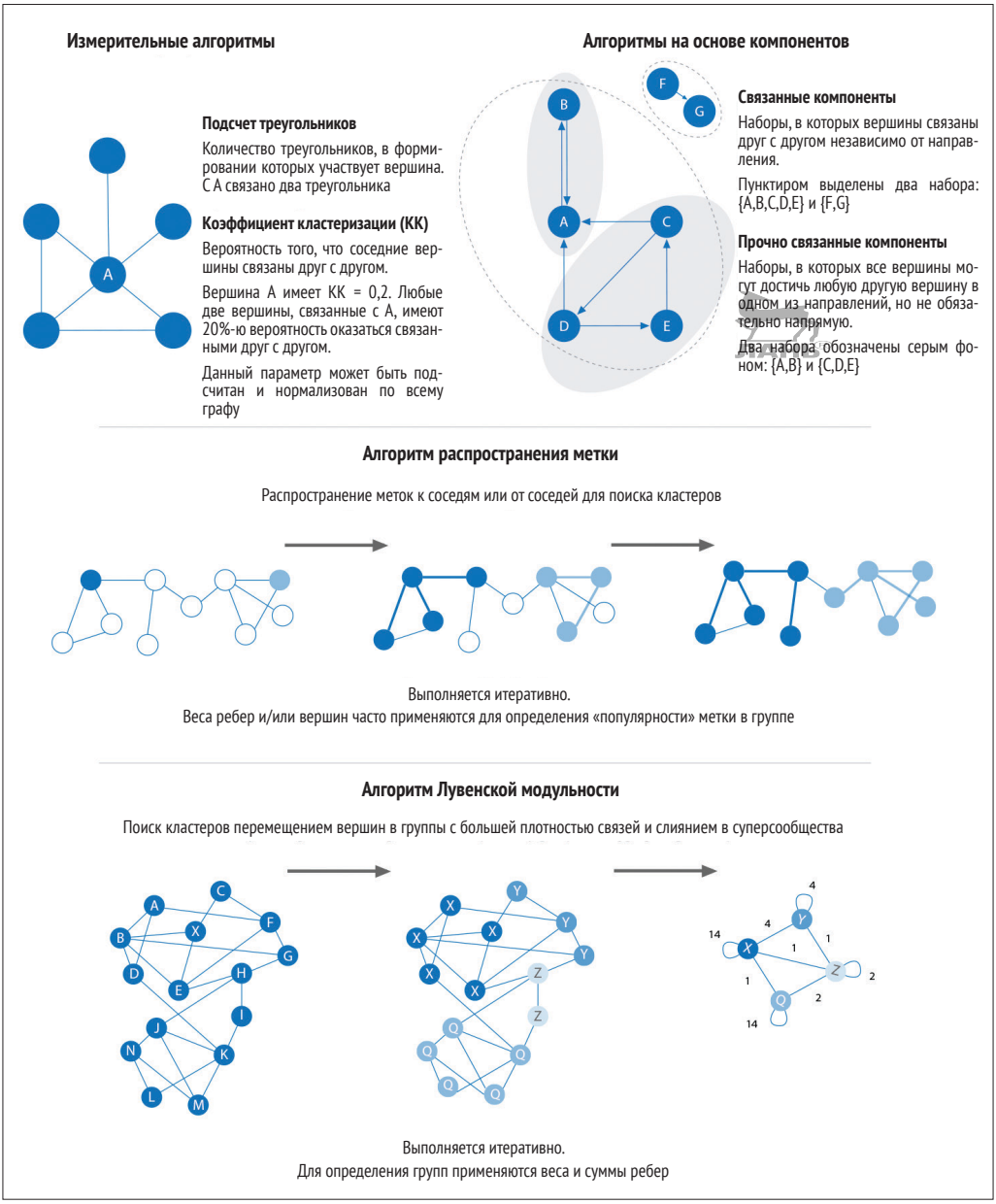


Рис. 6.1. Наиболее характерные алгоритмы выделения сообществ

Мы взаимозаменяемо используем термины *набор* (set), *раздел* (partition), *кластер* (cluster), *группа* (group) и *сообщество* (community). Эти термины являются различными способами указать, что похожие узлы сети могут образовать группу. Алгоритмы выделения сообществ также называются алгоритмами кластеризации и разделения. В каждом разделе мы используем наиболее часто употребляемые в литературе термины для конкретного алгоритма.

Таблица 6.1. Обзор алгоритмов выделения сообществ

Тип алгоритма	Что он делает	Пример использования	Пример Spark	Пример Neo4j
Подсчет треугольников и коэффициент кластеризации	Измеряет количество узлов, образующих треугольники, и степень, в которой узлы стремятся объединиться	Оценка стабильности группы и того, может ли сеть демонстрировать поведение «маленького мира», наблюдаемое на графах с тесно связанными кластерами	Да	Да
Прочно связанные компоненты	Находит группы, где каждый узел доступен из любого другого узла в той же группе, следуя направлению отношений	Выработка рекомендаций по продукту на основе групповой принадлежности или аналогичных позиций	Да	Да
Связанные компоненты	Находит группы, где каждый узел доступен из любого другого узла в той же группе независимо от направления отношений	Выполнение быстрой группировки для других алгоритмов и определения островов	Да	Да
Распространение меток	Находит кластеры, распространяя метки на основе соседства	Понимание консенсуса в социальных сообществах или нахождение опасных комбинаций возможных совместно прописанных лекарств	Да	Да
Лувенский модульный	Максимизирует предполагаемую точность группировок путем сравнения весов и плотностей отношений с ожидаемой оценкой или средним	В анализе мошенничества, оценивая, имеет ли группа только несколько отдельных плохих членов или действует как сообщество мошенников	Нет	Да

Сначала мы опишем данные для наших примеров и пройдемся по импорту данных в Spark и Neo4j. Алгоритмы описаны в порядке, указанном в табл. 6.1. Для каждого алгоритма вы найдете краткое описание и советы о том, когда его использовать. Большинство разделов также содержит рекомендации о том, когда использовать соответствующие алгоритмы. В конце каждого раздела мы приводим пример кода, использующего демонстрационные данные.

При использовании алгоритмов выделения сообществ помните о плотности отношений. Если граф очень плотный, вы можете в конечном итоге собрать все вершины в одном или нескольких кластерах. Вы можете противодействовать чрезмерно крупной кластеризации путем фильтрации по степени, весам отношений или показателям сходства.

С другой стороны, если граф слишком разрежен и состоит из нескольких связанных вершин, то в конечном итоге каждая вершина может очутиться в своем собственном кластере. В этом случае попробуйте включить дополнительные типы отношений, которые содержат более релевантную информацию.

Пример данных: граф зависимостей библиотек

Графы зависимостей библиотек программного обеспечения особенно хорошо подходят для демонстрации тонких различий между алгоритмами обнаружения сообществ, поскольку они, как правило, более связаны и имеют иерархическую структуру. Примеры в этой главе работают с графом, содержащим зависимости между библиотеками Python, хотя графы зависимостей используются в различных областях – от программного обеспечения до энергетических сетей. Этот вид графа зависимости программного обеспечения используется разработчиками для отслеживания транзитивных взаимозависимостей и конфликтов в программных проектах. Вы можете извлечь файлы данных из файлового архива книги.

Рисунок 6.2 визуализирует граф на основе данных. Глядя на этот граф, мы видим, что в нем присутствуют три кластера библиотек. Мы можем использовать визуализацию небольших наборов данных в качестве инструмента для проверки кластеров, полученных с помощью алгоритмов выделения сообществ.

Таблица 6.2. Содержимое файла sw-nodes.csv

id
six
pandas
numpy
python-dateutil
pytz
pyspark
matplotlib
spacy
py4j
jupyter
jpy-console
nbconvert
ipykernel
jpy-client
jpy-core

Таблица 6.3. Содержимое файла sw-relationships.csv

src	dst	relatonship
pandas	numpy	DEPENDS_ON
pandas	pytz	DEPENDS_ON
pandas	python-dateutil	DEPENDS_ON
python-dateutil	six	DEPENDS_ON
pyspark	py4j	DEPENDS_ON
matplotlib	numpy	DEPENDS_ON
matplotlib	python-dateutil	DEPENDS_ON
matplotlib	six	DEPENDS_ON
matplotlib	pytz	DEPENDS_ON
spacy	six	DEPENDS_ON
spacy	numpy	DEPENDS_ON
jupyter	nbconvert	DEPENDS_ON
jupyter	ipykernel	DEPENDS_ON
jupyter	jpy-console	DEPENDS_ON
jpy-console	jpy-client	DEPENDS_ON
jpy-console	ipykernel	DEPENDS_ON
jpy-client	jpy-core	DEPENDS_ON
nbconvert	jpy-core	DEPENDS_ON

Давайте создадим графы в Spark и Neo4j на основе CSV-файлов данных.

Импорт данных в Apache Spark

Сначала мы импортируем нужные нам пакеты из Apache Spark и пакета GraphFrames:

```
from graphframes import *
```

Следующая функция создает GraphFrame из примеров файлов CSV:

```
def create_software_graph():
    nodes = spark.read.csv("data/sw-nodes.csv", header=True)
    relationships = spark.read.csv("data/sw-relationships.csv", header=True)
    return GraphFrame(nodes, relationships)
```

Теперь можно вызвать эту функцию:

```
g = create_software_graph()
```

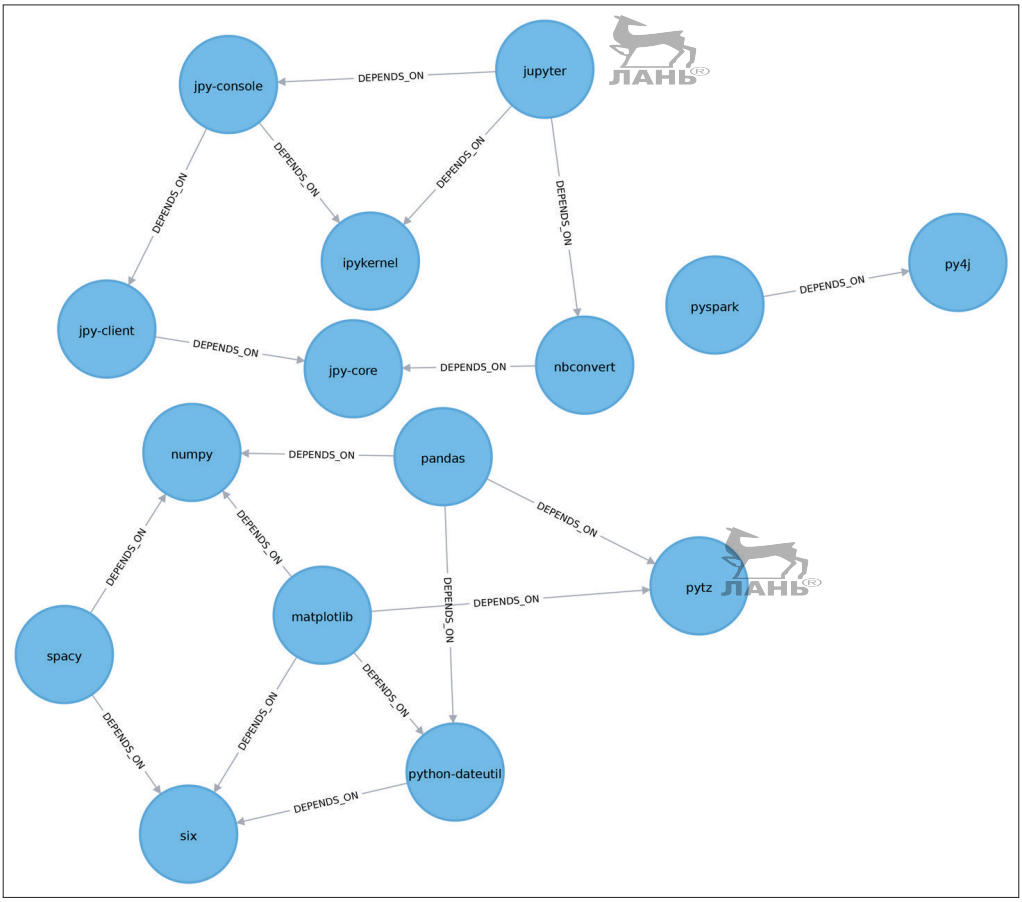


Рис. 6.2. Визуальная графовая модель данных

Импорт данных в Neo4j

Далее мы сделаем то же самое для Neo4j. Следующий запрос импортирует вершины:

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
WITH base + "sw-nodes.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MERGE (:Library {id: row.id})
```

Затем импортируем связи:

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
WITH base + "sw-relationships.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MATCH (source:Library {id: row.src})
```

```
MATCH (destination:Library {id: row.dst})
MERGE (source)-[:DEPENDS_ON]->(destination)
```

Теперь, когда наши графы загружены, займемся алгоритмами.

Подсчет треугольников и коэффициент кластеризации

Алгоритмы подсчета треугольников и *коэффициента кластеризации* представлены вместе, потому что они и в самом деле часто используются вместе. Подсчет числа треугольников возвращает количество треугольников, проходящих через каждую вершину на графе. *Треугольник* – это набор из трех вершин, где каждая вершина напрямую связана с остальными вершинами. Счетчик треугольников также может быть запущен глобально для оценки нашего общего набора данных.

Сети с большим количеством треугольников с большей вероятностью демонстрируют структуры и поведение «маленького мира».

Цель алгоритма вычисления коэффициента кластеризации – измерить, насколько плотно кластеризована группа по сравнению с тем, насколько плотно она *могла бы быть* кластеризована. Алгоритм использует подсчет треугольников, который выдает отношение количества существующих треугольников к возможному количеству треугольников. Максимальное значение 1 указывает на клику, где каждый узел связан с каждым другим узлом.

Существует два типа коэффициентов кластеризации: *локальная кластеризация* и *глобальная кластеризация*.

Локальный коэффициент кластеризации

Локальный коэффициент кластеризации (local clustering coefficient) вершины – это вероятность того, что ее соседи также связаны. Вычисление этой вероятности включает в себя подсчет треугольников.

Коэффициент кластеризации вершины можно найти, умножив количество треугольников, проходящих через вершину, на два, а затем умножив его на максимальное число связей в группе, которое всегда является степенью этой вершины минус один. Примеры различных треугольников и коэффициентов кластеризации для вершины с пятью связями изображены на рис. 6.3.

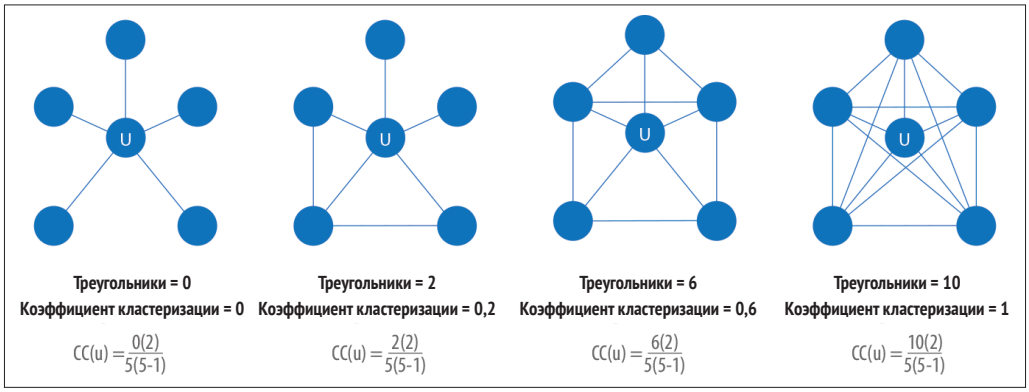


Рис. 6.3. Подсчеты треугольников и коэффициенты кластеризации для вершины *U*

Обратите внимание, что на рис. 6.3 мы используем вершину с пятью связями, из-за чего создается впечатление, что коэффициент кластеризации всегда будет равен 10 % от числа треугольников. Мы убеждаемся, что это не тот случай, когда меняем количество связей. Если мы изменим второй пример, чтобы иметь четыре связи (и те же два треугольника), то коэффициент будет равен 0,33.

Коэффициент кластеризации для узла использует формулу:

$$CC(U) = \frac{2R_U}{k_U\left(k_U - 1\right)},$$

где

U – вершина;

R_U – это число связей через соседей U (его можно получить, используя количество треугольников, проходящих через U);

k_U – степень U.

Глобальный коэффициент кластеризации

Глобальный коэффициент кластеризации (global clustering coefficient) является нормализованной суммой локальных коэффициентов кластеризации.

Коэффициенты кластеризации дают нам эффективный способ найти очевидные группы, такие как клики, где каждая вершина связана со всеми остальными вершинами, но мы также можем указать пороговые значения (скажем, где вершины связаны на 40 %).

Когда следует использовать подсчет треугольников и коэффициент кластеризации?

Используйте подсчет треугольников, когда вам нужно определить стабильность группы, или как часть расчета других сетевых показателей, таких как коэффициент кластеризации. Подсчет треугольников популярен в анализе социальных сетей, где он используется для обнаружения сообществ.

Коэффициент кластеризации может дать вам вероятность того, что случайно выбранные узлы будут связаны. Вы также можете использовать его для быстрой оценки сплоченности конкретной группы или вашей сети в целом. Вместе эти алгоритмы используются для оценки отказоустойчивости и поиска сетевых структур.

Примеры использования упомянутых алгоритмов включают в себя:

- определение признаков для классификации данного веб-сайта как носителя спам-контента. Этот подход описан в статье[2] Л. Беккетти и др. «Эффективные алгоритмы полупотоков для локального подсчета треугольников в массивных графах»;
- изучение структуры сообщества социального графа Facebook, где исследователи обнаружили возникновение плотных окружений соседствующих пользователей в разреженном глобальном графе. Это исследование представлено в статье[3] Дж. Угандер и др. «Анатомия социального графа Facebook»;
- изучение тематической структуры сети и выявление сообществ страниц с общими темами на основе взаимных ссылок между ними. Для получения дополнительной информации см. статью[4] «Искривление кросс-ссылок выявляет скрытые тематические слои во Всемирной паутине», J.-P. Эккманн и Э. Мозес.

Реализация подсчета треугольников с Apache Spark

Теперь мы готовы выполнить алгоритм подсчета треугольников. Мы можем использовать следующий код:

```
result = g.triangleCount()
(result.sort("count", ascending=False)
 .filter('count > 0')
 .show())
```

Запустив этот код, мы получим следующий результат:

---

[2] *L. Becchetti et al.*, «Efficient Semi-Streaming Algorithms for Local Triangle Counting in Massive Graphs», http://bit.ly/2ut0Lao.

[3] *J. Ugander et al.*, «The Anatomy of the Facebook Social Graph», https://bit.ly/2TXWsTC.

[4] *J.-P. Eckmann and E. Moses*, «Curvature of Co-Links Uncovers Hidden Thematic Layers in the World Wide Web», http://bit.ly/2YkCrFo.

count	id
1	jupyter
1	python-dateutil
1	six
1	ipykernel
1	matplotlib
1	jpy-console

Треугольник на этом графе будет означать, что два соседа вершины также являются соседями относительно друг друга. Шесть наших библиотек задействованы в таких треугольниках.

Что, если мы хотим узнать, какие узлы находятся в этих треугольниках? Вот где начинается *поток треугольников*. Для этого нам нужен Neo4j.

Реализация подсчета треугольников с Neo4j

Получение потока треугольников недоступно с помощью Spark, но мы можем вернуть его с помощью процедуры Neo4j:

```
CALL algo.triangle.stream("Library","DEPENDS_ON")
YIELD nodeA, nodeB, nodeC
RETURN algo.getNodeById(nodeA).id AS nodeA,
       algo.getNodeById(nodeB).id AS nodeB,
       algo.getNodeById(nodeC).id AS nodeC
```

Выполнив эту процедуру, мы получим:

nodeA	nodeB	nodeC
matplotlib	six	python-dateutil
jupyter	jpy-console	ipykernel

Мы видим те же шесть библиотек, что и раньше, но теперь мы знаем, как они связаны. matplotlib, six и python-dateutil образуют один треугольник. jupyter, jpy-console и ipykernel образуют другой треугольник. Эти треугольники изображены на рис. 6.4.

Локальный коэффициент кластеризации с Neo4j

Мы также можем определить локальный коэффициент кластеризации. Следующий запрос вычислит этот коэффициент для каждой вершины:

```
CALL algo.triangleCount.stream('Library', 'DEPENDS_ON')
YIELD nodeId, triangles, coefficient
WHERE coefficient > 0
```

```
RETURN algo.getNodeById(nodeId).id AS library, coefficient
ORDER BY coefficient DESC
```

Выполнение этой процедуры дает такой результат:

library	coefficient
ipykernel	1.0
jupyter	0.3333333333333333
jpy-console	0.3333333333333333
six	0.3333333333333333
python-dateutil	0.3333333333333333
matplotlib	0.16666666666666666

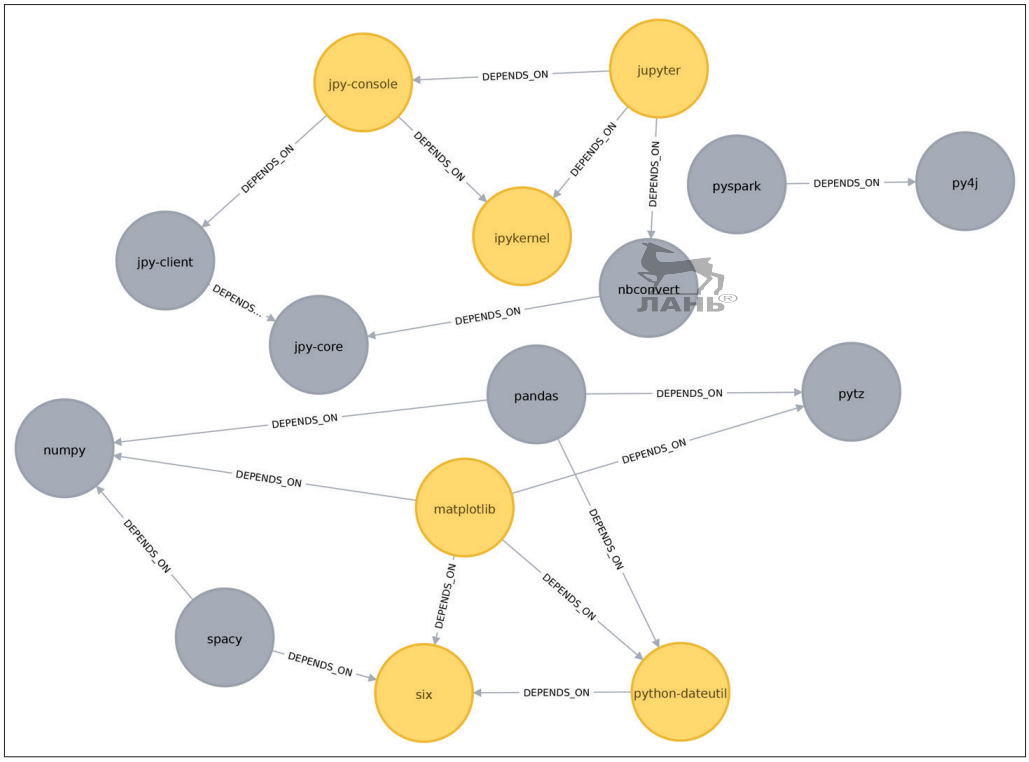


Рис. 6.4. Треугольники зависимостей программного обеспечения

Библиотека ipykernel имеет оценку 1, т. е. все соседи ipykernel являются соседями друг друга. Мы можем ясно видеть это на рис. 6.4. Это говорит нам о том, что сообщество непосредственно вокруг ipykernel очень сплоченное.

В этом примере кода мы отфильтровали вершины с коэффициентом 0, но иногда вершины с низкими коэффициентами также могут быть интересны. Низкий балл может быть показателем того, что вершина является *структурной дырой* (structural hole) – вершиной, хорошо связанной с вершинами в разных сообществах, которые иначе не связаны друг с другом. Это метод поиска *возможных* мостов, который мы обсуждали в главе 5.

Сильно связанные компоненты

Алгоритм сильно связанных компонентов (strongly connected components, SCC) является одним из самых ранних графовых алгоритмов. SCC находит наборы связанных вершин в ориентированном графе, где каждая вершина доступна в обоих направлениях от любой другой вершины в том же наборе. Время выполнения этого алгоритма хорошо масштабируется пропорционально количеству вершин. На рис. 6.5 вы можете видеть, что вершины в группе SCC не должны быть непосредственными соседями, но между всеми вершинами в наборе должны быть направленные пути.

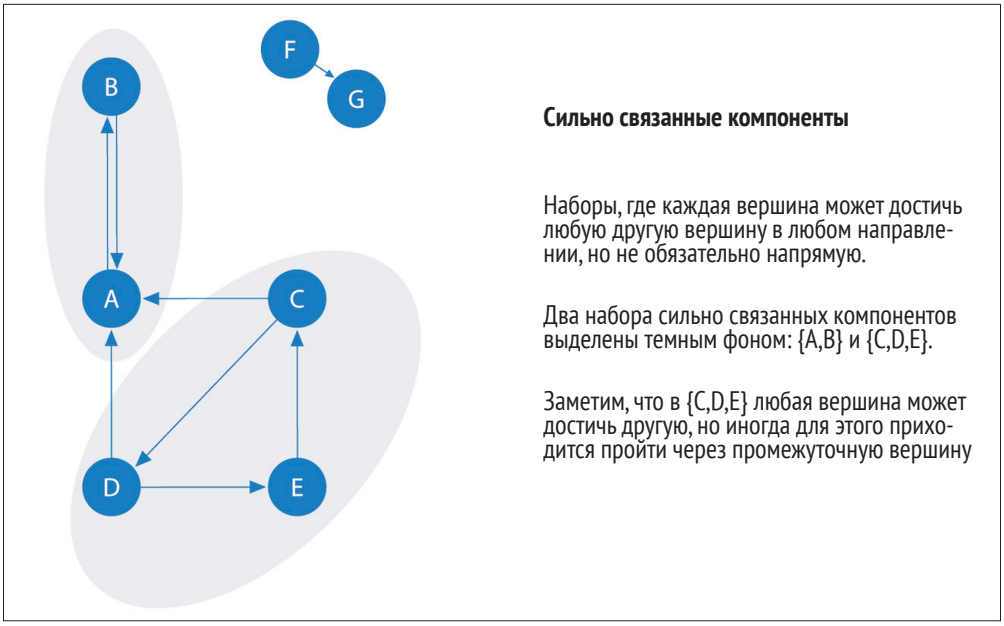


Рис. 6.5. Сильно связанные компоненты

Разложение ориентированного графа на сильно связанные компоненты является классическим применением алгоритма поиска в глубину. Neo4j использует DFS у себя под капотом как часть своей реализации алгоритма SCC.

Когда следует использовать сильно связанные компоненты?

Используйте сильно связанные компоненты в качестве начального этапа анализа графа, чтобы увидеть его структуру или определить прочные кластеры, которые могут потребовать независимого исследования. Алгоритм сильно связанных компонентов может применяться для выявления схожего поведения или склонностей в группе для таких приложений, как рекомендательные механизмы.

Многие алгоритмы выделения сообществ, такие как SCC, используются для поиска и объединения кластеров в отдельные вершины для последующего межкластерного анализа. Вы также можете использовать SCC для визуализации циклов при анализе, например для нахождения процессов, которые могут зайти в тупик, потому что каждый подпроцесс ожидает действий другого участника.

Примеры использования SCC включают в себя:

- поиск группы фирм, в которых каждый участник прямо и/или косвенно владеет акциями каждого другого участника, как, например, в «Сети глобального корпоративного контроля» – анализе[5] влиятельных транснациональных корпораций, выполненном С. Витали, Дж. Б. Глаттфельдером и С. Баттисто;
- вычисление связности различных конфигураций сети при измерении качества маршрутизации в беспроводных сетях с несколькими переходами. Подробнее читайте в статье[6] «Качество маршрутизации при наличии однонаправленных соединений в беспроводных сетях с несколькими переходами», М. К. Марина и С. Р. Дас;
- первый шаг во многих графовых алгоритмах, которые работают только на сильно связанных графах. В социальных сетях мы находим много сильно связанных групп. В этих группах люди часто имеют схожие предпочтения, и алгоритм SCC используется для поиска таких групп и рекомендации страниц, которые им могут понравиться, или товаров для покупки тем членам группы, которые еще этого не сделали.

У некоторых алгоритмов есть стратегии для выхода из бесконечных циклов, но, если мы пишем свои собственные алгоритмы или ищем не завершающиеся процессы, можем использовать для проверки циклов алгоритм SCC.

[5] *S. Vitali, J. B. Glattfelder, and S. Battiston*, «The Network of Global Corporate Control», http://bit.ly/2UU4EAP.

[6] *M. K. Marina and S. R. Das*, «Routing Performance in the Presence of Unidirectional Links in Multihop Wireless Networks», https://bit.ly/2uAJs7H.

Реализация поиска сильно связанных компонентов с Apache Spark

Начиная работу с Apache Spark, мы сначала импортируем нужные нам пакеты из Spark и пакета GraphFrames:

```
from graphframes import *
from pyspark.sql import functions as F
```


Теперь мы готовы выполнить алгоритм сильно связанных компонентов. Будем использовать его, чтобы выяснить, есть ли в нашем графе цикличес-кие связи.

Напишем код, выполняющий эту работу:

```
result = g.stronglyConnectedComponents(maxIter=10)
(result.sort("component")
.groupby("component")
.agg(F.collect_list("id").alias("libraries"))
.show(truncate=False))
```

Выполнив этот код в pyspark, мы получим такой результат:

component	libraries
180388626432	[jpy-core]
223338299392	[spacy]
498216206336	[numpy]
523986010112	[six]
549755813888	[pandas]
558345748480	[nbconvert]
661424963584	[ipykernel]
721554505728	[jupyter]
764504178688	[jpy-client]
833223655424	[pytz]
910533066752	[python-dateutil]
936302870528	[pyspark]
944892805120	[matplotlib]
1099511627776	[jpy-console]
1279900254208	[py4j]

Вы можете заметить, что каждой вершине библиотеки присвоен уни-кальный компонент. Это раздел или подгруппа, к которой она принадле-жит, и, как мы ожидали, каждая вершина находится в своем собственном

разделе. Это означает, что наш программный проект не имеет циклических зависимостей между библиотеками.

Две вершины могут находиться в одном и том же сильно связанном компоненте, если между ними есть пути в обоих направлениях.

Реализация поиска сильно связанных компонентов с Neo4j

Давайте выполним тот же алгоритм, используя Neo4j. Выполните следующий запрос для запуска алгоритма:

```
CALL algo.scc.stream("Library", "DEPENDS_ON")
YIELD nodeId, partition
RETURN partition, collect(algo.getNodeById(nodeId)) AS libraries
ORDER BY size(libraries) DESC
```

Мы передаем в алгоритм два параметра:

- `library` – метка вершины для чтения из графа;
- `DEPENSD_ON` – тип связи для чтения из графа.

После выполнения запроса мы получим результат:

partition	libraries
8	[ipykernel]
11	[six]
2	[matplotlib]
5	[jupyter]
14	[python-dateutil]
13	[numpy]
4	[py4j]
7	[nbconvert]
1	[pyspark]
10	[jpy-core]
9	[jpy-client]
3	[spacy]
12	[pandas]
6	[jpy-console]
0	[pytz]

Как и в примере с Spark, каждая вершина находится в своем собственном разделе.

Алгоритм продемонстрировал, что наши библиотеки Python ведут себя очень хорошо, но давайте создадим циклическую зависимость в графе, чтобы ситуация стала интереснее. Это будет означать, что в одном разделе окажется несколько вершин.

Следующий запрос добавляет в граф дополнительную библиотеку, которая создает циклическую зависимость между py4j и pyspark:

```
MATCH (py4j:Library {id: "py4j"})
MATCH (pyspark:Library {id: "pyspark"})
MERGE (extra:Library {id: "extra"})
MERGE (py4j)-[:DEPENDS_ON]->(extra)
MERGE (extra)-[:DEPENDS_ON]->(pyspark)
```

На рис. 6.6. ясно видна образовавшаяся циклическая зависимость.

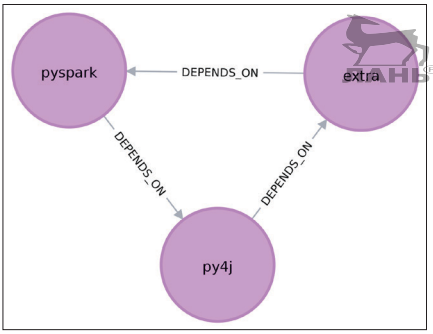


Рис. 6.6. Циклическая зависимость между pyspark, py4j и extra

Если мы снова запустим алгоритм SCC, то увидим немного другой результат:

partition	libraries
1	[pyspark, py4j, extra]
8	[ipykernel]
11	[six]
2	[matplotlib]
5	[jupyter]
14	[numpy]
13	[pandas]
7	[nbconvert]
10	[jpy-core]
9	[jpy-client]
3	[spacy]
15	[python-dateutil]
6	[jpy-console]
0	[pytz]

Библиотеки pyspark, py4j и extra входят в один раздел, и алгоритм SCC помог нам найти циклическую зависимость!

Прежде чем перейти к следующему алгоритму, удалим лишнюю библиотеку и ее взаимосвязи из графа:

```
MATCH (extra:Library {id: "extra"})
DETACH DELETE extra
```

Связанные компоненты

Алгоритм связанных компонентов (иногда называемых *объединением компонентов* или *слабо связанными компонентами*) находит наборы связанных вершин в ненаправленном графе, где каждая вершина доступна из любой другой вершины в том же наборе. Он отличается от алгоритма SCC, потому что ему достаточно связи между парами вершин в одном направлении, тогда как SCC нуждается в обоих направлениях. Бернард А. Галлер и Майкл Дж. Фишер впервые описали этот алгоритм в своей статье[7] 1964 года «Улучшенный алгоритм эквивалентности».

Когда следует использовать связанные компоненты?

Как и в случае с SCC, алгоритм связанных компонентов часто применяется на ранних этапах анализа для понимания структуры графа. Благодаря эффективному масштабированию этот алгоритм хорошо подходит для графов, требующих частых обновлений. Он может быстро показать новые общие вершины в составе группы, что полезно для анализа, такого как обнаружение мошенничества.

Возьмите за привычку запускать алгоритм связанных компонентов в качестве подготовительного шага для общего анализа графа. Выполнение этого быстрого теста позволяет избежать случайного запуска алгоритмов только для одного несвязанного компонента графа и получения неверных результатов.

Примеры использования алгоритма связанных компонентов включают в себя:

- отслеживание кластеров записей базы данных как часть процесса *дедупликации*. Дедупликация является важной задачей в приложениях управления данными; этот подход более подробно описан в статье[8] А. Монжа и К. Элкана «Эффективный предметно-независимый алгоритм обнаружения приблизительно повторяющихся записей базы данных»;
- анализ сетей цитирования. В одном из исследований алгоритм связанных компонентов используется, для того чтобы выяснить, насколько хорошо связана сеть, а затем посмотреть, сохраняется ли связность, если вершины типа «концентратор» или «авторитет» удалены из графа. Этот вариант использования подробно описан в ста-

[7] *Bernard A. Galler, Michael J. Fischer*, «An Improved Equivalence Algorithm», https://bit.ly/2WsPNxT.

[8] *A. Monge, C. Elkan*, «An Efficient Domain-Independent Algorithm for Detecting Approximately Duplicate Database Records», http://bit.ly/2CCNpgy.

тье[9] Й. Ан, Дж. С. М. Янссена и Э. Э. Милиоса «Извлечение и исследование графа цитирования из публикаций в области информатики».

Реализация алгоритма связанных компонентов с Apache Spark

Начиная работу с Apache Spark, сначала импортируем нужные нам пакеты из Spark и пакета GraphFrames:

```
from pyspark.sql import functions as F
```

Две вершины могут находиться в одном и том же сильно связанном компоненте, если между ними есть связь в любом направлении.

Теперь мы готовы выполнить алгоритм связанных компонентов при помощи следующего кода:

```
result = g.connectedComponents()
(result.sort("component")
 .groupby("component")
 .agg(F.collect_list("id").alias("libraries"))
 .show(truncate=False))
```

Если мы запустим этот код в pyspark, то получим следующий результат:

component	libraries
180388626432	[jpy-core, nbconvert, ipykernel, jupyter, jpy-client, jpy-console]
223338299392	[spacy, numpy, six, pandas, pytz, python-dateutil, matplotlib]
936302870528	[pyspark, py4j]

Результаты показывают три кластера вершин, которые также можно увидеть на рис. 6.7. Впрочем, в данном примере для обнаружения компонентов достаточно визуального осмотра. Этот алгоритм намного более полезен на больших графах, где визуальный осмотр невозможен или занимает много времени.

Реализация алгоритма связанных компонентов с Neo4j

Мы также можем запустить алгоритм связанных компонентов в Neo4j, выполнив следующий запрос:

[9] *Y. An, J. C. M. Janssen, E. E. Milios*, «Characterizing and Mining Citation Graph of Computer Science Literature», https://bit.ly/2U8cfi9.

```
CALL algo.unionFind.stream("Library", "DEPENDS_ON")
YIELD nodeId,setId
RETURN setId, collect(algo.getNodeById(nodeId)) AS libraries
ORDER BY size(libraries) DESC
```

Мы передаем в алгоритм два параметра:

- `Library` – имя вершины, загружаемой из графа;
- `DEPENDS_ON` – тип связи, загружаемой из графа.

Выход выглядит следующим образом:

setId	libraries
2	[pytz, matplotlib, spacy, six, pandas, numpy, python-dateutil]
5	[jupyter, jpy-console, nbconvert, ipykernel, jpy-client, jpy-core]
1	[pyspark, py4j]

Как и ожидалось, мы получаем точно такие же результаты, как и со Spark.

Оба алгоритма выявления сообществ, которые мы рассмотрели до сих пор, являются *детерминированными*: они возвращают одинаковые результаты при каждом их запуске. Следующие два алгоритма являются примерами недетерминированных алгоритмов, где мы можем увидеть разные результаты, если запустим их несколько раз, даже на одних и тех же данных.

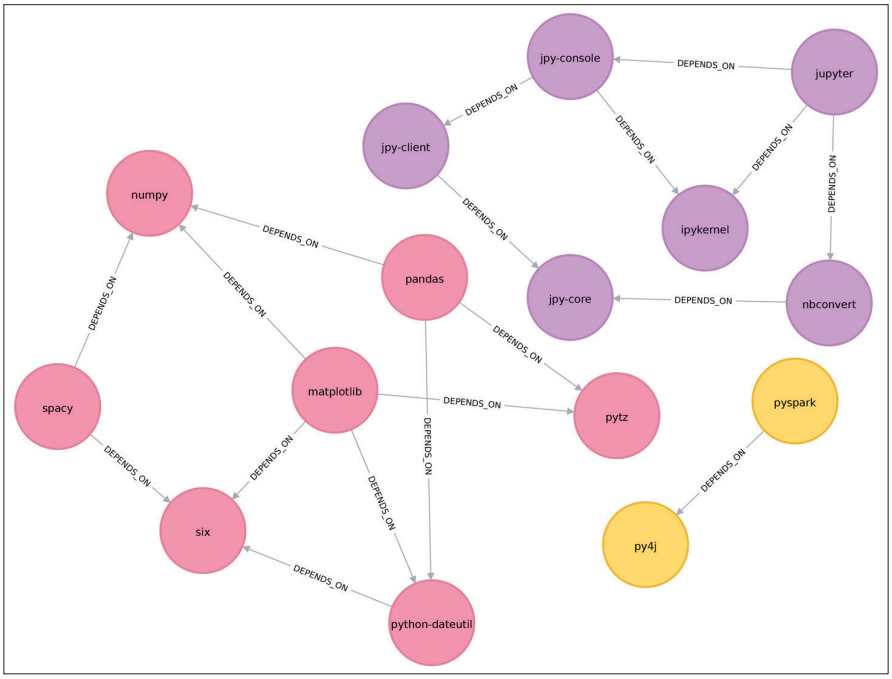


Рис. 6.7. Кластеры, найденные алгоритмом связанных компонентов

Алгоритм распространения меток

Алгоритм распространения меток (label propagation algorithm, LPA) – это быстрый алгоритм поиска сообществ в графе. В LPA вершины выбирают принадлежность к группе на основе связей с прямыми соседями. Этот процесс хорошо подходит для сетей, где группировки менее ясны, и, чтобы помочь вершине определить, в какое сообщество следует поместить себя, могут использоваться веса ребер. Кроме того, LPA хорошо подходит для обучения с частичным привлечением учителя, потому что вы можете начать процесс с предварительно назначенных, ориентировочных меток вершин.

Идея, лежащая в основе этого алгоритма, состоит в том, что одиночная метка может быстро стать доминирующей в плотно связанной группе вершин, но у нее будут проблемы с пересечением слабо связанной области. Метки «застревают» в плотно связанной группе вершин, а вершины, помеченные одинаковыми метками после завершения алгоритма, считаются частью одного и того же сообщества. Алгоритм допускает *перекрытия*, когда вершины потенциально являются частью нескольких кластеров, назначая принадлежность той окрестности метки, которая дает наибольшее совокупное значение связности и веса вершины. LPA – это сравнительно новый алгоритм, предложенный в 2007 году У. Н. Рагхаваном, Р. Альбертом и С. Кумарой в статье[10] под названием «Почти линейный во времени алгоритм для обнаружения сообществ в крупномасштабных сетях».

На рис. 6.8 показаны два варианта распространения меток: простой *метод загрузки* (push method) и более типичный *метод извлечения* (pull method), основанный на весах ребер. Метод извлечения хорошо подходит для распараллеливания.

Шаги, часто используемые для метода извлечения:

1. Каждая вершина инициализируется уникальной меткой (идентификатором), и при желании могут использоваться предварительные «начальные» метки.
2. Эти метки распространяются по графу.
3. На каждой итерации распространения каждая вершина обновляет свою метку, чтобы она соответствовала метке с максимальным весом, который рассчитывается на основе весов соседних вершин и их связей.
4. LPA достигает конвергенции, когда каждая вершина имеет метку большинства своих соседей.

По мере распространения меток плотно связанные группы вершин быстро достигают консенсуса по уникальной метке. По окончании распро-

[10] *U. N. Raghavan, R. Albert, S. Kumara*, «Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks», https://bit.ly/2Frb1Fu.

странения останется только несколько меток, а вершины с одинаковыми метками принадлежат одному сообществу.

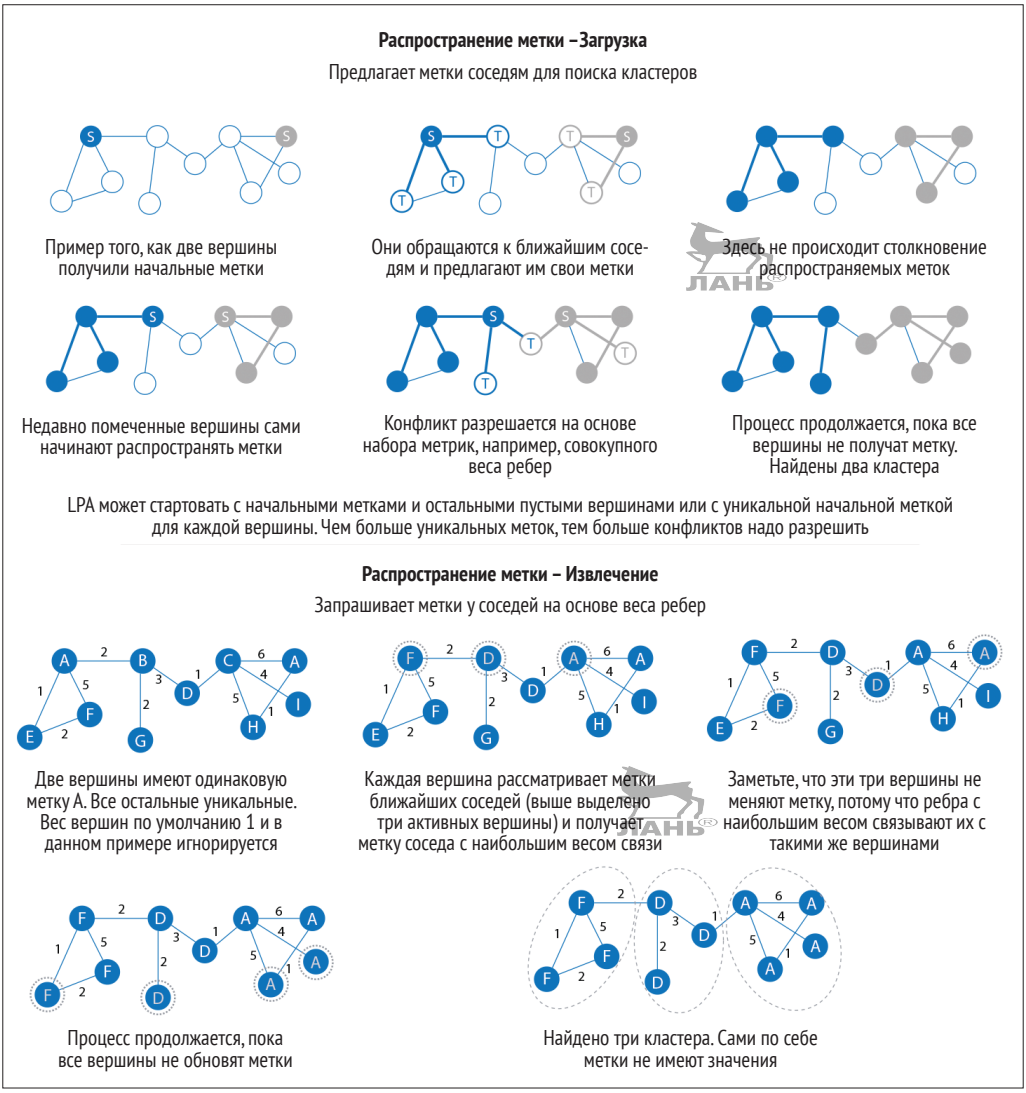


Рис. 6.8. Два варианта алгоритма LPA

Обучение с частичным привлечением учителя и начальные метки

В отличие от других алгоритмов распространение меток может возвращать различные структуры сообщества при многократном запуске на одном графе. Порядок, в котором LPA оценивает вершины, может влиять на конечные сообщества, которые он возвращает.

Диапазон решений сужается, когда некоторым вершинам присваиваются *предварительные метки* (т. е. *начальные метки*, seed labels), в то время как другие вершины не имеют меток. Немаркированные вершины с большей вероятностью примут начальные метки.

Такое использование распространения меток можно рассматривать как метод *обучения с частичным привлечением учителя* (semi-supervised learning) для поиска сообществ. Обучение с частичным привлечением учителя – это класс задач и методов машинного обучения, которые работают с небольшим количеством размеченных данных, а также с большим объемом неразмеченных данных. Мы также можем многократно запускать алгоритм на графах по мере их развития.

Наконец, LPA иногда не сходится на одном решении. В этой ситуации алгоритм будет постоянно метаться между несколькими похожими сообществами и никогда не завершится. Начальные метки помогают направить алгоритм к единственному решению. Spark и Neo4j используют заданное максимальное количество итераций, чтобы избежать бесконечного выполнения. Вам следует проверить и правильно установить настройки итерации для ваших данных, чтобы найти баланс между точностью и временем выполнения.

Когда следует использовать распространение меток?

Используйте распространение меток в крупных сетях для первоначального обнаружения сообщества, особенно при наличии весов. Этот алгоритм может быть распараллелен и поэтому чрезвычайно быстро выполняет анализ графа.

Примеры использования алгоритма распространения меток включают в себя:

- определение полярности твитов как часть семантического анализа. В этом сценарии в сочетании с графом подписчика в Твиттере используются положительные и отрицательные начальные метки из классификатора. Для получения дополнительной информации обратитесь к статье[11] «Классификация полярности в Твиттере с распространением меток по лексическим ссылкам и графу подписчика», авторы М. Спериозу и др.;
- поиск потенциально опасных комбинаций совместно назначаемых препаратов на основе профиля химического сходства и побочных эффектов. Пример такого исследования описан в статье[12] П. Чжан и др. «Прогнозирование распространения меток в межлекарственном взаимодействии на основе клинических побочных эффектов»;

[11] *M. Speriosu et al.*, «Twitter Polarity Classification with Label Propagation over Lexical Links and the Follower Graph», https://bit.ly/2FBq2pv.

[12] *P. Zhang et al.*, «Label Propagation Prediction of Drug–Drug Interactions Based on Clinical Side Effects», https://www.nature.com/articles/srep12339.

- определение признаков диалога и намерений пользователя для модели машинного обучения. Для получения дополнительной информации см. статью[13] Ю. Мурасе и др. «Определение признаков на основе распространения метки на графе данных Wiki для DST».

Реализация алгоритма распространения меток с Apache Spark

Начиная работу с Apache Spark, сначала импортируем нужные нам пакеты из Spark и пакета GraphFrames:

```
from pyspark.sql import functions as F
```

Теперь мы готовы выполнить алгоритм распространения меток при помощи следующего кода:

```
result = g.labelPropagation(maxIter=10)
(result
 .sort("label")
 .groupby("label")
 .agg(F.collect_list("id"))
 .show(truncate=False))
```

Запустив этот код в pyspark, мы получим следующий результат:

label	collect_list(id)
180388626432	[jpy-core, jpy-console, jupyter]
223338299392	[matplotlib, spacy]
498216206336	[python-dateutil, numpy, six, pytz]
549755813888	[pandas]
558345748480	[nbconvert, ipykernel, jpy-client]
936302870528	[pyspark]
1279900254208	[py4j]

По сравнению с алгоритмом связанных компонентов в этом примере у нас больше кластеров библиотек. LPA менее строг, чем связанные компоненты, в отношении того, как он определяет кластеры. При использовании распространения метки два соседа (напрямую соединенные вершины) могут очутиться в разных кластерах. Однако при использовании связанных компонентов вершина всегда будет находиться в том же кластере, что и ее соседи, потому что этот алгоритм основан на строгой группировке отношений.

[13] *Y. Murase et al.*, «Feature Inference Based on Label Propagation on Wikidata Graph for DST», https://bit.ly/2FtGpTK.

В нашем примере наиболее очевидным отличием является то, что библиотеки Jupyter были разделены на два сообщества: одно содержит основные части библиотеки, а другое – инструменты для работы с клиентами.

Реализация алгоритма распространения меток с Neo4j

Теперь давайте попробуем запустить тот же алгоритм на платформе Neo4j, выполнив следующий запрос:

```
CALL algo.labelPropagation.stream("Library", "DEPENDS_ON",
  { iterations: 10 })
YIELD nodeId, label
RETURN label,
       collect(algo.getNodeById(nodeId).id) AS libraries
ORDER BY size(libraries) DESC
```

Мы передаем в алгоритм следующие параметры:

- `Library` – имя вершины, загружаемой из графа;
- `DEPENDS_ON` – тип связи, загружаемой из графа;
- `iterations:10` – максимальное количество итераций,

и получаем следующий результат:

label	libraries
11	[matplotlib, spacy, six, pandas, python-dateutil]
10	[jupyter, jpy-console, nbconvert, jpy-client, jpy-core]
4	[pyspark, py4j]
8	[ipykernel]
13	[numpy]
0	[pytz]

Результаты, которые графически представлены на рис. 6.9, довольно похожи на те, которые мы получили с помощью Apache Spark.

Мы также можем запустить алгоритм, предполагая, что граф является ненаправленным, т. е. вершины будут пытаться принять метки как от зависимых библиотек, так и от тех библиотек, от которых зависят сами.

Для этого мы передаем алгоритму параметр `DIRECTION:BOTH`:

```
CALL algo.labelPropagation.stream("Library", "DEPENDS_ON",
  { iterations: 10, direction: "BOTH" })
YIELD nodeId, label
RETURN label,
       collect(algo.getNodeById(nodeId).id) AS libraries
ORDER BY size(libraries) DESC
```

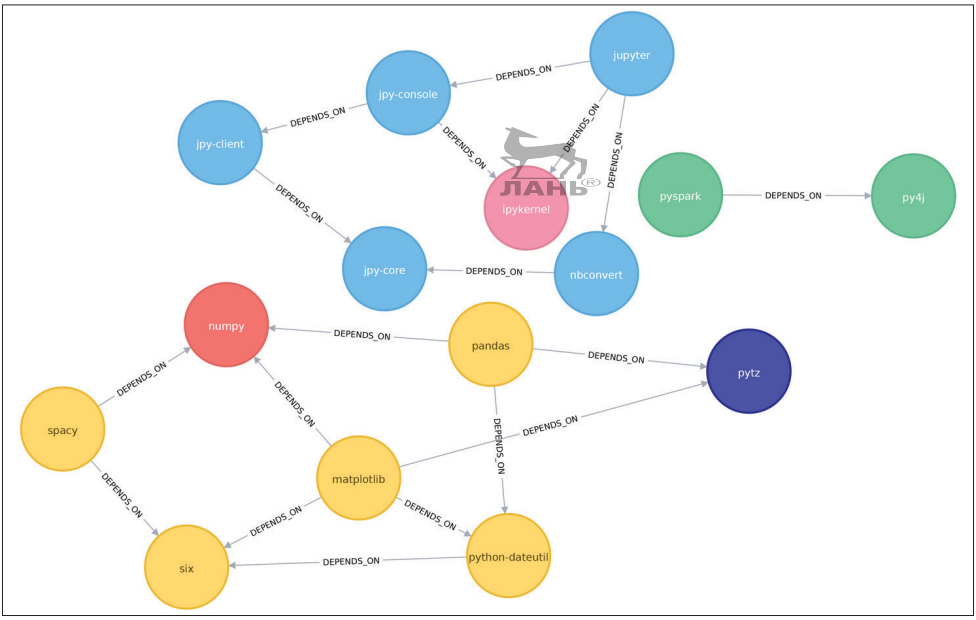


Рис. 6.9. Кластеры, обнаруженные алгоритмом LPA

Выполнив этот запрос, мы получим следующий результат:

label	libraries
11	[pytz, matplotlib, spacy, six, pandas, numpy, python-dateutil]
10	[nbconvert, jpy-client, jpy-core]
6	[jupyter, jpy-console, ipykernel]
4	[pyspark, py4j]

Количество кластеров уменьшилось с шести до четырех, и теперь все вершины вокруг библиотеки matplotlib сгруппированы вместе. Это хорошо видно на рис. 6.10.

Хотя на этих данных результаты выполнения LPA для ненаправленных и направленных вычислений очень похожи, на сложных графах вы увидите более существенные различия. Это потому, что игнорирование направления заставляет вершины пытаться принять больше меток, независимо от источника взаимосвязи.

Лувенский модульный алгоритм

Лувенский[14] *модульный алгоритм* (Louvian modularity) находит кластеры путем сравнения плотности сообщества, поскольку он относит вершины к различным группам. Вы можете рассматривать это как анализ «что, если»,

[14] По названию бельгийского города Лувен (Louvain), в котором трудились разработчики метода. – *Прим. перев.*

чтобы попробовать различные группировки с целью достижения глобального оптимума.

Предложенный в 2008 году Лувенский алгоритм является одним из самых быстрых модульных алгоритмов. Помимо выявления сообществ, он также выявляет иерархию сообществ в разных масштабах. Это полезно для понимания структуры сети на разных уровнях детализации.

Лувенский алгоритм количественно определяет, насколько хорошо узел сети соответствует группе, анализируя плотность связей в кластере по сравнению со средней или случайной выборкой. Эта мера принадлежности к сообществу называется модульностью.

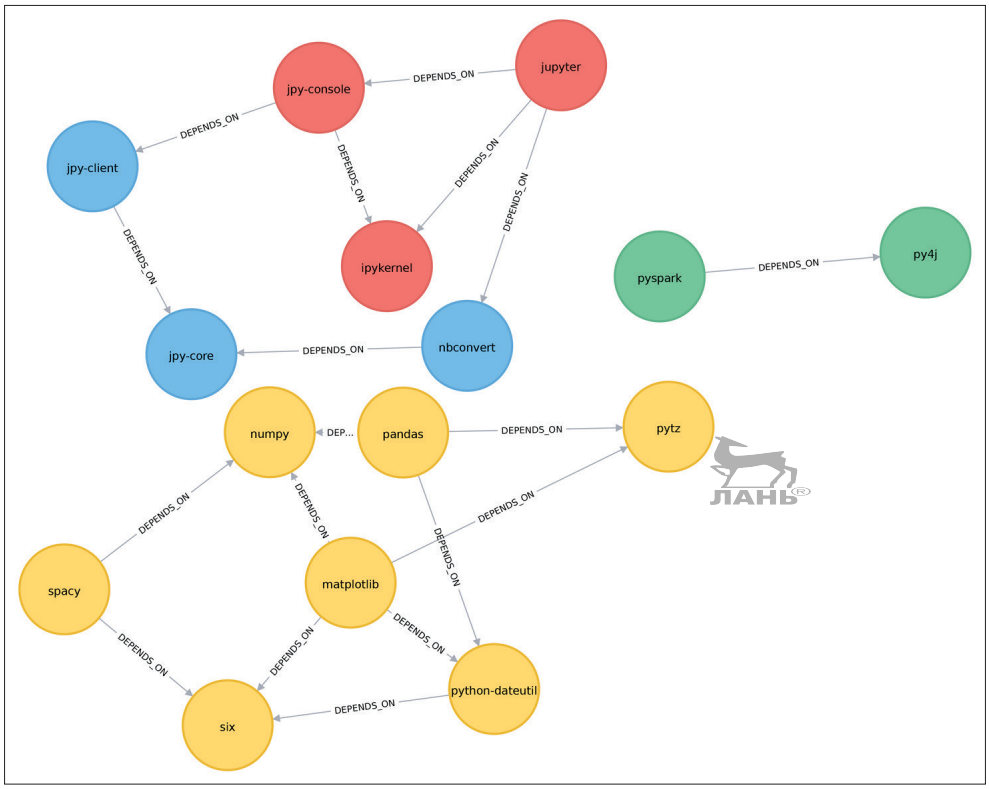


Рис. 6.10. Кластеры, обнаруженные алгоритмом LPA, если игнорировать направление связей

Группировка по качеству через модульность

Модульность – это техника выявления сообществ путем разделения графа на относительно крупные модули (или кластеры) и последующего измерения силы связывания группы. В отличие от простого анализа концентрации соединений внутри кластера этот метод сравнивает плотности отношений в данных кластерах с плотностями между кластерами. Мера качества этих группировок называется модульностью.

Вычисление модульности

Простой расчет модульности основан на доле связей внутри определенных групп минус ожидаемая доля, если бы отношения были распределены случайно между всеми вершинами. Значение всегда находится в диапазоне от 1 до –1, причем положительные значения указывают на бóльшую плотность отношений, чем вы можете ожидать от случайного распределения, а отрицательные значения указывают на меньшую плотность. Рисунок 6.11 иллюстрирует несколько различных показателей модульности, основанных на группировке вершин.

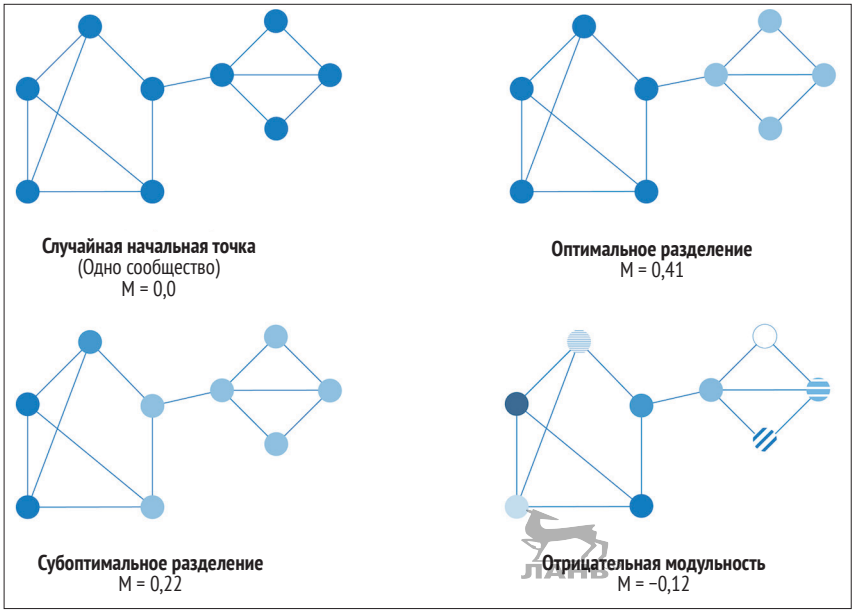


Рис. 6.11. Четыре оценки модульности, основанные на разном разделении

Формула для вычисления модульности выглядит следующим образом:

$$M = \sum_{C=1}^{n_c}\left[\frac{L_C}{L} - \left(\frac{k_C}{2L}\right)^2\right],$$

где

- L – количество связей в группе;
- L_C – количество связей в сегменте;
- k_C – полная степень вершин в сегменте.

Расчет для оптимального разбиения в верхней части рис. 6.11 выглядит следующим образом:

- Темная часть $\left(\frac{7}{13} - \left(\frac{15}{2(13)}\right)^2\right) = 0,205$
- Светлая часть $\left(\frac{5}{13} - \left(\frac{11}{2(13)}\right)^2\right) = 0,206$
- Сложив их вместе, получим M = 0,205 + 0,206 = 0,41

Алгоритмы модульности оптимизируют сообщества локально, а затем и глобально, используя многократные итерации для проверки различных группировок и увеличивая площадность охвата. Эта стратегия определяет иерархию сообществ и обеспечивает широкое понимание общей структуры. Однако все алгоритмы модульности имеют два недостатка:

- они объединяют небольшие сообщества в более крупные;
- когда присутствует несколько вариантов разделения с одинаковой модульностью, может возникать плато, образующее локальные максимумы и препятствующее дальнейшему движению.

Для получения дополнительной информации см. статью[15] «Качество максимизации модульности в практическом контексте», Б. Х. Гуд, Я.-А. де Монжуа и А. Клозе.

Сначала Лувенский алгоритм оптимизирует модульность локально на всех вершинах, что выявляет небольшие сообщества; затем каждое небольшое сообщество группируется в более крупный конгломерат-вершину, и первый шаг повторяется, пока мы не достигнем глобального оптимума.

Алгоритм состоит из повторяющегося применения двух шагов, как показано на рис. 6.12.

Шаги Лувенского алгоритма включают в себя:

1. «Жадное» присвоение вершин сообществам, благоприятствующее локальной оптимизации модульности.
2. Определение более *крупнозернистой сети* (coarse-grained network) на основе сообществ, найденных на первом этапе. Эта крупнозернистая сеть будет использоваться в следующей итерации алгоритма.

Эти два шага повторяются до тех пор, пока возможно очередное перераспределение сообществ, приводящее к увеличению модульности.

Частью первого шага оптимизации является оценка модульности группы. Для этого Лувенский алгоритм использует следующую формулу:

$$Q = \frac{1}{2m}\sum_{u,v}\left[A_{uv} - \frac{k_u k_v}{2m}\right]\delta\left(c_u, c_v\right),$$

где

u и v – вершины;

m – общий вес связи по всему графу ($2m$ - это общее значение нормализации в формулах модульности);

[15] *B. H. Good, Y.-A. de Montjoye, and A. Clauset*, «The Performance of Modularity Maximization in Practical Contexts», https://arxiv.org/abs/0910.0165.

$A_{uv} - \frac{k_u k_v}{2m}$ является силой связи между u и v по сравнению с тем, что мы ожидаем при случайном назначении (имеет тенденцию к усреднению) этих вершин;

- A_{uv} – вес связи между u и v;
- k_u – сумма весов связей для u;
- k_v – сумма весов связей для v;

$\delta(c_u, c_v)$ равно 1, если u и v назначены одному сообществу, и 0, если нет.

Другая часть этого первого шага оценивает изменение модульности, если вершина перемещается в другую группу. Лувенский алгоритм использует более сложный вариант показанной выше формулы, а затем определяет лучший вариант привязки к группе.

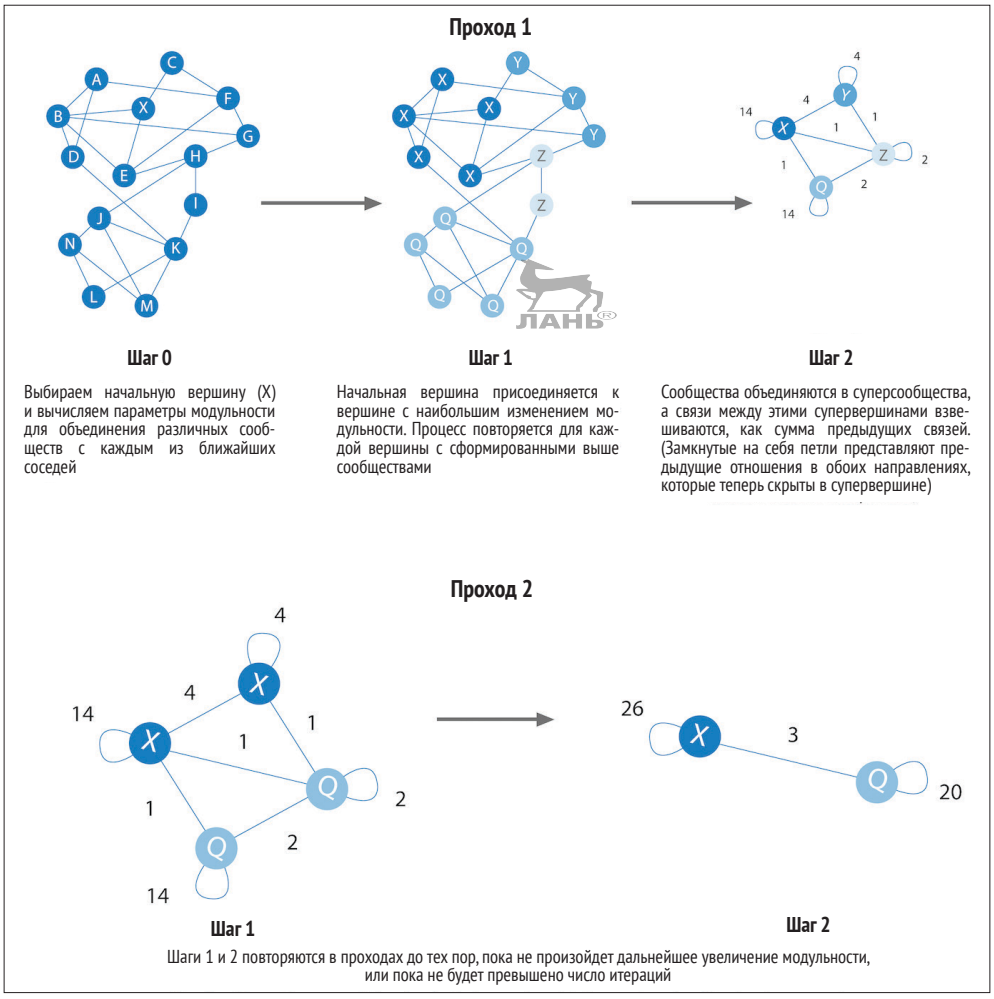


Рис. 6.12. Шаги Лувенского алгоритма

Когда следует использовать Лувенский алгоритм?

Используйте Лувенский алгоритм для поиска сообществ в обширных сетях. Этот алгоритм применяет эвристическую, а не точную модульность, требующую больших вычислительных ресурсов. Поэтому Лувенский алгоритм может применяться на больших графах, с которыми не справляются стандартные алгоритмы модульности.

Этот алгоритм также очень полезен для оценки структуры сложных сетей, в частности для выявления многих уровней иерархии, которые обычно присутствуют в преступной организации. Алгоритм может давать результаты, в которых вы можете увеличивать различные уровни детализации и находить буквально подсообщества внутри подсообществ внутри подсообществ.

Алгоритмы оптимизации модульности, включая Лувенский, страдают от двух проблем. Во-первых, алгоритмы могут пропускать небольшие сообщества в крупных сетях. Вы можете преодолеть эту проблему, просмотрев промежуточные этапы консолидации. Во-вторых, в больших графах с перекрывающимися сообществами оптимизаторы модульности могут некорректно определять глобальные максимумы. В последнем случае мы рекомендуем использовать любой алгоритм модульности в качестве подхода для грубой оценки, но не для достижения окончательной точности.

Примеры использования Лувенского алгоритма включают в себя:

- обнаружение кибератак. Алгоритм был использован в 2016 году в исследовании[16] С. В. Шанбака по быстрому обнаружению сообществ в крупномасштабных киберсетях для приложений кибербезопасности. После обнаружения этих сообществ их можно использовать для обнаружения кибератак;
- извлечение тем из онлайновых социальных платформ, таких как Twitter и YouTube, на основе совместного использования терминов в документах в рамках процесса моделирования тем. Этот подход описан в статье[17] Дж. С. Кидо, Р. А. Игавы и С. Барбона «Моделирование тем на основе Лувенского метода в онлайновых социальных сетях»;

[16] *S. V. Shanbhaq*, «A faster version of Louvain method for community detection for efficient modeling and analytics of cyber systems», https://bit.ly/2FAxaIS.

[17] *G. S. Kido, R. A. Igawa, and S. Barbon Jr.*, «Topic Modeling Based on Louvain Method in Online Social Networks», http://bit.ly/2UbCCUl.

- поиск иерархических структур сообщества в функциональной сети мозга, как описано в статье[18] Д. Менье и др. «Иерархическая модульность в функциональных сетях мозга человека».

Реализация Лувенского алгоритма с Neo4j

Давайте посмотрим Лувенский алгоритм в действии, выполнив следующий запрос на нашем графе:

```
CALL algo.louvain.stream("Library", "DEPENDS_ON")
YIELD nodeId, communities
RETURN algo.getNodeById(nodeId).id AS libraries, communities
```

Мы задаем алгоритму следующие параметры:

- `Library` – имя вершины, загружаемой из графа;
- `DEPENDS_ON` – тип связи, загружаемой из графа.

Результат выглядит следующим образом:

libraries	communities
pytz	[0, 0]
pyspark	[1, 1]
matplotlib	[2, 0]
spacy	[2, 0]
py4j	[1, 1]
jupyter	[3, 2]
jpy-console	[3, 2]
nbconvert	[4, 2]
ipykernel	[3, 2]
jpy-client	[4, 2]
jpy-core	[4, 2]
six	[2, 0]
pandas	[0, 0]
numpy	[2, 0]
python-dateutil	[2, 0]

Колонка `communities` описывает сообщества на двух уровнях, в которые попадают вершины. Последнее значение в массиве – это окончательное сообщество, а предыдущее значение – промежуточное сообщество.

[18] *D. Meunier et al.*, «Hierarchical Modularity in Human Brain Functional Networks», https://bit.ly/2HFHXxu.

Числа, присвоенные промежуточному и окончательному сообществам, являются просто метками без измеримого значения. Относитесь к ним как к меткам, которые указывают, к каким сообществам принадлежат вершины, т. е. «принадлежит сообществу с меткой 0», «сообществу с меткой 4» и т. д.

Например, результат matplotlib равен [2,0]. Это означает, что конечное сообщество matplotlib помечено числом 0, а его промежуточное сообщество – числом 2.

Нам будет проще увидеть, как это работает, если мы сохраним эти сообщества, используя записывающую версию алгоритма, а затем рассмотрим запись. Следующий запрос запустит Лувенский алгоритм и сохранит результат в свойстве `community` на каждой вершине:

```
CALL algo.louvain("Library", "DEPENDS_ON")
```

Мы также можем сохранить полученные сообщества, используя потоковую версию алгоритма, а затем использовать оператор SET для сохранения результата. Следующий запрос показывает, как это можно сделать:

```
CALL algo.louvain.stream("Library", "DEPENDS_ON")
YIELD nodeId, communities
WITH algo.getNodeById(nodeId) AS node, communities
SET node.communities = communities
```

Выполнив любой из этих запросов, мы можем затем написать следующий, чтобы найти окончательные кластеры:

```
MATCH (l:Library)
RETURN l.communities[-1] AS community, collect(l.id) AS libraries
ORDER BY size(libraries) DESC
```

`l.communities[-1]` возвращает последний элемент из массива, который хранит это свойство.

Запрос возвращает следующие выходные данные:

community	librarires
0	[pytz, matplotlib, spacy, six, pandas, numpy, python-dateutil]
2	[jupyter, jpy-console, nbconvert, ipykernel, jpy-client, jpy-core]
1	[pyspark, py4j]

Эта кластеризация такая же, как мы видели в алгоритме связанных компонентов.

На рис. 6.13 ясно видно, что matplotlib находится в сообществе с pytz, spacy, six, pandas, numpy и pythondateutil.

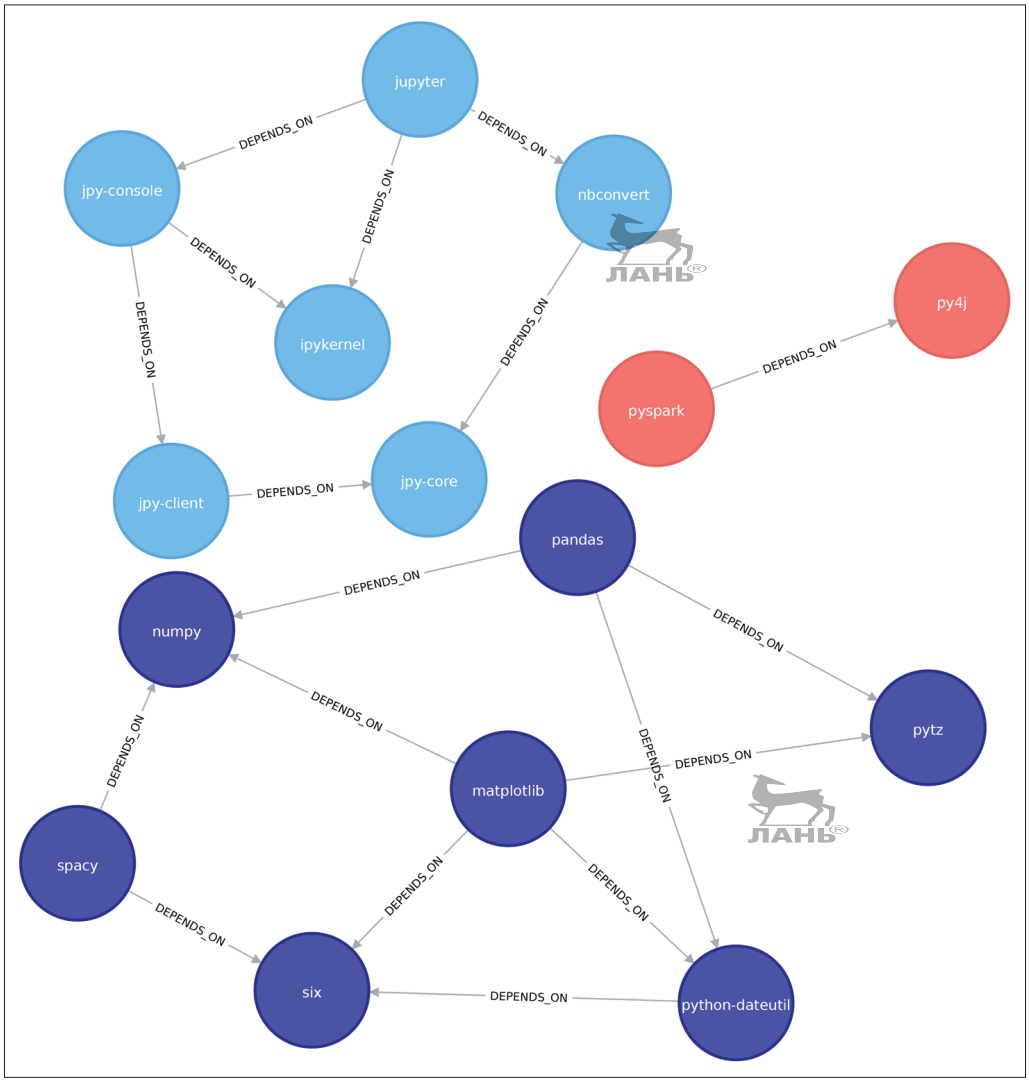


Рис. 6.13. Кластеры, найденные Лувенским алгоритмом

Дополнительной особенностью Лувенского алгоритма является то, что мы можем видеть и промежуточную кластеризацию. Это покажет нам более мелкозернистые кластеры, чем в последнем слое:

```
MATCH (l:Library)
RETURN l.communities[0] AS community, collect(l.id) AS libraries
ORDER BY size(libraries) DESC
```

Выполнение запроса дает следующие выходные данные:

community	libraries
2	[matplotlib, spacy, six, python-dateutil]
4	[nbconvert, jpy-client, jpy-core]
3	[jupyter, jpy-console, ipykernel]
1	[pyspark, py4j]
0	[pytz, pandas]
5	[numpy]

Библиотеки в сообществе matplotlib теперь разбиты на три небольших сообщества:

- matplotlib, spacy, six и python-dateutil;
- pitz и pandas;
- numpy.

Визуальное представление этой разбивки показано на рис. 6.14.

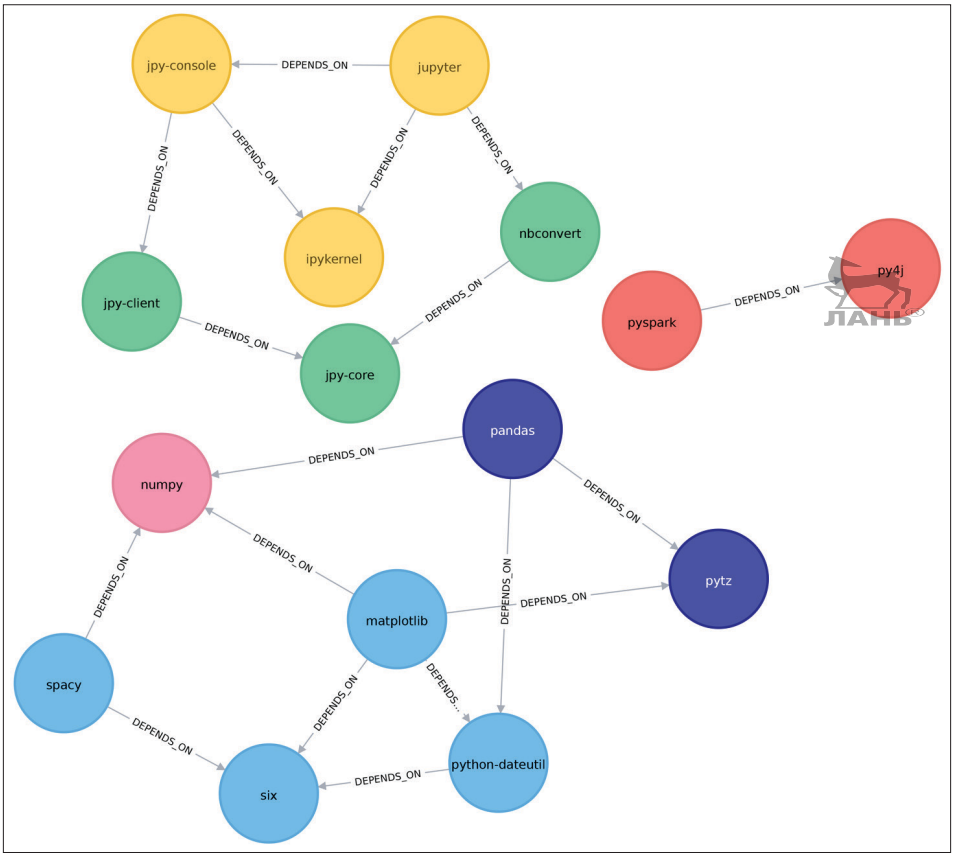


Рис. 6.14. Промежуточные кластеры, найденные Лувенским алгоритмом

Хотя этот граф показывает только два уровня иерархии, если мы запустим этот алгоритм на большом графе, то увидим более сложную иерархию. Промежуточные кластеры, раскрываемые Лувенским алгоритмом, могут быть очень полезны для обнаружения мелкозернистых сообществ, которые не могут быть обнаружены другими алгоритмами обнаружения сообществ.

Проверка достоверности сообществ

Алгоритмы выявления сообществ обычно имеют одну и ту же цель: идентифицировать группы. Однако, поскольку разные алгоритмы начинаются с разных допущений, они могут выявить разные сообщества. Это делает выбор правильного алгоритма для конкретной задачи более сложным и требующим пояснений.

Большинство алгоритмов выявления сообществ работает достаточно хорошо, когда плотность отношений в группах высока по сравнению с их окружением, но сети реального мира часто менее разделены. Мы можем проверить достоверность найденных сообществ, сравнив наши результаты с эталоном, основанным на данных известных сообществ.

Двумя наиболее известными генераторами тестов являются алгоритмы Гирвана–Ньюмана (Girvan–Newman, GN) и Ланчикинетти–Фортунато–Радиччи (Lancichinetti–Fortunato–Radicchi, LFR). Эталонные сети, которые генерируют эти алгоритмы, весьма различны: GN генерирует случайную сеть, которая является относительно однородной, тогда как LFR создает более гетерогенный граф, где степени вершин и размер сообщества распределены в соответствии со степенным законом.

Поскольку достоверность нашего тестирования зависит от используемого эталонного теста, важно сопоставить эталонный тест с нашим набором данных. Максимально старательно ищите аналогичные плотности, распределения связей, определения сообществ и связанные области.

Заключение

Алгоритмы выявления сообществ полезны для понимания того, как сгруппированы вершины в графе.

В этой главе мы начали с изучения алгоритмов подсчета треугольников и коэффициента кластеризации. Затем перешли к двум детерминированным алгоритмам выявления сообществ: сильно связанных компонентов и связанных компонентов. Эти алгоритмы имеют строгие определения критериев, по которым выявляется сообщество, и очень полезны, для того чтобы оценить структуру графа на ранних этапах последовательного анализа.

Мы закончили главу двумя недетерминированными алгоритмами – Лувенским и алгоритмом распространения метки, – которые лучше способны обнаруживать мелкозернистые кластеры. Лувенский алгоритм также показывает нам иерархию сообществ в разных масштабах.

В следующей главе мы возьмем намного более обширный набор данных и расскажем, как объединить алгоритмы, чтобы получить еще больше информации о связанных данных.

Глава 7

Графовые алгоритмы на практике

Подход, который вы применяете к анализу графов, усложняется, по мере того как вы глубже изучаете поведение различных алгоритмов на определенных наборах данных. В этой главе мы рассмотрим несколько примеров, чтобы дать вам углубленное представление о том, как проводить анализ данных крупномасштабных графов, таких как набор данных от Yelp и Министерства транспорта США. Мы выполним анализ данных Yelp на платформе Neo4j, включая общий обзор данных, объединение алгоритмов для составления рекомендаций по поездкам и анализ пользовательских и бизнес-данных для консультирования. На платформе Spark мы исследуем данные американских авиакомпаний, чтобы разобраться с транспортными схемами и задержками, а также узнать, как аэропорты связаны разными авиакомпаниями.

Поскольку алгоритмы поиска путей просты и очевидны, в наших примерах будут использоваться следующие алгоритмы вычисления центральности и выделения сообществ:

- PageRank, чтобы найти влиятельных обозревателей Yelp, а затем сопоставить их рейтинги для конкретных отелей;
- центральность по посредничеству, чтобы выявить обозревателей, связанных с несколькими группами, и затем извлечь их предпочтения;
- распространение метки с прогнозом для создания суперкатегорий одинаковых предприятий Yelp;
- степенная центральность для быстрого определения авиационных хабов в наборах данных транспортной системы США;
- сильно связанные компоненты, чтобы выделить кластеры авиационных маршрутов в США.

Анализ данных Yelp на платформе Neo4j

Yelp помогает людям находить местные предприятия и услуги, основываясь на отзывах, предпочтениях и рекомендациях. По состоянию на конец 2018 года на портале было написано более 180 млн отзывов. С 2013 года Yelp регулярно проводит конкурс Yelp Dataset, призывая всех желающих исследовать и изучать открытый набор данных Yelp.

Открытый набор данных 12-го раунда, проведенного в 2018 году, содержал:

- более 7 млн отзывов плюс рекомендации;
- более 1,5 млн пользователей и 280 000 изображений;
- более 188 000 предприятий с 1,4 млн атрибутов;
- 10 мегаполисов.

С момента запуска конкурса этот набор данных стал популярным, и с использованием этого материала написаны сотни научных работ. Набор данных Yelp представляет реальные данные, которые очень хорошо структурированы и тесно взаимосвязаны. Это отличная демонстрация графовых алгоритмов, которую вы также можете скачать и изучить.

Социальная сеть Yelp

Помимо написания и чтения отзывов о компаниях, пользователи Yelp образуют социальную сеть. Пользователи могут отправлять запросы на добавление в друзья другим пользователям, с которыми они встречаются при просмотре Yelp.com, а также могут подключать свои адресные книги или графы Facebook.

Набор данных Yelp также охватывает социальную сеть. На рис. 7.1 показан снимок экрана раздела «Друзья» в профиле Yelp Марка.

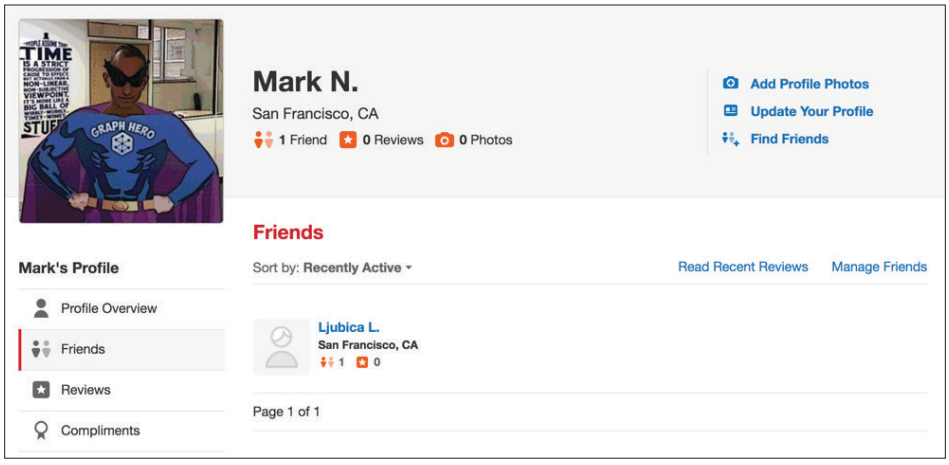


Рис. 7.1. Профиль Марка в социальной сети Yelp

Если не считать того, что Марку нужно завести побольше друзей, мы готовы начать. Чтобы проиллюстрировать анализ данныхYelp на платформе Neo4j, представим себе сценарий, будто мы работаем в области туристической информатики. Сначала мы изучим данные Yelp, а затем подумаем, как помочь людям планировать поездки при помощи специального приложения. Мы расскажем, как получить хорошие рекомендации по подбору мест для отдыха и развлечений в крупных городах, таких как Лас-Вегас.

Другая часть нашего бизнес-сценария будет включать в себя консультирование предприятий, занимающихся путешествиями. В нашем примере мы поможем отелям определить влиятельных посетителей, а затем выбрать предприятия, на которые следует ориентироваться в программах перекрестного продвижения.

Импорт данных

Существует много различных способов импорта данных в Neo4j, включая инструмент импорта[1], команду[2] `LOAD CSV`, которую мы видели в предыдущих главах, и драйверы[3] Neo4j.

Для работы с данными Yelp нам необходимо выполнить однократный импорт большого объема данных, поэтому лучший выбор – инструмент импорта. Прочтите также раздел «Пакетный импорт данных Yelp в Neo4j» в приложении для получения более подробной информации.

Графовая модель

Данные Yelp можно представить в виде графовой модели, как показано на рис. 7.2.

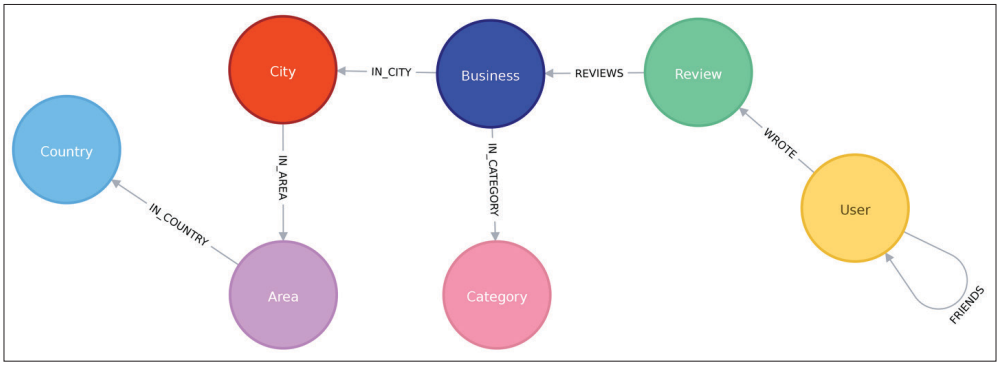


Рис. 7.2. Графовая модель Yelp

Наш граф содержит вершины, помеченные меткой `User` (пользователь) которые имеют связи `FRIENDS` с другими пользователями. Пользователи

[1] https://neo4j.com/docs/operations-manual/current/tools/import/
[2] https://neo4j.com/developer/guide-import-csv/
[3] https://neo4j.com/docs/driver-manual/1.7/

также пишут отзывы (`Reviews`) и советы о бизнесе (`Business`). Все метаданные хранятся в виде свойств вершин, за исключением бизнес-категорий, которые представлены отдельными вершинами категории (`Category`). Для данных о местоположении мы извлекли атрибуты `City` (город), `Area` (регион) и `Country` (страна) в подграф. В других случаях использования может иметь смысл извлечь в вершины другие атрибуты, например даты.

Набор данных Yelp также содержит советы пользователей и фотографии, но мы не будем использовать их в нашем примере.

Краткий обзор данных Yelp

Загрузив данные в Neo4j, мы сразу выполним несколько ознакомительных запросов. Чтобы получить представление о данных Yelp, мы спросим, сколько вершин входит в каждую категорию или какие типы связей существуют. Ранее в наших примерах работы с Neo4j мы показывали запросы Cypher, но можно использовать и другие языки программирования. Поскольку Python является первоочередным языком для исследователей данных, в этом разделе мы воспользуемся драйвером Python Neo4j, когда захотим связать результаты с другими библиотеками из экосистемы Python. Если же мы просто захотим посмотреть результат запроса, то применим Cypher.

Далее мы покажем, как объединить Neo4j с популярной библиотекой pandas, которая эффективно обрабатывает данные вне баз данных. Вы увидите, как использовать библиотеку tabulate для привлекательного представления результатов, полученных из pandas, и как создавать визуальные представления данных с помощью matplotlib.

Мы также воспользуемся библиотекой процедур APOC Neo4j, которая поможет нам писать еще более мощные запросы Cypher. Более подробная информация об APOC приведена в разделе приложения «APOC и другие инструменты Neo4j».

Давайте сначала установим библиотеки Python:

```
pip install neo4j-driver tabulate pandas matplotlib
```

Затем импортируем эти библиотеки:

```
from neo4j.v1 import GraphDatabase
import pandas as pd
from tabulate import tabulate
```

Импорт matplotlib может вызвать затруднения на macOS, но следующие строки должны сделать свое дело:

```
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
```

Если вы работаете в другой операционной системе, средняя строка может не потребоваться. Теперь давайте создадим экземпляр драйвера Neo4j, указывающего на локальную базу данных Neo4j:

```
driver = GraphDatabase.driver("bolt://localhost", auth=("neo4j", "neo"))
```

Вы должны изменить строку инициализации драйвера, чтобы использовать свой собственный хост и учетные данные.

Для начала давайте взглянем на некоторые общие числа для вершин и ребер графа. Следующий код вычисляет *кардинальности* меток вершин (т. е. подсчитывает количество вершин для каждой метки) в базе данных:

```
result = {"label": [], "count": []}
with driver.session() as session:
    labels = [row["label"] for row in session.run("CALL db.labels()")]
    for label in labels:
        query = f"MATCH (:`{label}`) RETURN count(*) as count"
        count = session.run(query).single()["count"]
        result["label"].append(label)
        result["count"].append(count)

df = pd.DataFrame(data=result)
print(tabulate(df.sort_values("count"), headers='keys',
                                tablefmt='psql', showindex=False))
```

Запустив этот код, мы увидим, сколько у нас вершин для каждой метки:

label	count
Country	17
Area	54
City	1093
Category	1293
Business	174567
User	1326101
Review	5261669

Мы также можем создать визуальное представление кардинальностей при помощи следующего кода:

```
plt.style.use('fivethirtyeight')

ax = df.plot(kind='bar', x='label', y='count', legend=None)

ax.xaxis.set_label_text("")
plt.yscale("log")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

Диаграмма, сгенерированная этим кодом, представлена на рис. 7.3. Обратите внимание, что эта диаграмма использует логарифмический масштаб.

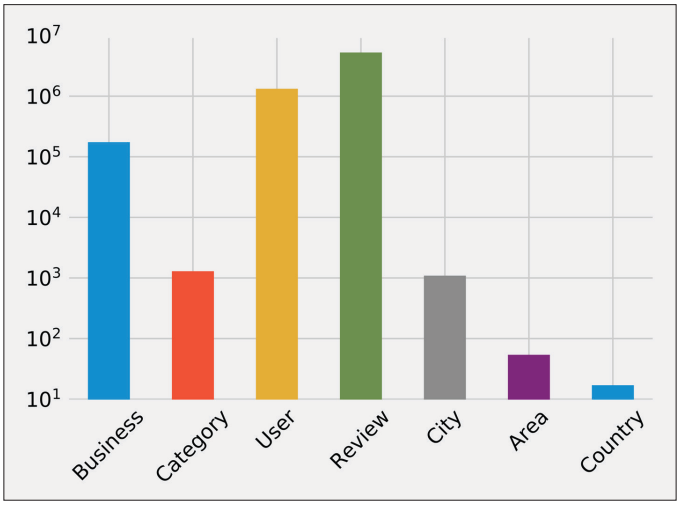


Рис. 7.3. Количество узлов для каждой категории

Точно так же мы можем вычислить кардинальность ребер:

```
result = {"relType": [], "count": []}
with driver.session() as session:
    rel_types = [row["relationshipType"] for row in session.run
                                ("CALL db.relationshipTypes()")]
    for rel_type in rel_types:
        query = f"MATCH ()-[:`{rel_type}`]->() RETURN count(*) as count"
        count = session.run(query).single()["count"]
        result["relType"].append(rel_type)
        result["count"].append(count)

df = pd.DataFrame(data=result)
print(tabulate(df.sort_values("count"), headers='keys',
                                tablefmt='psql', showindex=False))
```

Запустив код, мы увидим количество каждого типа ребер:

relType	count
IN_COUNTRY	54
IN_AREA	1154
IN_CITY	174566
IN_CATEGORY	667527
WROTE	5261669
REVIEWS	5261669
FRIENDS	10645356

Соответствующая диаграмма кардинальности изображена на рис. 7.4. Как и в случае с диаграммой количества вершин, эта диаграмма использует логарифмическую шкалу.

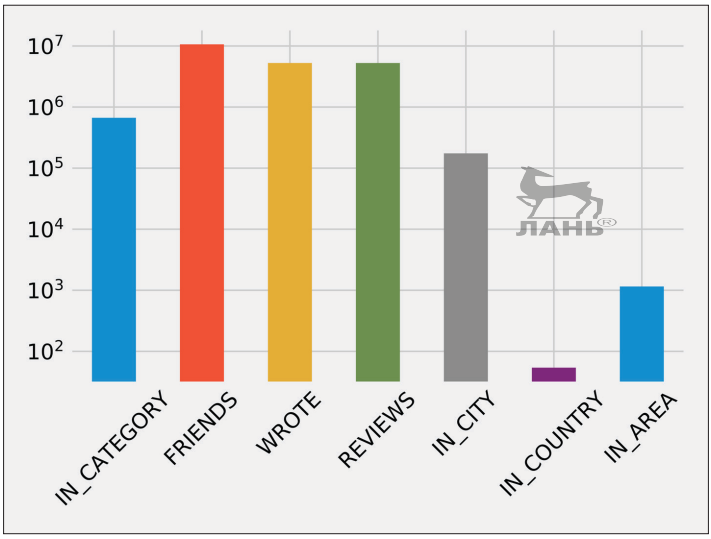


Рис. 7.4. Количество ребер с распределением по типам

Эти запросы не должны показывать ничего удивительного, но они полезны для понимания того, что находится в данных, а также служат для быстрой проверки правильности импорта данных.

Мы предполагаем, что Yelp содержит много отзывов об отелях, но имеет смысл убедиться, прежде чем сосредоточиться на этом секторе. Мы можем узнать, сколько гостиничных предприятий находится в этих данных и сколько отзывов они имеют, выполнив следующий запрос:

```
MATCH (category:Category {name: "Hotels"})
RETURN size((category)<-[:IN_CATEGORY]-()) AS businesses,
```

```
    size((:Review)-[:REVIEWS]->(:Business)-[:IN_CATEGORY]->
                                (category)) AS reviews
```

Результат выглядит следующим образом:

business	reviews
2683	183759

У нас много предприятий, с которыми мы можем поработать, и множество отзывов о них! В следующем разделе мы подробнее изучим данные уже с учетом нашего бизнес-сценария.

Приложение для планирования поездки

Чтобы добавить в наше приложение полезные рекомендации, начнем с поиска отелей с самым высоким рейтингом в качестве эвристического источника популярных вариантов бронирования. Мы можем посмотреть, насколько высокую оценку отелю поставили посетители, и отсюда вывести реальный опыт потребителя. Чтобы выбрать 10 самых популярных отелей и построить распределение их рейтингов, мы используем следующий код:

```
# Ищем 10 отелей с наибольшим количеством отзывов
query = """
MATCH (review:Review)-[:REVIEWS]->(business:Business),
      (business)-[:IN_CATEGORY]->(category:Category {name: $category}),
      (business)-[:IN_CITY]->(:City {name: $city})
RETURN business.name AS business, collect(review.stars) AS allReviews
ORDER BY size(allReviews) DESC
LIMIT 10
"""

fig = plt.figure()
fig.set_size_inches(10.5, 14.5)
fig.subplots_adjust(hspace=0.4, wspace=0.4)

with driver.session() as session:
    params = { "city": "Las Vegas", "category": "Hotels"}
    result = session.run(query, params)
    for index, row in enumerate(result):
        business = row["business"]
        stars = pd.Series(row["allReviews"])
        total = stars.count()
        average_stars = stars.mean().round(2)
        # Calculate the star distribution
        stars_histogram = stars.value_counts().sort_index()
```

```
        stars_histogram /= float(stars_histogram.sum())
        # Plot a bar chart showing the distribution of star ratings
        ax = fig.add_subplot(5, 2, index+1)
        stars_histogram.plot(kind="bar", legend=None, color="darkblue",
                             title=f"{business}\nAve:
                                     {average_stars}, Total: {total}")
plt.tight_layout()
plt.show()
```

В этом коде мы в явном виде указали город и категорию, чтобы сосредоточиться на отелях Лас-Вегаса. Запустив этот код, мы получим диаграмму, показанную на рис. 7.5. Обратите внимание, что ось *X* представляет рейтинг отеля в звездах, а ось *Y* – общий процент каждого рейтинга.

Эти отели имеют больше отзывов, чем кто-либо способен прочитать. Будет намного лучше, если мы покажем пользователям самые полезные отзывы и сделаем их более заметными в нашем приложении. Для этого анализа мы перейдем от базового исследования графов к использованию графовых алгоритмов.

Поиск влиятельных отельных обозревателей

Один из способов, которым мы можем решить, какие обзоры публиковать, – это сортировать отзывы, основываясь на *влиянии обозревателя* на Yelp. Мы запустим алгоритм PageRank над прогнозируемым графом всех пользователей, которые просмотрели как минимум три отеля. Как вы знаете из предыдущих глав, проекция графа может помочь отфильтровать несущественную информацию, а также добавить данные об отношениях (иногда предполагаемые). В качестве источника данных о связях между пользователями мы будем использовать граф друзей Yelp. Алгоритм PageRank покажет нам обозревателей, у которых больше влияния на большее количество пользователей, даже если они не являются прямыми друзьями.

Если два человека – друзья в сети Yelp, между ними установлено две связи `FRIENDS`. Например, если *A* и *B* являются друзьями, одна связь `FRIENDS` будет направлена от *A* к *B*, а другая – от *B* к *A*.

Нам нужно написать запрос, который создает проекцию подграфа пользователей с более чем тремя отзывами, а затем выполняет алгоритм PageRank над этой проекцией.

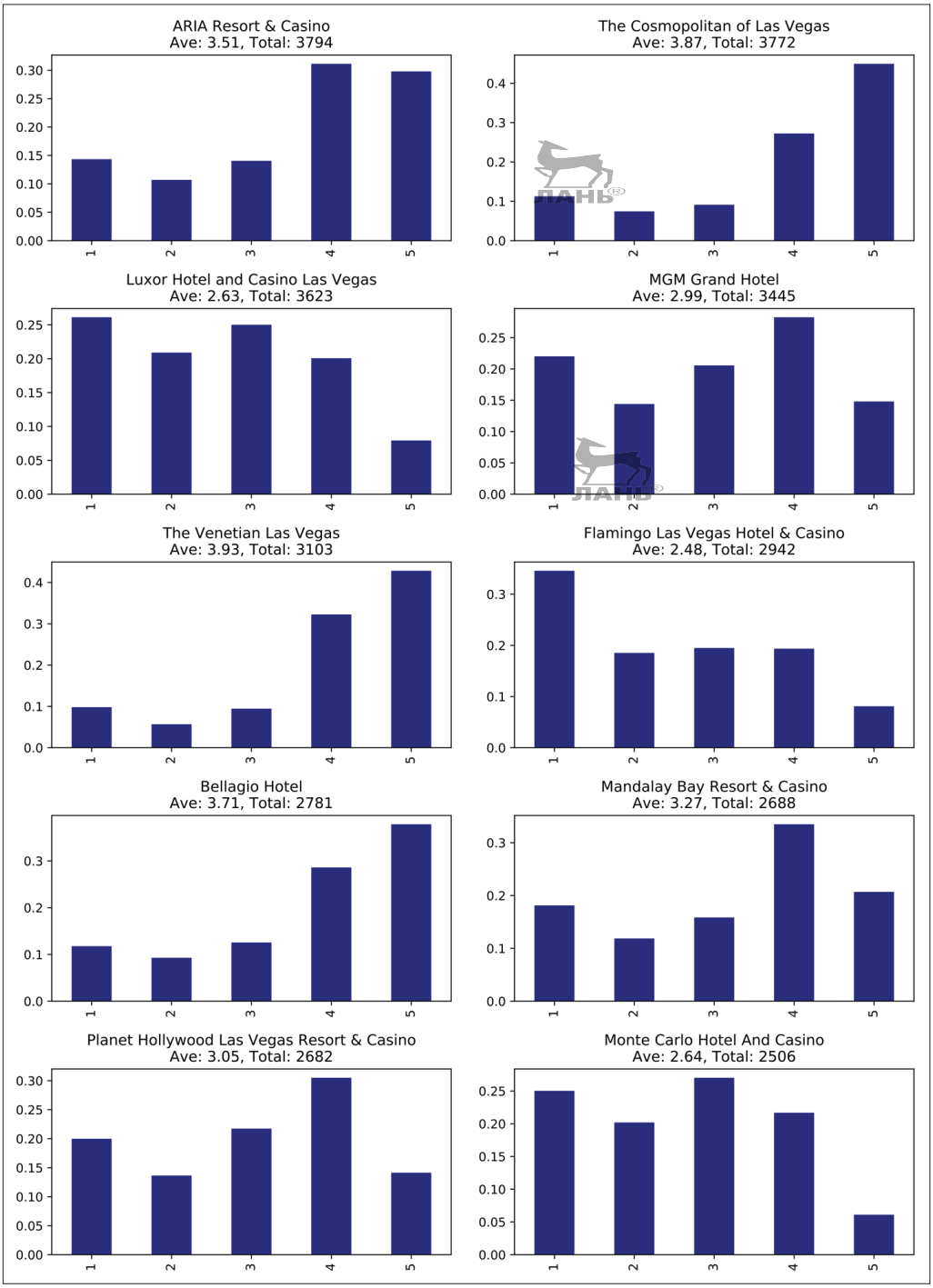


Рис. 7.5. 10 самых популярных отелей с количеством звезд по оси *X* и их общим рейтингом по оси *Y*

Проще понять, как работает проекция подграфа, на небольшом примере. На рис. 7.6 показан граф трех общих друзей – Марка, Арьи и Правины. Марк и Правина написали отзывы на три отеля и станут частью проецируемого подграфа. Арья, с другой стороны, написала отзыв только на один отель и поэтому будет исключена из проекции.

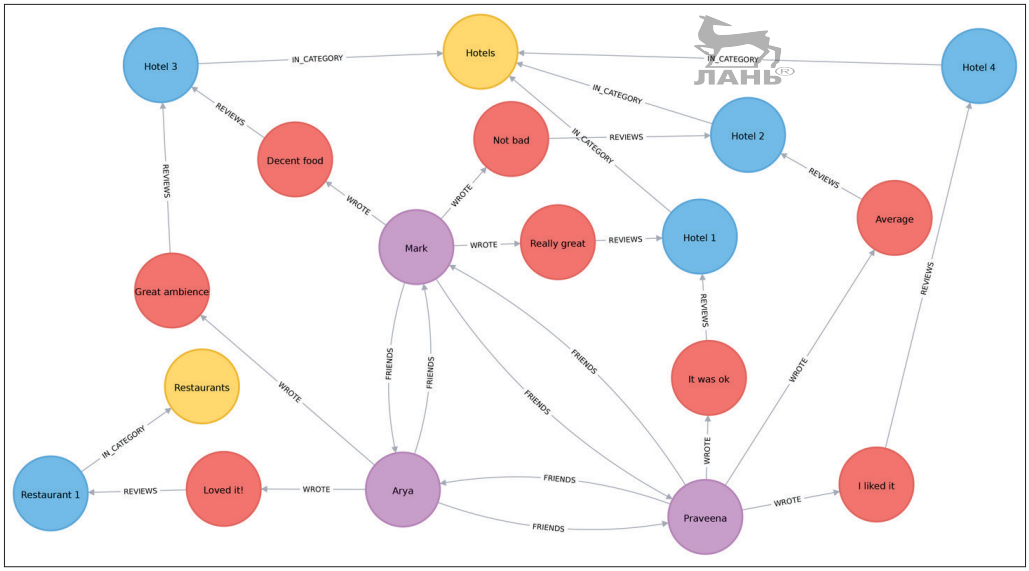


Рис. 7.6. Пример графа Yelp

Следовательно, проекция графа будет включать только Марка и Правину, как показано на рис. 7.7.

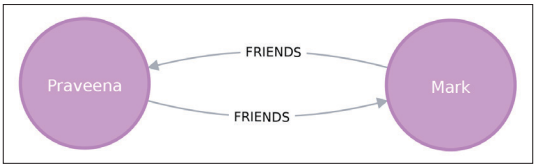


Рис. 7.7. Пример проекции графа

Теперь, когда мы разобрались, как работает проекция графа, давайте двигаться вперед. Следующий запрос выполняет алгоритм PageRank над проекцией графа и сохраняет результат в свойстве `hotelPageRank` каждой вершины:

```
CALL algo.pageRank(
  'MATCH (u:User)-[:WROTE]->()-[:REVIEWS]->()-[:IN_CATEGORY]->
                            (:Category {name: $category})
  WITH u, count(*) AS reviews
  WHERE reviews >= $cutOff
  RETURN id(u) AS id',
```

```
  'MATCH (u1:User)-[:WROTE]->()-[:REVIEWS]->()-[:IN_CATEGORY]->
                                (:Category {name: $category})
  MATCH (u1)-[:FRIENDS]->(u2)
  RETURN id(u1) AS source, id(u2) AS target',
  {graph: "cypher", write: true, writeProperty: "hotelPageRank",
  params: {category: «Hotels», cutOff: 3}}
)
```

Возможно, вы заметили, что мы не установили коэффициент затухания или максимальное число итераций, о которых говорили в главе 5. Если явно не задано иное, Neo4j по умолчанию использует коэффициент затухания 0,85 и максимальное количество итераций `maxIterations=20`.

Теперь давайте рассмотрим распределение значений PageRank, чтобы решить, как фильтровать наши данные:

```
MATCH (u:User)
WHERE exists(u.hotelPageRank)
RETURN count(u.hotelPageRank) AS count,
       avg(u.hotelPageRank) AS ave,
       percentileDisc(u.hotelPageRank, 0.5) AS `50%`,
       percentileDisc(u.hotelPageRank, 0.75) AS `75%`,
       percentileDisc(u.hotelPageRank, 0.90) AS `90%`,
       percentileDisc(u.hotelPageRank, 0.95) AS `95%`,
       percentileDisc(u.hotelPageRank, 0.99) AS `99%`,
       percentileDisc(u.hotelPageRank, 0.999) AS `99.9%`,
       percentileDisc(u.hotelPageRank, 0.9999) AS `99.99%`,
       percentileDisc(u.hotelPageRank, 0.99999) AS `99.999%`,
       percentileDisc(u.hotelPageRank, 1) AS `100%`
```

Выполнив запрос, мы получим следующий выход:

count	ave	50 %	75 %	90 %	95 %	99 %	99.9 %	99.99 %	99.999 %	100 %
1326101	0.1614898	0.15	0.15	0.157497	0.181875	0.330081	1.649511	6.825738	15.27376	22.98046

Как интерпретировать эту таблицу процентилей? Пара значений `90%` и `0.157497` означает, что 90 % пользователей имели более низкий балл PageRank. 99,99 % отражает рейтинг влияния для лучших 0,0001 % рецензентов, а 100 % – это просто самый высокий балл PageRank.

Интересно, что 90 % наших пользователей имеют балл менее `0.16`, что близко к общему среднему значению и лишь незначительно превышает начальное `0.15`, с которыми они инициализируются алгоритмом PageRank. Похоже, что эти данные отражают степенное распределение с несколькими очень влиятельными обозревателями.

Поскольку мы хотим найти только самых влиятельных пользователей, напишем запрос, который найдет пользователей с оценкой PageRank сре-

ди лучших 0,001 % пользователей. Следующий запрос находит рецензентов с рейтингом PageRank выше `1.64951` (обратите внимание, что это группа `99.9%`):

```
// Отбираем только пользователей, у которых hotelPageRank входит в 0.001%
MATCH (u:User)
WHERE u.hotelPageRank > 1.64951

// Ищем десятку лучших среди этих пользователей.
WITH u ORDER BY u.hotelPageRank DESC
LIMIT 10

RETURN u.name AS name,
       u.hotelPageRank AS pageRank,
       size((u)-[:WROTE]->()-[:REVIEWS]->()-[:IN_CATEGORY]->
            (:Category {name: "Hotels"})) AS hotelReviews,
       size((u)-[:WROTE]->()) AS totalReviews,
       size((u)-[:FRIENDS]-()) AS friends
```

Выполнив запрос, мы получим следующий список пользователей:

name	pageRank	hotelReviews	totalReviews	friends
Phil	17.361242	15	134	8154
Philip	16.871013	21	620	9634
Carol	12.416060999999997	6	119	6218
Misti	12.239516000000004	19	730	6230
Joseph	12.003887499999998	5	32	6596
Michael	11.460049	13	51	6572
J	11.431505999999997	103	1322	6498
Abby	11.376136999999998	9	82	7922
Erica	10.993773	6	15	7071
Randy	10.748785999999999	21	125	7846

Эти результаты показывают, что Фил (Phil) – самый заслуживающий доверия обозреватель, хотя он и не проверял многие отели. Он, вероятно, связан с некоторыми очень влиятельными людьми, но, если нам нужен поток *новых* обзоров, его профиль не будет лучшим выбором. У Филиппа (Philip) немного ниже балл, но зато больше друзей, и он написал в пять раз больше отзывов, чем Фил. Хотя пользователь Джей (J) написал больше всего отзывов и имеет достаточное количество друзей, его балл PageRank не самый высокий, но он все еще входит в топ-10. Для нашего приложения мы выбираем обзоры отелей от Фила, Филиппа и Джея, чтобы получить разумное сочетание влияния и количества отзывов.

Теперь, когда мы улучшили рекомендации в приложении за счет релевантных обзоров, давайте перейдем к другой стороне нашего бизнеса – консалтингу.

Туристический бизнес-консалтинг

В рамках нашего консалтингового обслуживания отели подписываются на уведомление, как только влиятельный посетитель пишет в соцсети о своем пребывании, чтобы они могли предпринять любые необходимые действия. Сначала рассмотрим рейтинги отеля «Белладжио», отсортированные по наиболее влиятельным обозревателям:

```
query = """\
MATCH (b:Business {name: $hotel})
MATCH (b)<-[:REVIEWS]-(review)<-[:WROTE]-(user)
WHERE exists(user.hotelPageRank)
RETURN user.name AS name,
       user.hotelPageRank AS pageRank,
       review.stars AS stars
"""
```



```
with driver.session() as session:
    params = { "hotel": "Bellagio Hotel" }
    df = pd.DataFrame([dict(record) for record in session.run(query, params)])
    df = df.round(2)
    df = df[["name", "pageRank", "stars"]]

top_reviews = df.sort_values(by=["pageRank"], ascending=False).head(10)
print(tabulate(top_reviews, headers='keys', tablefmt='psql', showindex=False))
```

Выполнив этот код, мы получим следующий список:

name	pageRank	stars
Misti	12.239516000000004	5
Michael	11.460049	4
J	11.431505999999997	5
Erica	10.993773	4
Christine	10.740770499999998	4
Jeremy	9.576763499999998	5
Connie	9.118103499999998	5
Joyce	7.621449000000001	4
Henry	7.299146	5
Flora	6.7570075	4

Обратите внимание, что эти результаты отличаются от нашей предыдущей таблицы лучших обозревателей отелей. Это потому что здесь мы смотрим только на обозревателей, которые оценили «Белладжио».

У команды по обслуживанию клиентов в отеле «Белладжио» дела обстоят неплохо – все 10 влиятельных обозревателей ставят своему отелю высокие оценки. Возможно, администрация захочет побудить этих гостей посетить отель снова и поделиться своим опытом.

Но есть ли влиятельные обозреватели, у которых не было столь хорошего опыта? Мы можем запустить следующий код, чтобы выбрать гостей с самым высоким PageRank, которые оценили свой опыт менее чем на четыре звезды:

```
query = """\
MATCH (b:Business {name: $hotel})
MATCH (b)<-[:REVIEWS]-(review)<-[:WROTE]-(user)
WHERE exists(user.hotelPageRank) AND review.stars < $goodRating
RETURN user.name AS name,
       user.hotelPageRank AS pageRank,
       review.stars AS stars
"""

with driver.session() as session:
    params = { "hotel": "Bellagio Hotel", "goodRating": 4 }
    df = pd.DataFrame([dict(record) for record in session.run(query, params)])
    df = df.round(2)
    df = df[["name", "pageRank", "stars"]]

top_reviews = df.sort_values(by=["pageRank"], ascending=False).head(10)
print(tabulate(top_reviews, headers='keys', tablefmt='psql', showindex=False))
```

Выполнение кода дает следующий список:

name	pageRank	stars
Chris	5.84	3
Lorrie	4.95	2
Dani	3.47	1
Victor	3.35	3
Francine	2.93	3
Rex	2.79	2
Jon	2.55	3
Rachel	2.47	3
Leslie	2.46	2
Benay	2.46	3

Наши авторитетные обозреватели с самым высоким рейтингом, поставившие низкие оценки отелю «Белладжио», Крис (Chris) и Лори (Lorrie), входят в число 1000 самых влиятельных пользователей (согласно результатам нашего предыдущего запроса), поэтому, возможно, для администрации отеля имеет смысл обратиться к ним лично. Кроме того, поскольку многие обозреватели пишут отзывы во время своего пребывания, оповещения администрации о действиях обозревателей в режиме реального времени могут способствовать еще более позитивным результатам.

Совместное продвижение отеля «Белладжио»

После того как мы помогли отелю «Белладжио» найти влиятельных обозревателей, администратор отеля попросил нас помочь найти другие компании для *совместного взаимного продвижения* (cross-promotion) с помощью клиентов с хорошими связями. В нашем сценарии в качестве новой возможности мы рекомендуем увеличить клиентскую базу за счет привлечения новых гостей из различных типов сообществ. Мы можем использовать алгоритм центральности по посредничеству, который обсуждали ранее, чтобы определить, какие обозреватели отеля «Белладжио» не только обладают обширными связями по всей сети Yelp, но также могут выступать в качестве моста между различными группами.

Мы заинтересованы только в поиске влиятельных лиц в Лас-Вегасе, поэтому сначала настроим условие отбора:

```
MATCH (u:User)
WHERE exists((u)-[:WROTE]->()-[:REVIEWS]->()-[:IN_CITY]->
                                (:City {name: "Las Vegas"}))
SET u:LasVegas
```

Чтобы выполнить алгоритм центральности по посредничеству, для пользователей из Лас-Вегаса потребовалось бы много времени, поэтому мы воспользуемся вариантом алгоритма Брандеса. Этот алгоритм вычисляет оценку посредничества по выборкам вершин и изучает только кратчайшие пути до определенной глубины.

После некоторых экспериментов мы улучшили результаты, установив несколько параметров, отличных от значений по умолчанию. Мы будем использовать кратчайшие пути до четырех переходов (`maxDepth=4`) и отбирать 20 % вершин (`probability=0.2`). Обратите внимание, что увеличение количества переходов и вершин, как правило, повышает точность, но требует больше времени для вычисления результатов. Подбор оптимальных параметров для любой конкретной проблемы обычно требует тестирования и определения точки убывающей отдачи.

Следующий запрос выполнит алгоритм и сохранит результат в свойстве `between`:

```
CALL algo.betweenness.sampled('LasVegas', 'FRIENDS',
  {write: true, writeProperty: "between", maxDepth: 4, probability: 0.2}
)
```

Прежде чем использовать эти оценки в наших запросах, давайте напишем быстрый контрольно-исследовательский запрос, чтобы увидеть, как распределяются оценки:


```
MATCH (u:User)
WHERE exists(u.between)
RETURN count(u.between) AS count,
       avg(u.between) AS ave,
       toInteger(percentileDisc(u.between, 0.5)) AS `50%`,
       toInteger(percentileDisc(u.between, 0.75)) AS `75%`,
       toInteger(percentileDisc(u.between, 0.90)) AS `90%`,
       toInteger(percentileDisc(u.between, 0.95)) AS `95%`,
       toInteger(percentileDisc(u.between, 0.99)) AS `99%`,
       toInteger(percentileDisc(u.between, 0.999)) AS `99.9%`,
       toInteger(percentileDisc(u.between, 0.9999)) AS `99.99%`,
       toInteger(percentileDisc(u.between, 0.99999)) AS `99.999%`,
       toInteger(percentileDisc(u.between, 1)) AS p100
```


Выполнив запрос, мы получим следующие данные:

count	ave	50%	75%	90%	95%	99%	99.9%	99.99%	99.999%	100%
506028	320538.6014	0	10005	318944	1001655	4436409	34854988	214080923	621434012	1998032952

У половины наших пользователей оценка 0, что означает, что они не очень хорошо связаны. Верхний 1-й процентиль (столбец 99%) соответствует как минимум 4 млн кратчайших путей между нашим набором из 500 000 пользователей. В целом мы уже знаем, что большинство наших пользователей плохо связаны, но некоторые из них имеют большой контроль над информацией – это классическое поведение сетей малого мира.

Мы можем узнать, кто наши суперконнекторы, выполнив следующий запрос:

```
MATCH(u:User)-[:WROTE]->()-[:REVIEWS]->(:Business {name:"Bellagio Hotel"})
WHERE exists(u.between)
RETURN u.name AS user,
       toInteger(u.between) AS betweenness,
       u.hotelPageRank AS pageRank,
       size((u)-[:WROTE]->()-[:REVIEWS]->()-[:IN_CATEGORY]->
                           (:Category {name: "Hotels"}))
       AS hotelReviews
ORDER BY u.between DESC
LIMIT 10
```

Результат запроса выглядит следующим образом:

user	betweenness	pageRank	hotelReviews
Misti	841707563	12.239516000000004	19
Christine	236269693	10.740770499999998	16
Erica	235806844	10.993773	6
Mike	215534452	NULL	2
J	192155233	11.431505999999997	103
Michael	161335816	5.105143	31
Jeremy	160312436	9.576763499999998	6
Michael	139960910	11.460049	13
Chris	136697785	5.838922499999999	5
Connie	133372418	9.118103499999998	7

Здесь мы видим некоторых из тех же людей, которых встречали ранее в нашем запросе PageRank, но Майк (Mike) является интересным исключением. Его не принимали в расчет, потому что он не проверял достаточное количество отелей (три отзыва были минимальным порогом), но, похоже, он достаточно хорошо связан с миром пользователей Yelp в Лас-Вегасе.

Чтобы охватить более широкий круг клиентов, рассмотрим другие предпочтения, на которые указывают эти «соединители», чтобы увидеть, что нам следует продвигать. Многие из этих пользователей также оставили обзоры на рестораны, поэтому мы напишем следующий запрос, чтобы выяснить, какие из ресторанов им нравятся больше всего:

```
// Отбираем топ 50 пользователей, оставивших отзыв на «Белладжио»
MATCH (u:User)-[:WROTE]->()-[:REVIEWS]->(:Business {name:"Bellagio Hotel"})
WHERE u.between > 4436409
WITH u ORDER BY u.between DESC LIMIT 50

// Отбираем рестораны, которые эти пользователи посетили в Лас-Вегасе
MATCH (u)-[:WROTE]->(review)-[:REVIEWS]-(business)
WHERE (business)-[:IN_CATEGORY]->(:Category {name: "Restaurants"})
AND (business)-[:IN_CITY]->(:City {name: "Las Vegas"})

// Оставляем только рестораны, получившие более 3 отзывов от этих пользователей
WITH business, avg(review.stars) AS averageReview, count(*) AS numberOfReviews
WHERE numberOfReviews >= 3

RETURN business.name AS business, averageReview, numberOfReviews
ORDER BY averageReview DESC, numberOfReviews DESC
LIMIT 10
```

Этот запрос находит 50 самых влиятельных «соединителей» и выявляет 10 лучших ресторанов Лас-Вегаса, которые оценили как минимум три выбранных ранее пользователя. Запустив этот запрос, мы получим такой результат:

business	averageReview	numberOfReviews
Jean Georges Steakhouse	5.0	6
Sushi House Goyemon	5.0	6
Art of Flavors	5.0	4
é by José Andrés	5.0	4
Parma By Chef Marc	5.0	4
Yonaka Modern Japanese	5.0	4
Kabuto	5.0	4
Harvest by Roy Ellamar	5.0	3
Portofino by Chef Michael LaPlaca	5.0	3
Montesano's Eateria	5.0	3

Теперь можно порекомендовать отелю «Белладжио» провести совместную акцию с этими ресторанами, чтобы привлечь новых гостей из групп, с которыми они обычно не взаимодействуют. Пользователи, высоко оценивающие «Белладжио», становятся нашими посредниками для оценки того, какие рестораны могут привлечь внимание целевых посетителей из новых групп.

Теперь, когда мы помогли «Белладжио» охватить новые группы, можно обсудить, как использовать алгоритм выделения сообществ для дальнейшего улучшения нашего приложения.

Поиск похожих категорий

Хотя наши конечные пользователи используют приложение для поиска отелей, мы хотим заодно порекомендовать другие бизнес-идеи, которые могут быть им интересны. Набор данных Yelp содержит более 1000 категорий, и некоторые из них явно похожи друг на друга. Мы воспользуемся этим сходством, чтобы рекомендовать новые компании, которые, вероятно, будут интересны нашим пользователям.

Наша графовая модель не имеет каких-либо связей между категориями, но можно использовать идеи, описанные в главе 2, раздел «Однокомпонентные, двудольные и *k*-дольные графы», чтобы построить граф сходства категорий на основе того, как компании позиционируют себя.

Например, представьте, что только одна компания позиционирует себя как в категории отелей, так и в категории организаторов исторических туров, как показано на рис. 7.8.

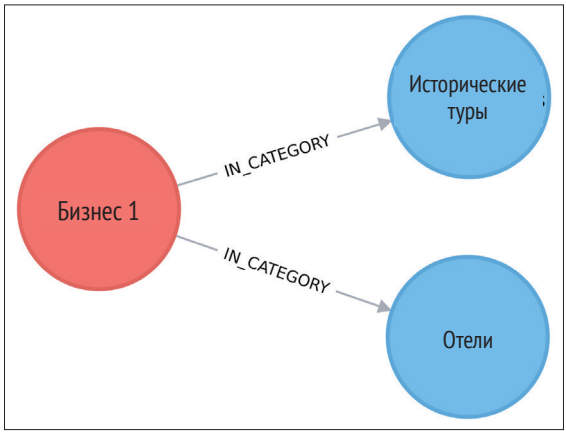


Рис. 7.8. Бизнес с двумя категориями

Это дает нам проекцию графа, которая имеет связь между отелями и историческими турами с весом 1, как показано на рис. 7.9.

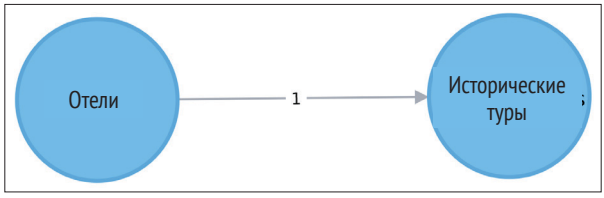


Рис. 7.9. Проекция графа категорий

В этом случае нам на самом деле не нужно создавать граф сходства – вместо этого мы можем запустить на проекции графа категорий алгоритм выделения сообществ, такой как распространение метки. Использование подобного алгоритма позволит эффективно объединить предприятия в суперкатегорию, с которой у них больше всего общего:

```
CALL algo.labelPropagation.stream(
  'MATCH (c:Category) RETURN id(c) AS id',
  'MATCH (c1:Category)<-[:IN_CATEGORY]-()-[:IN_CATEGORY]->(c2:Category)
   WHERE id(c1) < id(c2)
   RETURN id(c1) AS source, id(c2) AS target, count(*) AS weight',
 {graph: "cypher"}
)
YIELD nodeId, label
MATCH (c:Category) WHERE id(c) = nodeId
MERGE (sc:SuperCategory {name: "SuperCategory-" + label})
MERGE (c)-[:IN_SUPER_CATEGORY]->(sc)
```

Давайте дадим этим суперкатегориям более понятное имя – здесь хорошо работает название их самой большой категории:

```
MATCH (sc:SuperCategory)<-[:IN_SUPER_CATEGORY]-(category)
WITH sc, category, size((category)<-[:IN_CATEGORY]-()) as size
ORDER BY size DESC
WITH sc, collect(category.name)[0] as biggestCategory
SET sc.friendlyName = "SuperCat " + biggestCategory
```

Пример категорий и суперкатегорий изображен на рис. 7.10.

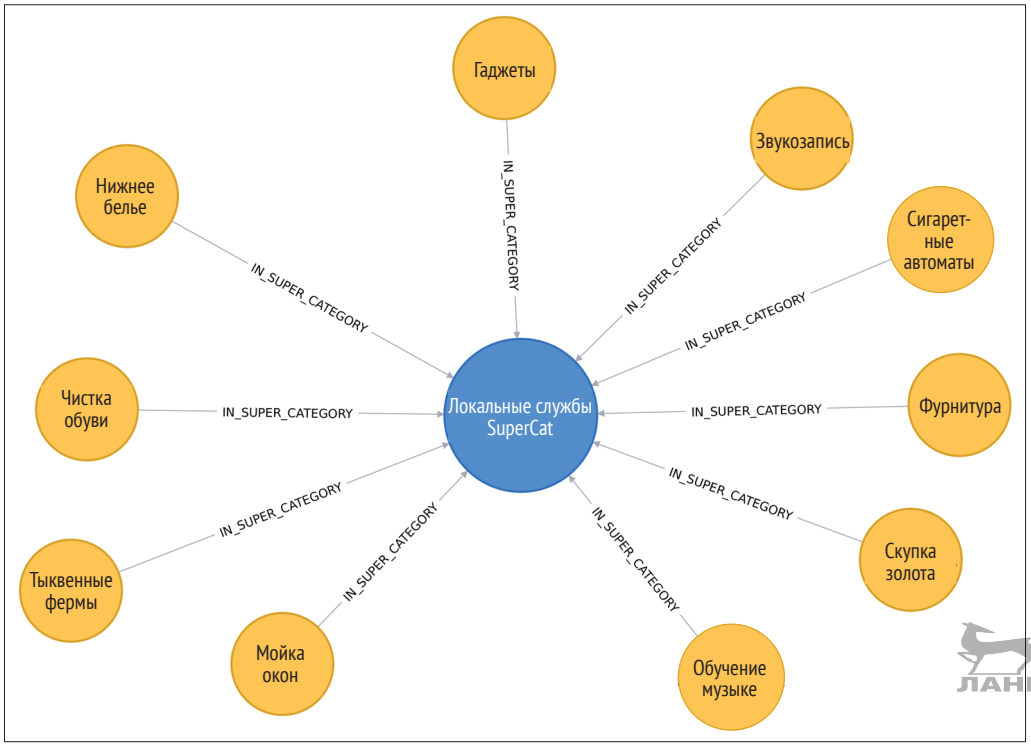


Рис. 7.10. Категории и суперкатегории

Следующий запрос находит наиболее распространенные похожие категории для отелей в Лас-Вегасе:

```
MATCH (hotels:Category {name: "Hotels"}),
      (lasVegas:City {name: "Las Vegas"}),
      (hotels)-[:IN_SUPER_CATEGORY]->()<-[:IN_SUPER_CATEGORY]-
                                        (otherCategory)
RETURN otherCategory.name AS otherCategory,
       size((otherCategory)<-[:IN_CATEGORY]-(:Business)-
                            [:IN_CITY]->(lasVegas)) AS businesses
ORDER BY count DESC
LIMIT 10
```

Выполнение запроса дает следующий результат:

otherCategory	business
Tours	189
Car Rental	160
Limos	84
Resorts	73
Airport Shuttles	52
Taxis	35
Vacation Rentals	29
Airports	25
Airlines	23
Motorcycle Rental	19

Эти результаты кажутся вам странными? Очевидно, что такси и туры не являются отелями, но помните, что этот список основан на самооценке принадлежности. Алгоритм распространения меток показывает нам смежные предприятия и услуги.

Теперь давайте найдем несколько компаний с рейтингом выше среднего в каждой из этих категорий:

```
// Ищем бизнес-проекты в Лас-Вегасе, которые входят в одну суперкатегорию с отелями
MATCH (hotels:Category {name: “Hotels”}),
      (hotels)-[:IN_SUPER_CATEGORY]->()<-[:IN_SUPER_CATEGORY]-
                                       (otherCategory),
      (otherCategory)<-[:IN_CATEGORY]-(business)
WHERE (business)-[:IN_CITY]->(:City {name: “Las Vegas”})

// Выбираем 10 случайных категорий и вычисляем рейтинг для 90-й процентили
WITH otherCategory, count(*) AS count,
     collect(business) AS businesses,
     percentileDisc(business.averageStars, 0.9) AS p90Stars
ORDER BY rand() DESC
LIMIT 10

// В каждой из этих категорий выбираем бизнес со средним рейтингом
// выше, чем у входящих в шаблон 90-й процентили
WITH otherCategory, [b in businesses where b.averageStars >= p90Stars]
                     AS businesses

// Выбираем один бизнес на категорию
WITH otherCategory, businesses[toInteger(rand() * size(businesses))] AS business

RETURN otherCategory.name AS otherCategory,
```

```
    business.name AS business,
    business.averageStars AS averageStars
```

В этом запросе мы впервые используем *отбор по шаблону* (pattern comprehension). Отбор по шаблону – это синтаксическая конструкция языка Cypher для создания списка на основе сопоставления с шаблоном. Она находит указанный шаблон, используя оператор `MATCH` вместе с оператором `WHERE`, а затем выдает пользовательскую проекцию. Эта функция Cypher была добавлена как развитие идей GraphQL – языка запросов для API.

Выполнив этот запрос, мы увидим следующий результат:

otherCategory	business	averageStars
Motorcycle Rental	Adrenaline Rush Slingshot Rentals	5.0
Snorkeling	Sin City Scuba	5.0
Guest Houses	Hotel Del Kacvinsky	5.0
Car Rental	The Lead Team	5.0
Food Tours	Taste BUZZ Food Tours	5.0
Airports	Signature Flight Support	5.0
Public Transportation	JetSuiteX	4.6875
Ski Resorts	Trikke Las Vegas	4.833333333333332
Town Car Service	MW Travel Vegas	4.866666666666665
Campgrounds	McWilliams Campground	3.875

Теперь мы можем вырабатывать рекомендации в режиме реального времени, основываясь на непосредственном поведении пользователя в приложении. Например, пока пользователи смотрят отели в Лас-Вегасе, мы можем выделить множество смежных бизнес-проектов с хорошими рейтингами. Мы можем применить этот подход в любом городе и для любой бизнес-категории, например ресторанов или театров.

Упражнения для читателя

- Можете ли вы построить диаграмму изменения отзывов о гостиницах города с течением времени?
- А как насчет конкретного отеля или другого бизнеса?
- Наблюдаются ли в изменениях популярности какие-либо тенденции – сезонные или иные?
- Связаны ли наиболее влиятельные обозреватели (исходящие связи) только с другими влиятельными обозревателями?

Анализ данных о рейсах авиакомпании с помощью Apache Spark

В этом разделе мы воспользуемся другим сценарием для иллюстрации анализа данных аэропортов США с помощью Spark. Представьте, что вы специалист по обработке данных со сложным расписанием поездок и хотели бы получить информацию о рейсах и задержках вылетов. Сначала мы исследуем информацию об аэропортах и рейсах, а затем углубимся в анализ задержек в двух конкретных аэропортах. Для анализа маршрутов и оптимального использования маршрутных точек часто летающими пассажирами мы будем использовать алгоритм выделения сообществ.

Бюро статистики транспорта США предоставляет значительный объем транспортной информации. В качестве примера данных для анализа мы будем использовать их данные о времени выполнения авиаперелетов в мае 2018 года, которые включают рейсы, начинающиеся и заканчивающиеся в Соединенных Штатах. Чтобы добавить более подробную информацию об аэропортах, например информацию о местоположении, мы также загрузим данные из отдельного источника OpenFlights (https://openflights.org/data.html).

Давайте загрузим данные в Spark. Как и в предыдущих разделах, наши данные находятся в файлах CSV, которые доступны в файловом архиве книги.

```
nodes = spark.read.csv("data/airports.csv", header=False)

cleaned_nodes = (nodes.select("_c1", "_c3", "_c4", "_c6", "_c7")
                  .filter("_c3 = 'United States'")
                  .withColumnRenamed("_c1", "name")
                  .withColumnRenamed("_c4", "id")s
                  .withColumnRenamed("_c6", "latitude")
                  .withColumnRenamed("_c7", "longitude")
                  .drop("_c3"))
cleaned_nodes = cleaned_nodes[cleaned_nodes["id"] != "\\N"]

relationships = spark.read.csv("data/188591317_T_ONTIME.csv", header=True)

cleaned_relationships = (relationships
                         .select("ORIGIN", "DEST", "FL_DATE", "DEP_DELAY",
                                 "ARR_DELAY", "DISTANCE", "TAIL_NUM", "FL_NUM",
                                 "CRS_DEP_TIME", "CRS_ARR_TIME",
                                 "UNIQUE_CARRIER")
                         .withColumnRenamed("ORIGIN", "src")
                         .withColumnRenamed("DEST", "dst")
                         .withColumnRenamed("DEP_DELAY", "deptDelay")
```

```
                        .withColumnRenamed("ARR_DELAY", "arrDelay")
                        .withColumnRenamed("TAIL_NUM", "tailNumber")
                        .withColumnRenamed("FL_NUM", "flightNumber")
                        .withColumnRenamed("FL_DATE", "date")
                        .withColumnRenamed("CRS_DEP_TIME", "time")
                        .withColumnRenamed("CRS_ARR_TIME", "arrivalTime")
                        .withColumnRenamed("DISTANCE", "distance")
                        .withColumnRenamed("UNIQUE_CARRIER", "airline")
                        .withColumn("deptDelay",
                             F.col("deptDelay").cast(FloatType()))
                        .withColumn("arrDelay",
                             F.col("arrDelay").cast(FloatType()))
                        .withColumn("time", F.col("time").cast(IntegerType()))
                        .withColumn("arrivalTime",
                             F.col("arrivalTime").cast(IntegerType()))
                        )

g = GraphFrame(cleaned_nodes, cleaned_relationships)
```

Мы должны выполнить небольшую очистку вершин, потому что некоторые аэропорты не имеют действительного кода. Дадим столбцам более описательные имена и преобразуем некоторые элементы в соответствующие числовые типы. Нам также необходимо убедиться, что у нас есть столбцы с именами `id`, `dst` и `src`, как того ожидает библиотека Spark GraphFrames.

Затем мы создадим отдельный DataFrame, который отображает коды авиакомпаний на названия авиакомпаний. Он понадобится нам позже в этой главе:

```
airlines_reference = (spark.read.csv("data/airlines.csv")
      .select("_c1", "_c3")
      .withColumnRenamed("_c1", "name")
      .withColumnRenamed("_c3", "code"))

airlines_reference = airlines_reference[airlines_reference["code"] != "null"]
```

Предварительный анализ

Давайте начнем с небольшого предварительного анализа, чтобы получить представление о том, как выглядят данные.

Сначала посмотрим, сколько у нас аэропортов:

```
g.vertices.count()

1435
```

И сколько у нас связей между этими аэропортами?

```
g.edges.count()
```

```
616529
```

Популярные аэропорты

Из каких аэропортов совершается больше всего вылетов? Мы можем рассчитать количество исходящих рейсов из аэропорта, используя алгоритм степенной централизации:

```
airports_degree = g.outDegrees.withColumnRenamed("id", "oId")

full_airports_degree = (airports_degree
                        .join(g.vertices, airports_degree.oId == g.vertices.id)
                        .sort("outDegree", ascending=False)
                        .select("id", "name", "outDegree"))

full_airports_degree.show(n=10, truncate=False)
```

Запустив код, получим следующий выход:

id	name	outDegree
ATL	Hartsfield Jackson Atlanta International Airport	33837
ORD	Chicago O'Hare International Airport	28338
DFW	Dallas Fort Worth International Airport	23765
CLT	Charlotte Douglas International Airport	20251
DEN	Denver International Airport	19836
LAX	Los Angeles International Airport	19059
PHX	Phoenix Sky Harbor International Airport	15103
SFO	San Francisco International Airport	14934
LGA	La Guardia Airport	14709
IAH	George Bush Intercontinental Houston Airport	14407

В этом списке фигурируют большинство крупных городов США. Действительно, в Чикаго, Атланте, Лос-Анджелесе и Нью-Йорке есть очень популярные аэропорты. Мы также можем визуализировать информацию об исходящих рейсах, используя следующий код:

```
plt.style.use('fivethirtyeight')

ax = (full_airports_degree
```

```
        .toPandas()
        .head(10)
        .plot(kind='bar', x='id', y='outDegree', legend=None))

ax.xaxis.set_label_text("")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

Получившаяся диаграмма изображена на рис. 7.11.

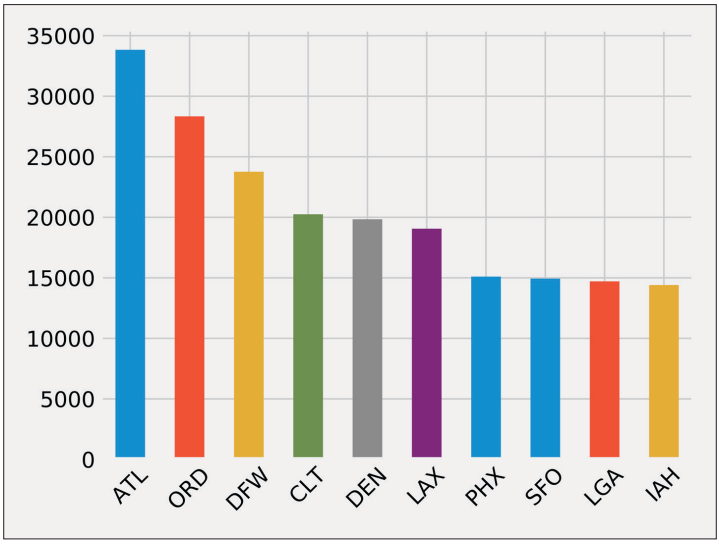


Рис. 7.11. Распределение вылетающих рейсов по аэропортам

Поразительно, как резко снижается количество рейсов. Международный аэропорт Денвера (DEN), пятый по популярности аэропорт, имеет в два раза меньше исходящих вылетов, чем лидер – международный аэропорт Хартсфилд Джексон в Атланте (ATL).

Задержки вылетов из аэропорта Чикаго

В нашем сценарии мы часто путешествуем между западным и восточным побережьями США и хотим видеть задержки в промежуточном хабе – международном аэропорту О'Хара в Чикаго (ORD). Демонстрационный набор данных содержит информацию о задержке рейсов, поэтому мы можем немедленно погрузиться в работу.

Следующий код определяет среднюю задержку рейсов, вылетающих из ORD и сгруппированных по аэропорту назначения:

```
delayed_flights = (g.edges
                   .filter("src = 'ORD' and deptDelay > 0")
```

```
                    .groupBy("dst")
                    .agg(F.avg("deptDelay"), F.count("deptDelay"))
                    .withColumn("averageDelay",
                                F.round(F.col("avg(deptDelay)"), 2))
                    .withColumn("numberOfDelays",
                                F.col("count(deptDelay)")))
(delayed_flights
 .join(g.vertices, delayed_flights.dst == g.vertices.id)
 .sort(F.desc("averageDelay"))
 .select("dst", "name", "averageDelay", "numberOfDelays")
 .show(n=10, truncate=False))
```

Рассчитав среднюю задержку, сгруппированную по месту назначения, мы объединяем итоговый набор DataFrame с набором DataFrame, содержащим все вершины, чтобы напечатать полное название аэропорта назначения.

Запуск этого кода возвращает 10 пунктов назначения с худшими задержками:

dst	name	averageDelay	numberOfDelays
CKB	North Central West Virginia Airport	145.08	12
OGG	Kahului Airport	119.67	9
MQT	Sawyer International Airport	114.75	12
MOB	Mobile Regional Airport	102.2	10
TTN	Trenton Mercer Airport	101.18	17
AVL	Asheville Regional Airport	98.5	28
ISP	Long Island Mac Arthur Airport	94.08	13
ANC	Ted Stevens Anchorage International Airport	83.74	23
BTV	Burlington International Airport	83.2	25
CMX	Houghton County Memorial Airport	79.18	17

Это любопытно – одна точка данных действительно выделяется: 12 рейсов из ORD в CKB были задержаны в среднем более чем на два часа! Давайте найдем рейсы между этими аэропортами и посмотрим, что там происходит:

```
from_expr = 'id = "ORD"'
to_expr = 'id = "CKB"'
ord_to_ckb = g.bfs(from_expr, to_expr)

ord_to_ckb = ord_to_ckb.select(
  F.col("e0.date"),
```

```
  F.col("e0.time"),
  F.col("e0.flightNumber"),
  F.col("e0.deptDelay"))
```

Затем нарисуем диаграмму рейсов:

```
ax = (ord_to_ckb
      .sort("date")
      .toPandas()
      .plot(kind='bar', x='date', y='deptDelay', legend=None))

ax.xaxis.set_label_text("")
plt.tight_layout()
plt.show()
```

Запустив этот код, мы получим диаграмму, изображенную на рис. 7.12.

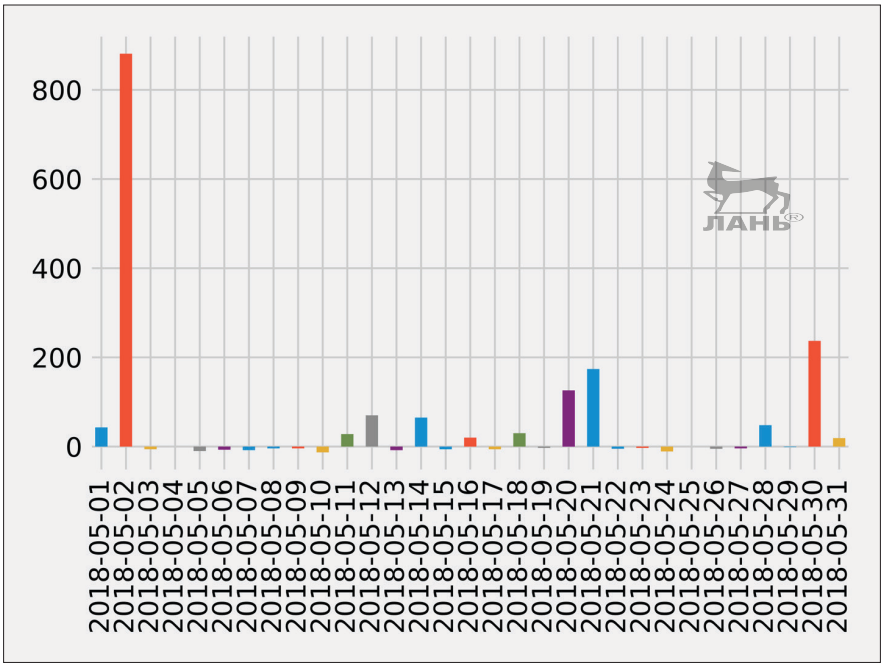


Рис. 7.12. Рейсы из аэропорта ORD в CKB

Действительно, около половины рейсов были задержаны, но задержка более чем на 14 часов 2 мая 2018 года существенно исказила среднее значение.

Что, если мы хотим обнаружить задержки входящих и исходящих рейсов прибрежного аэропорта? Эти аэропорты часто страдают от неблагоприятных погодных условий, поэтому мы могли бы найти некоторые интересные задержки.

Плохой день в Сан-Франциско

Давайте проанализируем задержки в аэропорту, известном проблемами с «низким потолком» тумана – международном аэропорту Сан-Франциско (SFO). Одним из методов анализа может быть изучение так называемых *мотивов* (motif), представляющих собой повторяющиеся подграфы или шаблоны.

Эквивалентом мотивов в Neo4j являются графовые шаблоны, которые можно найти с помощью оператора `MATCH` или с помощью шаблонных выражений в Cypher.

GraphFrames позволяет нам искать мотивы, поэтому мы можем использовать структуру рейсов как часть запроса. Давайте воспользуемся мотивами, чтобы найти самые задерживаемые рейсы, прибывающие и вылетающие из SFO 11 мая 2018 года. Следующий код найдет эти задержки:

```
motifs = (g.find("(a)-[ab]->(b); (b)-[bc]->(c)")
          .filter("""(b.id = 'SFO') and
                  (ab.date = '2018-05-11' and bc.date = '2018-05-11') and
                  (ab.arrDelay > 30 or bc.deptDelay > 30) and
                  (ab.flightNumber = bc.flightNumber) and
                  (ab.airline = bc.airline) and
                  (ab.time < bc.time)"""))
```


Мотив `(a)-[ab]->(б); (b)-[bc]->(c)` находит рейсы, прибывающие и вылетающие из одного и того же аэропорта. Затем мы отфильтруем полученный шаблон, чтобы найти рейсы, удовлетворяющие условиям:

- последовательность, при которой первый рейс прибывает в SFO, а второй рейс отправляется из SFO;
- задержка более 30 минут при прибытии в SFO или отбытии из него;
- тот же самый номер рейса и авиакомпания.

Теперь мы можем взять результат и выбрать интересующие нас столбцы:

```
result = (motifs.withColumn("delta", motifs.bc.deptDelay - motifs.ab.arrDelay)
          .select("ab", "bc", "delta")
          .sort("delta", ascending=False))

result.select(
    F.col("ab.src").alias("a1"),
    F.col("ab.time").alias("a1DeptTime"),
```

```
    F.col("ab.arrDelay"),
    F.col("ab.dst").alias("a2"),
    F.col("bc.time").alias("a2DeptTime"),
    F.col("bc.deptDelay"),
    F.col("bc.dst").alias("a3"),
    F.col("ab.airline"),
    F.col("ab.flightNumber"),
    F.col("delta")
).show()
```

Мы также вычисляем разницу между прибывающими и отправляющимися рейсами, чтобы увидеть, какие задержки действительно относятся к SFO.

Выполнив этот код, получим следующий результат:

airline	flightNumber	a1	a1DeptTime	arrDelay	a2	a2DeptTime	deptDelay	a3	delta
WN	1454	PDX	1130	–18.0	SFO	1350	178.0	BUR	196.0
OO	5700	ACV	1755	–9.0	SFO	2235	64.0	RDM	73.0
UA	753	BWI	700	–3.0	SFO	1125	49.0	IAD	52.0
UA	1900	ATL	740	40.0	SFO	1110	77.0	SAN	37.0
WN	157	BUR	1405	25.0	SFO	1600	39.0	PDX	14.0
DL	745	DTW	835	34.0	SFO	1135	44.0	DTW	10.0
WN	1783	DEN	1830	25.0	SFO	2045	33.0	BUR	8.0
WN	5789	PDX	1855	119.0	SFO	2120	117.0	DEN	–2.0
WN	1585	BUR	2025	31.0	SFO	2230	11.0	PHX	–20.0

Наихудший бездельник, рейс WN 1454, показан в верхнем ряду – он прибыл рано, но улетел почти на три часа позже. Мы также видим, что в столбце `arrDelay` есть некоторые отрицательные значения; это означает, что рейс прибыл в SFO раньше времени. Также обратите внимание на то, что некоторые рейсы, такие как WN 5789 и WN 1585, сократили простой на земле в SFO, чему соответствует отрицательная дельта.

Взаимосвязи аэропортов через авиакомпании

Теперь предположим, что мы много путешествовали и у нас скоро закончатся бонусы для часто летающих пассажиров, поэтому мы хотим использовать остаток наиболее эффективно. Если начнем с конкретного аэропорта США, сколько разных аэропортов мы сможем посетить и вернуться в исходный аэропорт, используя одну и ту же авиакомпанию?

Давайте сначала составим список авиакомпаний и выясним, сколько рейсов приходится на каждую из них:

```
airlines = (g.edges
 .groupBy("airline")
 .agg(F.count("airline").alias("flights"))
 .sort("flights", ascending=False))

full_name_airlines = (airlines_reference
                      .join(airlines, airlines.airline
                            == airlines_reference.code)
                      .select("code", "name", "flights"))
```

А теперь нарисуем гистограмму, показывающую наши авиакомпании:

```
ax = (full_name_airlines.toPandas()
      .plot(kind='bar', x='name', y='flights', legend=None))

ax.xaxis.set_label_text("")
plt.tight_layout()
plt.show()
```

Выполнив этот запрос, мы получим диаграмму, изображенную на рис. 7.13.

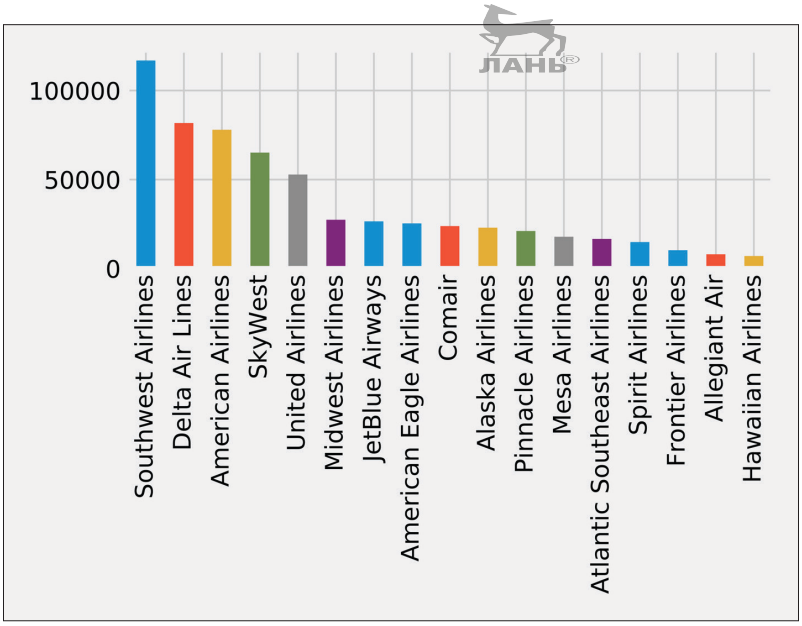


Рис. 7.13. Количество рейсов по авиакомпаниям

А теперь напишем функцию, которая использует алгоритм сильно связанных компонентов для поиска таких групп аэропортов для каждой авиакомпании, где все аэропорты имеют рейсы туда и обратно для всех других аэропортов этой группы:

```
def find_scc_components(g, airline):
    # Создаем подграф, содержащий рейсы только заданной авиакомпании.
    airline_relationships = g.edges[g.edges.airline == airline]
    airline_graph = GraphFrame(g.vertices, airline_relationships)

    # Вычисляем сильно связанные компоненты.
    scc = airline_graph.stronglyConnectedComponents(maxIter=10)

    # Находим размер наибольшего компонента и возвращаем его
    return (scc
        .groupBy("component")
        .agg(F.count("id").alias("size"))
        .sort("size", ascending=False)
        .take(1)[0]["size"])
```

Мы можем написать следующий код для создания набора DataFrame, содержащего каждую авиакомпанию и количество аэропортов ее крупнейшего сильно связанного компонента:

```
# Вычисляем крупнейший сильно связанный компонент (ССК) для каждой авиакомпании
airline_scc = [(airline, find_scc_components(g, airline))
               for airline in airlines.toPandas()["airline"].tolist()]
airline_scc_df = spark.createDataFrame(airline_scc, ['id', 'sccCount'])

# Связываем наборы DataFrame ССК и DataFrame авиакомпаний, чтобы иметь
# возможность показать количество рейсов авиакомпании одновременно
# с аэропортами, доступными для сильно связанного компонента
airline_reach = (airline_scc_df
 .join(full_name_airlines, full_name_airlines.code == airline_scc_df.id)
 .select("code", "name", "flights", "sccCount")
 .sort("sccCount", ascending=False))
```

А теперь нарисуем гистограмму, показывающую наши авиакомпании:

```
ax = (airline_reach.toPandas()
      .plot(kind='bar', x='name', y='sccCount', legend=None))

ax.xaxis.set_label_text("")
plt.tight_layout()
plt.show()
```

Выполнив этот запрос, мы получим диаграмму, изображенную на рис. 7.14.

У авиакомпании SkyWest есть самое большое сообщество, с более чем 200 сильно связанными аэропортами. Это может частично отражать ее бизнес-модель в качестве партнерской (шеринговой) авиакомпании, ко-

торая эксплуатирует самолеты, выполняющие рейсы авиакомпаний-партнеров. Авиакомпания Southwest, с другой стороны, имеет наибольшее количество рейсов, но соединяет только порядка 80 аэропортов.

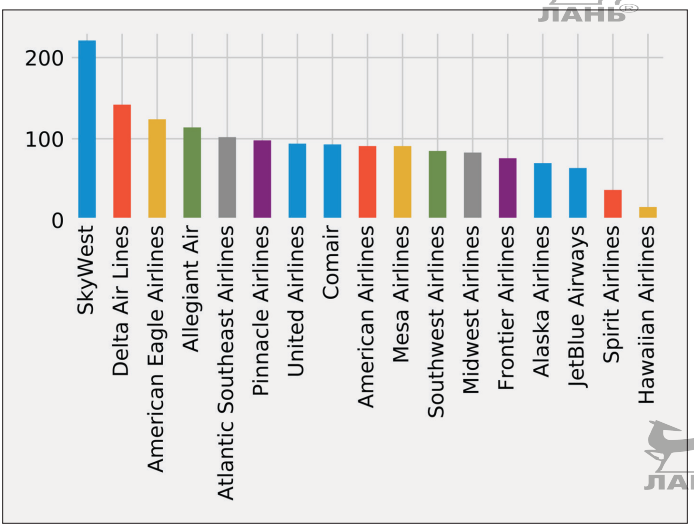


Рис. 7.14. Количество доступных аэропортов по авиакомпаниям

Теперь допустим, что наибольшее количество бонусов для часто летающих пассажиров у нас есть от компании Delta Airlines (DL). Можем ли мы найти аэропорты, которые образуют сообщество для этого конкретного авиаперевозчика?

```
airline_relationships = g.edges.filter("airline = 'DL'")
airline_graph = GraphFrame(g.vertices, airline_relationships)

clusters = airline_graph.labelPropagation(maxIter=10)
(clusters
 .sort("label")
 .groupby("label")
 .agg(F.collect_list("id").alias("airports"),
      F.count("id").alias("count"))
 .sort("count", ascending=False)
 .show(truncate=70, n=10))
```

Выполнив этот запрос, мы получим следующий вывод:

label	airports	count
1606317768706	[IND, ORF, ATW, RIC, TRI, XNA, ECP, AVL, JAX, SYR, BHM, GSO, MEM, C...	89
1219770712067	[GEG, SLC, DTW, LAS, SEA, BOS, MSN, SNA, JFK, TVC, LIH, JAC, FLL, M...	53
17179869187	[RHV]	1

25769803777	[CWT]	1
25769803776	[CDW]	1
25769803782	[KNW]	1
25769803778	[DRT]	1
25769803779	[FOK]	1
25769803781	[HVR]	1
42949672962	[GTF]	1

Большинство аэропортов, используемых DL, объединены в две группы; давайте изучим их более детально. Здесь слишком много аэропортов, поэтому мы просто покажем аэропорты с наибольшей степенью (количество входящих и исходящих рейсов). Мы можем вычислить степень аэропорта при помощи одной строки кода:

```
all_flights = g.degrees.withColumnRenamed("id", "aId")
```

Затем объединяем результат с аэропортами, которые принадлежат к крупнейшему кластеру:

```
(clusters
 .filter("label=1606317768706")
 .join(all_flights, all_flights.aId == result.id)
 .sort("degree", ascending=False)
 .select("id", "name", "degree")
 .show(truncate=False))
```

Выполнив этот запрос, мы получим следующий вывод:

id	name	degree
DFW	Dallas Fort Worth International Airport	47514
CLT	Charlotte Douglas International Airport	40495
IAH	George Bush Intercontinental Houston Airport	28814
EWR	Newark Liberty International Airport	25131
PHL	Philadelphia International Airport	20804
BWI	Baltimore/Washington International Thurgood Marshall Airport	18989
MDW	Chicago Midway International Airport	15178
BNA	Nashville International Airport	12455
DAL	Dallas Love Field	12084
IAD	Washington Dulles International Airport	11566
STL	Lambert St Louis International Airport	11439

HOU	William P Hobby Airport	9742
IND	Indianapolis International Airport	8543
PIT	Pittsburgh International Airport	8410
CLE	Cleveland Hopkins International Airport	8238
CMH	Port Columbus International Airport	7640
SAT	San Antonio International Airport	6532
JAX	Jacksonville International Airport	5495
BDL	Bradley International Airport	4866
RSW	Southwest Florida International Airport	4569

На рис 7.15 хорошо видно, что этот кластер сосредоточен от Восточного побережья до Среднего Запада Соединенных Штатов.

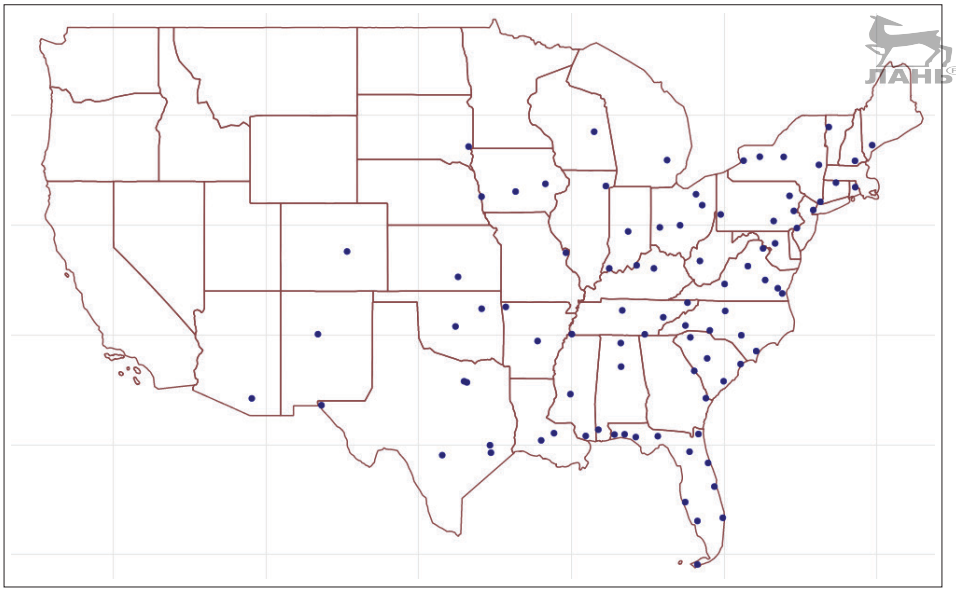


Рис. 7.15. Кластер аэропортов с меткой 1606317768706

А теперь давайте проделаем то же самое со вторым по величине кластером:

```
(clusters
 .filter("label=1219770712067")
 .join(all_flights, all_flights.aId == result.id)
 .sort("degree", ascending=False)
 .select("id", "name", "degree")
 .show(truncate=False))
```

Выполнение этого запроса даст нам следующий вывод:

id	name	degree
ATL	Hartsfield Jackson Atlanta International Airport	67672
ORD	Chicago O'Hare International Airport	56681
DEN	Denver International Airport	39671
LAX	Los Angeles International Airport	38116
PHX	Phoenix Sky Harbor International Airport	30206
SFO	San Francisco International Airport	29865
LGA	La Guardia Airport	29416
LAS	McCarran International Airport	27801
DTW	Detroit Metropolitan Wayne County Airport	27477
MSP	Minneapolis-St Paul International/Wold-Chamberlain Airport	27163
BOS	General Edward Lawrence Logan International Airport	26214
SEA	Seattle Tacoma International Airport	24098
MCO	Orlando International Airport	23442
JFK	John F Kennedy International Airport	22294
DCA	Ronald Reagan Washington National Airport	22244
SLC	Salt Lake City International Airport	18661
FLL	Fort Lauderdale Hollywood International Airport	16364
SAN	San Diego International Airport	15401
MIA	Miami International Airport	14869
TPA	Tampa International Airport	12509

На рис 7.16 мы видим, что этот кластер, вероятно, имеет более выраженные хабы с несколькими промежуточными северо-западными пересадками.

Код, который мы использовали для создания этих карт, доступен в файловом архиве книги. Проверяя веб-сайт DL на наличие программ для часто летающих пассажиров, мы замечаем акцию «Каждый третий полет бесплатно». Если мы используем наши бонусы для оплаты двух рейсов, то получим еще один полет бесплатно, но только если будем летать в одном из двух кластеров! Очевидно, что лучше использовать наше время и, конечно, наши бонусы таким образом, чтобы оставаться в кластере.

Упражнения для читателя

- Используйте алгоритм кратчайшего пути, чтобы оценить количество рейсов из произвольного аэропорта США в международный аэропорт Бозман Йеллоустоун (BZN).
- Есть ли различия, если вы используете весовые коэффициенты?

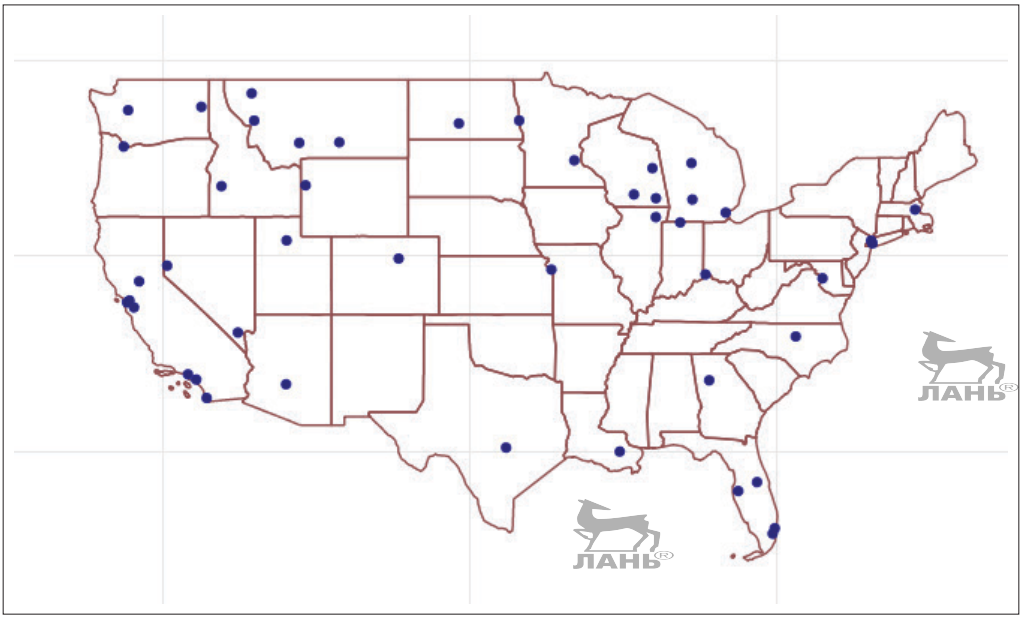


Рис. 7.16. Кластер аэропортов с меткой 1219770712067

Заключение

В нескольких предыдущих главах мы подробно рассказали о том, как работают ключевые графовые алгоритмы для поиска путей, вычисления централизации и выделения сообществ на платформах Apache Spark и Neo4j. В этой главе мы рассмотрели рабочие процессы, включающие использование нескольких алгоритмов в контексте с другими задачами и анализом. Мы использовали сценарий туристического бизнеса для анализа данных Yelp в Neo4j и сценарий личных авиаперелетов для анализа данных американских авиакомпаний в Spark.

Далее будет рассмотрено использование графовых алгоритмов, которое приобретает все большее значение: машинное обучение с графовым дополнением.

Глава 8

Графовые алгоритмы и машинное обучение

В предыдущих главах мы рассмотрели несколько алгоритмов, изучающих и обновляющих состояние графа на каждой итерации, – например, алгоритм распространения меток; однако до этого момента мы уделяли особое внимание графовым алгоритмам для общего применения. Поскольку графы все шире применяются в *машинном обучении* (machine learning, ML), далее мы рассмотрим использование графовых алгоритмов для совершенствования процессов ML.

В этой главе мы сосредоточимся на наиболее практичном способе улучшения прогнозов ML с использованием графовых алгоритмов – извлечении связанных признаков и использовании их в прогнозировании связей. Сначала рассмотрим некоторые базовые концепции ML и важность контекстных данных для улучшения прогнозирования. Затем кратко обсудим практические способы применения графовых признаков, включая обнаружение спама и прогнозирование связей.

Мы покажем, как создать конвейер машинного обучения, а затем обучить и оценить модель для прогнозирования связей, опираясь на платформы Neo4j и Spark. Наш пример будет основан на наборе данных сети цитирования, который содержит авторов, статьи, отношения авторов и отношения цитирования. Мы воспользуемся несколькими моделями, чтобы предсказать, могут ли авторы исследования сотрудничать в будущем, и покажем, как графовые алгоритмы улучшают результаты предсказаний.

Машинное обучение и важность контекста

Машинное обучение – это не искусственный интеллект (ИИ), а метод достижения ИИ. ML использует алгоритмы для обучения программного обеспечения с помощью конкретных примеров и накопительных улучшений, основанных на ожидаемых результатах, без явного программирования того, как добиться этих лучших результатов. Обучение подразумевает предоставление большого количества данных для модели и позволяет ей научиться обрабатывать и использовать эту информацию.

С этой точки зрения машинное обучение означает, что алгоритмы повторяются, постоянно внося небольшие изменения, чтобы приблизиться к объективной цели, например уменьшить ошибку классификации по сравнению с обучающими данными. ML работает динамично и масштабируемо, обладая возможностью изменять и оптимизировать себя, когда представлено больше данных. Оптимизация может происходить как во время обучения перед использованием, так и в режиме онлайн во время использования.

Недавние успехи в обучении ML, а также увеличение доступности больших наборов данных и мощности параллельных вычислений повысили целесообразность применения ML с точки зрения разработчиков вероятностных моделей в приложениях ИИ. Поскольку машинное обучение становится все более распространенным, важно помнить о его основной цели – делать выбор так же, как делают люди. Если мы забудем эту цель, у нас может получиться еще одна версия узкопрофильного программного обеспечения, основанного на правилах.

Чтобы повысить точность машинного обучения и расширить охват наших приложений, нам необходимо использовать много контекстуальной информации – так же, как люди должны учитывать контекст для принятия лучших решений. Люди используют не только непосредственные точки данных, но и окружающий их контекст, чтобы выяснить, что важно в данной ситуации, оценить недостающую информацию и определить, как применять навыки в новых ситуациях. Контекст помогает нам улучшить прогнозы.

Графы, контекст и точность

Решения, которые пытаются предсказать поведение или дать рекомендации для различных обстоятельств, но не располагают побочной контекстной информацией, требуют более тщательного обучения и предписывающих правил. Это частично объясняет, почему ИИ хорош в отдельных, четко определенных задачах, но плохо работает с неоднозначностью. Графовое расширение ML способно извлечь из данных недостающую контекстную информацию, которая так важна для принятия лучших решений.

Мы знаем из теории графов и из реальной жизни, что отношения часто являются самыми сильными предикторами поведения. Например, если один человек сходит на выборы и расскажет об этом, возрастет вероятность того, что его друзья, семья и даже коллеги тоже проголосуют. Рисунок 8.1 иллюстрирует волновой эффект на основе сообщений о голосовании и связей друзей из Facebook, описанный в 2012 году в исследовательской работе[1] «Эксперимент на 61 млн человек в области социального влияния и политической мобилизации», Р. Бонд и др.

[1] *R. Bond et al.*, «A 61-Million-Person Experiment in Social Influence and Political Mobilization», https://www.nature.com/articles/nature11421.

Рис. 8.1. Люди находятся под влиянием голосования в своих социальных сетях. В этом примере друзья в двух переходах оказали большее общее влияние, чем прямые отношения

Авторы обнаружили, что друзья, сообщившие о голосовании, оказали влияние на еще 1,4% пользователей, просто сказав, что они проголосовали, и, что особенно интересно, друзья друзей добавили еще 1,7%. Небольшой процент может оказать значительное влияние, и мы видим на рис. 8.1, что люди, расположенные на расстоянии двух переходов, оказали в целом большее влияние, чем просто прямые друзья. Голосование и другие примеры того, как наши социальные сети влияют на нас, освещены в книге[2] «Связанные» Николаса Кристакиса и Джеймса Фаулера.

Добавление графовых признаков и контекста улучшает предсказания, особенно в ситуациях, когда важны связи. Например, розничные компании персонализируют рекомендации по продуктам, используя не только историю покупок, но и контекстные данные о сходстве клиентов и поведении в интернете. Речевой помощник Alexa от компании Amazon использует многоуровневую контекстную модель, которая демонстрирует повышенную точность. В 2018 году Amazon также ввела «перенос контекста» для использования предыдущих тем разговора при ответе на новые вопросы.

К сожалению, многие подходы машинного обучения сегодня упускают богатую контекстную информацию. Это связано с тем, что ML полагается на обучающие данные, построенные из кортежей и исключающие множество прогнозирующих связей и сетевых данных. Кроме того, контекстная информация не всегда легкодоступна или слишком трудна для извлечения и обработки. Даже нахождение контекстных связей, которые находятся на расстоянии четырех или более переходов, в масштабе больших данных может стать проблемой для традиционных методов. Используя графы, легче находить и использовать связанные данные.

[2] *Nicholas Christakis and James Fowler*, «Connected», на русском языке не издавалась. – *Прим. перев.*

Извлечение и отбор связанных признаков

Извлечение и отбор признаков помогает нам брать необработанные данные и создавать подходящие подмножества и формат для обучения наших моделей. Это важный этап, который при правильном выполнении приводит к ML, дающему более точные прогнозы.

Извлечение и отбор признаков

Извлечение признаков (feature extraction) – это способ переместить большие объемы данных и атрибутов в набор репрезентативных описательных атрибутов. Процесс выводит числовые значения (признаки) для отличительных характеристик или шаблонов во входных данных, чтобы по этим признакам мы могли дифференцировать категории в других данных. Данный подход используется, когда модели сложно анализировать данные напрямую – возможно, из-за размера, формата или необходимости случайных сравнений.

Отбор признаков (feature selection) – это процесс определения подмножества извлеченных признаков, которые являются наиболее важными или влияющими на цель. Он используется для выявления прогностической важности и эффективности. Например, если у нас есть 20 признаков и 13 из них вместе объясняют 92 % того, что нам нужно прогнозировать, можем исключить остальные 7 признаков из нашей модели.

Использование правильного сочетания признаков может повысить точность модели, так как это существенно влияет на то, как наши модели учатся. Поскольку в масштабах больших данных даже незначительные улучшения могут иметь существенное значение, в этой главе мы сосредоточимся на связанных признаках. *Связанные признаки* (connected features) – это признаки, извлеченные из структуры данных. Эти признаки могут быть получены из локальных графовых запросов на основе частей графа, окружающих вершину, или глобальных графовых запросов, использующих графовые алгоритмы для идентификации прогнозирующих элементов в данных на основе отношений для извлечения связанных признаков.

И здесь важно не только получить правильную комбинацию признаков, но и устранить ненужные признаки, чтобы уменьшить вероятность того, что наши модели будут гипернаправленными. Это спасает нас от создания моделей, которые хорошо работают только с обучающими данными (явление, известное как *переобучение* (overfitting)), и значительно расширяет область применения. Мы также можем использовать графовые алгоритмы, чтобы оценить эти признаки и определить, какие из них больше всего влияют на нашу модель для выбора связанных признаков. Например, мы можем сопоставить признаки с вершинами на графе, построить

ребра на основе одинаковых признаков, а затем вычислить центральность признаков. Связи признаков могут быть определены через способность сохранять плотности кластеров точек данных. Этот метод описывается с использованием наборов данных с высокой размерностью и малым размером выборки в статье[3] К. Хенниаба, Н. Мезгани и К. Гуин-Валлеранда «Неуправляемая графовая выборка признаков через подпространство и центральность PageRank».

Представление графа

Представление графа (graph embedding) – это представление вершин и ребер в графе в виде *векторов признаков* (feature vector). Это просто наборы признаков, которые имеют размерные отображения, такие как координаты (*x*, *y*, *z*), показанные на рис. 8.2.

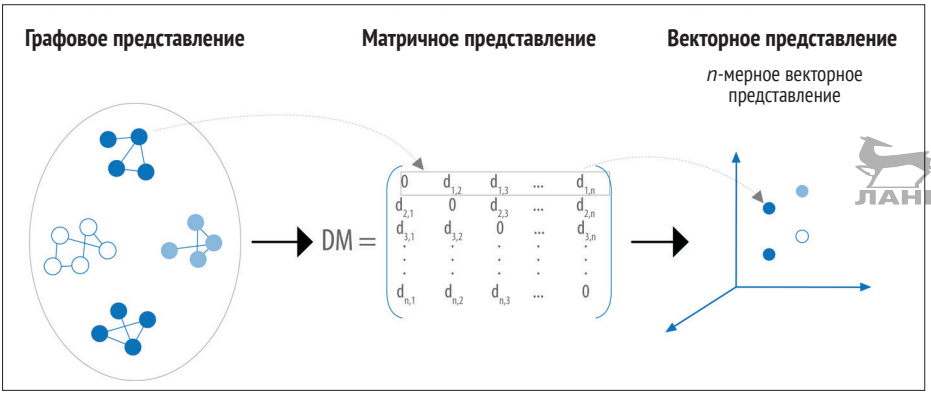


Рис. 8.2. Представление графа отображает данные графа в векторы объектов, которые можно визуализировать в пространстве многомерных координат

Представление графа использует данные графа немного иначе, чем при извлечении связанных признаков, и позволяет нам представлять целые графы или подмножества графовых данных в числовом формате, пригодном для задач машинного обучения. Это особенно полезно для обучения без учителя, когда данные не классифицируются, поскольку они извлекают больше контекстуальной информации через связи. Представление графа также полезно для исследования данных, вычисления сходства между объектами и уменьшения размерности в приложениях статистического анализа.

Это быстро развивающаяся область исследований и разработок с несколькими популярными решениями, включая node2vec, struc2vec, GraphSAGE, DeepWalk и DeepGL.

[3] *K. Henniab, N. Mezghani, and C. Gouin-Vallerand,* «Unsupervised Graph-Based Feature Selection Via Subspace and PageRank Centrality», https://bit.ly/2HGON5B.

Теперь давайте рассмотрим некоторые типы связанных признаков и то, как они используются.

Графовые признаки

Графовые признаки (graphy features) включают любое количество *обоснованных связями* (connection related) метрик нашего графа, таких как количество связей, входящих или выходящих из вершин, количество потенциальных треугольников и общих соседей. В нашем примере мы начнем с этих мер, потому что они просты для сбора и являются хорошим тестом ранних гипотез.

Кроме того, когда мы точно знаем, что ищем, мы можем использовать конструирование признаков. Например, нужно узнать, сколько людей имеют мошенническую учетную запись, располагающую связями на расстоянии до четырех переходов. Этот подход использует обход графа, эффективно находя глубокие пути связей и принимая в расчет такие вещи, как метки, атрибуты, количества и предполагаемые отношения.

Мы также можем легко автоматизировать эти процессы и внедрить прогнозирующие признаки графа в наш существующий конвейер обучения. Например, мы могли бы абстрагировать количество связей мошенников и добавить это число в качестве атрибута вершины, который будет использоваться для других задач машинного обучения.

Признаки и графовые алгоритмы

Также можно использовать графовые алгоритмы, чтобы найти признаки, применительно к которым мы знаем общую структуру, но не точную модель. Например, известно, что определенные типы группировок сообществ указывают на мошенничество – возможно, в них присутствует определенная плотность или иерархия отношений. В этом случае нам нужен не жесткий признак конкретной организации, а скорее гибкая и глобально применимая структура. Для извлечения связанных признаков в нашем примере мы будем использовать алгоритмы выделения сообществ, но кроме них часто применяются и алгоритмы централизации, такие как PageRank.

Кроме того, подходы, которые объединяют несколько типов связанных признаков, как правило, работают лучше, чем приверженность к одному единственному методу. Например, при предсказании мошенничества мы могли бы объединить связанные признаки с индикаторами, основанными на сообществах, найденных с помощью Лувенского алгоритма, влиятельными узлами, найденными при помощи PageRank, и количеством известных мошенников в трех переходах.

Комбинированный подход продемонстрирован на рис. 8.3, где авторы комбинируют графовые алгоритмы, такие как PageRank и Coloring,

с типичной графовой мерой, такой как входящая степень и исходящая степень вершины. Эта диаграмма взята из статьи[4] С. Факрая и др. «Совместное обнаружение спамеров в развивающихся многосвязных социальных сетях».

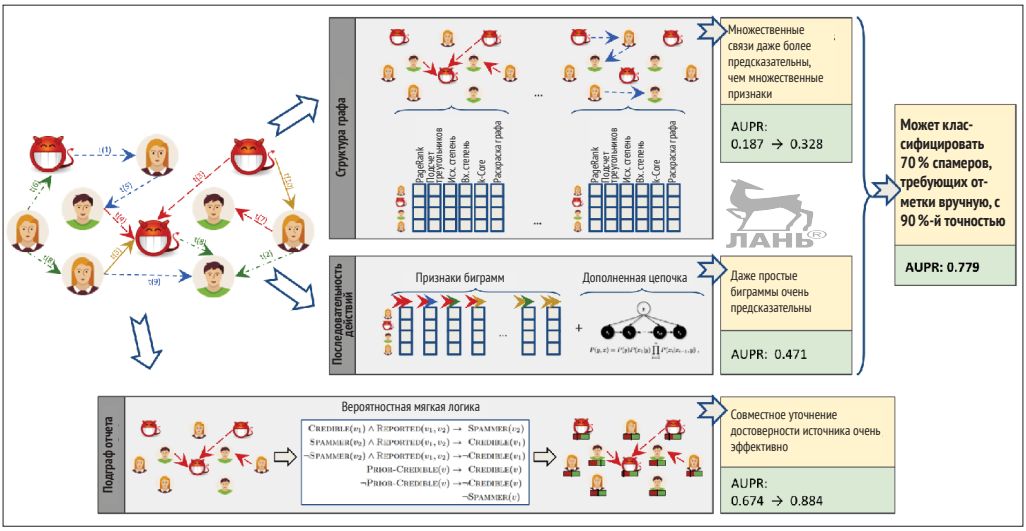


Рис. 8.3. Извлечение связанных признаков можно комбинировать с другими методами прогнозирования для улучшения результатов. AUPR соответствует пространству под кривой точного отклика, причем предпочтительными являются более высокие числа

Блок «Структура графа» иллюстрирует извлечение связанных признаков с использованием нескольких графовых алгоритмов. Интересно, что авторы обнаружили, что извлечение связанных признаков из множества типов связей даже более предсказательно, чем простое добавление новых признаков. В блоке «Подграф отчета» показано, как графовые признаки преобразуются в признаки, которые может использовать модель ML. Комбинируя несколько методов в дополненном графами рабочем процессе ML, авторы смогли улучшить предыдущие методы обнаружения и классифицировать 70 % спамеров, которые ранее требовали ручной маркировки, с точностью до 90 %.

Даже после того как мы извлекли связанные признаки, можно улучшить наше обучение, используя графовые алгоритмы, такие как PageRank, чтобы расставить приоритеты для признаков с наибольшим влиянием. Это позволяет нам адекватно представлять данные, исключая шумовые переменные, которые могут ухудшить результаты или замедлить обработку. С помощью этого типа информации мы также можем идентифицировать объекты с высокой степенью совпадения для дальнейшей настройки

[4] *S. Fakhraei et al.*, «Collective Spammer Detection in Evolving Multi-Relational Social Networks», https://bit.ly/2TyG6Mm.

модели посредством сокращения признаков. Упомянутый метод описан в исследовательской работе[5] «Использование PageRank в выборе признаков» Д. Айенко, Р. Мео и М. Ботта.

Мы обсудили, как связанные признаки применяются в сценариях, связанных с мошенничеством и обнаружением спама. В этих ситуациях действия часто скрыты в нескольких слоях обфускации и сетевых отношениях. Традиционные методы выделения и выбора признаков могут быть неспособны обнаружить это поведение без контекстной информации, которую представляют графы.

Но эта глава посвящена предсказанию связей – еще одной области, где связанные признаки улучшают машинное обучение. *Предсказание связей* (link prediction) – это способ оценить, какова вероятность того, что связь будет сформирована в будущем или что она должна быть на нашем графе, но отсутствует из-за неполных данных. Поскольку сети являются динамичными и могут расти довольно быстро, возможность предсказывать связи, которые вскоре появятся, имеет широкое применение: от рекомендаций новых товаров до смены предназначения лекарств и даже выявления криминальных отношений.

Связанные признаки из графов часто используются для улучшения предсказания связей с использованием базовых признаков графа, а также признаков, извлеченных алгоритмами центральности и выделения сообществ. Также стандартным методом является предсказание связи на основе близости или сходства вершин; в статье[6] «Проблема прогнозирования связей в социальных сетях» Д. Либен-Новелл и Дж. Кляйнберг предполагают, что одна только структура сети может содержать достаточно скрытой информации, чтобы обнаружить близость узлов и превзойти более прямые измерения.

Теперь, когда мы рассмотрели способы, которыми связанные признаки могут улучшить машинное обучение, давайте разберем предсказание связей на примере и посмотрим, как можно применять графовые алгоритмы для повышения точности предсказаний.

Графы и машинное обучение на практике: прогнозирование связей

В оставшейся части главы будет продемонстрирован практический пример, основанный на наборе данных Citation Network – исследовательском наборе данных, извлеченном из DBLP, ACM и MAG. Набор данных описан в статье[7] Дж. Танга и др. «ArnetMiner: извлечение и составление академи-

[5] *D. Ienco, R. Meo, and M. Botta*, «Using PageRank in Feature Selection», https://bit.ly/2JDDwVw.

[6] *D. Liben-Nowell and J. Kleinberg*, «The Link Prediction Problem for Social Networks», https://www.cs.cornell.edu/home/kleinber/link-pred.pdf.

[7] *J. Tang et al.*, «ArnetMiner: Extraction and Mining of Academic Social Networks», http://bit.ly/2U4C3fb.

ческих социальных сетей». Последняя версия содержит 3 079 007 статей, 1 766 547 авторов, 9 437 718 авторских отношений и 25 166 994 ссылки.

Мы будем работать с подмножеством, сосредоточенном на статьях, которые появились в следующих публикациях:

- конспекты лекций в области информатики;
- публикации ACM (Association for Computing Machinery, ассоциация вычислительной техники США);
- международные конференции по программированию;
- достижения в области вычислительной техники и связи;

Наш результирующий набор данных содержит 51 956 статей, 80 299 авторов, 140 575 связей между авторами и 28 706 отношений цитирования. Мы создадим граф соавторов, основанный на авторах, которые работали над статьями, а затем спрогнозируем будущее сотрудничество между парами авторов. Нас интересует только новое сотрудничество между авторами, которые раньше не сотрудничали, и не интересует повторение уже сложившегося сотрудничества.

В оставшейся части главы мы настроим необходимые инструменты и импортируем данные в Neo4j. Затем расскажем, как правильно сбалансировать данные и разбить выборки на блоки DataFrames для обучения и тестирования. После этого мы объясним нашу гипотезу и методы прогнозирования связей, а затем перейдем к созданию обучаемой модели в Spark. Наконец, проведем обучение и оценим различные прогнозные модели, начиная с базовых графовых признаков и добавляя дополнительные признаки, извлеченные с помощью Neo4j.

Инструменты и данные

Давайте начнем с настройки наших инструментов и данных. Затем изучим наш набор данных и создадим модель.

Сначала мы настроим библиотеки, используемые в этой главе:

- `py2neo` – библиотека Python Neo4j, которая хорошо интегрируется с экосистемой науки о данных Python;
- `pandas` – высокопроизводительная библиотека для обработки данных вне базы данных с простыми в использовании структурами данных и инструментами анализа данных;
- `Spark MLlib` – библиотека машинного обучения Spark.

Мы используем MLlib в качестве примера библиотеки машинного обучения. Подход, показанный в этой главе, можно использовать в сочетании с другими библиотеками ML, такими как scikit-learn.

Весь показанный код будет запущен внутри pyspark REPL. Мы можем запустить REPL, выполнив следующую команду:

```
export SPARK_VERSION="spark-2.4.0-bin-hadoop2.7"
./${SPARK_VERSION}/bin/pyspark \
  --driver-memory 2g \
  --executor-memory 6g \
  --packages julioasotodv:spark-tree-plotting:0.2
```

Это похоже на команду, которую мы использовали для запуска REPL в главе 3, но вместо GraphFrames мы загружаем пакет spark-tree-plotting. На момент написания книги последней выпущенной версией Spark была spark-2.4.0-bin-hadoop2.7, но, поскольку она наверняка изменилась к моменту прочтения, обязательно исправьте значение переменной среды `SPARK_VERSION` соответствующим образом.

После запуска кода мы первым делом импортируем нужные библиотеки:

```
from py2neo import Graph
import pandas as pd
from numpy.random import randint

from pyspark.ml import Pipeline
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.feature import StringIndexer, VectorAssembler
from pyspark.ml.evaluation import BinaryClassificationEvaluator

from pyspark.sql.types import *
from pyspark.sql import functions as F

from sklearn.metrics import roc_curve, auc
from collections import Counter

from cycler import cycler
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
```

А теперь давайте создадим соединение с нашей базой данных Neo4j:

```
graph = Graph("bolt://localhost:7687", auth=("neo4j", "neo"))
```

Импорт данных в Neo4j

Теперь мы готовы загрузить данные в Neo4j и создать сбалансированные наборы для обучения и тестирования. Вам нужно скачать ZIP-файл

версии 10 набора данных по адресу https://www.aminer.cn/billboard/citation, распаковать его и поместить содержимое в папку для *импорта*. У вас должны быть наготове следующие файлы:

- *dblp-ref-0.json*;
- *dblp-ref-1.json*;
- *dblp-ref-2.json*;
- *dblp-ref-3.json*.

Как только мы поместим эти файлы в папку импорта, нужно добавить следующее свойство в файл настроек Neo4j, чтобы мы могли обработать их с помощью библиотеки APOC:

```
apoc.import.file.enabled=true
apoc.import.file.use_neo4j_config=true
```

Сначала мы создадим ограничения, чтобы гарантировать, что не будут созданы дубликаты статей или авторов:

```
CREATE CONSTRAINT ON (article:Article)
ASSERT article.index IS UNIQUE;

CREATE CONSTRAINT ON (author:Author)
ASSERT author.name IS UNIQUE;
```

Теперь можем выполнить следующий запрос для импорта данных из файлов JSON:

```
CALL apoc.periodic.iterate(
  'UNWIND ["dblp-ref-0.json","dblp-ref-1.json",
           "dblp-ref-2.json","dblp-ref-3.json"] AS file
  CALL apoc.load.json("file:///" + file)
  YIELD value
  WHERE value.venue IN ["Lecture Notes in Computer Science",
                        "Communications of The ACM",
                        "international conference on software engineering",
                        "advances in computing and communications"]
  return value',
 'MERGE (a:Article {index:value.id})
   ON CREATE SET a += apoc.map.clean(value,["id","authors","references"],[0])
   WITH a,value.authors as authors
   UNWIND authors as author
   MERGE (b:Author{name:author})
   MERGE (b)<-[:AUTHOR]-(a)'
, {batchSize: 10000, iterateList: true});
```

В результате мы получим схему данных, показанную на рис. 8.4.

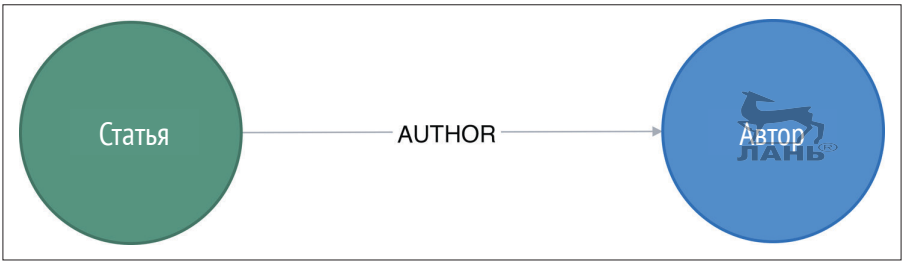


Рис. 8.4. Граф цитирования

Это простой граф, который всего лишь связывает статьи и авторов. Чтобы помочь с предсказаниями, добавим больше информации, которую можно извлечь из связей.

Граф соавторства

Мы хотим предсказать будущее сотрудничество между авторами, поэтому начнем с создания графа соавторства. Следующий запрос к Neo4j на языке Cypher создаст отношение `CO_AUTHOR` (соавтор) между каждой парой авторов, которые сотрудничали при написании статьи:

```
MATCH (a1)<-[:AUTHOR]-(paper)-[:AUTHOR]->(a2:Author)
WITH a1, a2, paper
ORDER BY a1, paper.year
WITH a1, a2, collect(paper)[0].year AS year, count(*) AS collaborations
MERGE (a1)-[coauthor:CO_AUTHOR {year: year}]-(a2)
SET coauthor.collaborations = collaborations;
```

Свойство `year`, которое мы установили для отношения `CO_AUTHOR` в запросе, отражает самый ранний год, когда эти два автора сотрудничали. Мы заинтересованы только в том, чтобы пара авторов сотрудничала в первый раз – последующее сотрудничество не имеет значения.

На рис. 8.5 показан пример части созданного нами графа. Мы уже можем видеть некоторые интересные структуры сообщества.

Каждый кружок на этой диаграмме представляет одного автора, а линии между ними – связи `CO_AUTHOR`, поэтому слева у нас есть четыре автора, которые сотрудничали друг с другом, а справа два примера сотрудничества трех авторов. Теперь, когда у нас есть загруженные данные и базовый граф, давайте создадим два набора данных, которые понадобятся для обучения и тестирования.

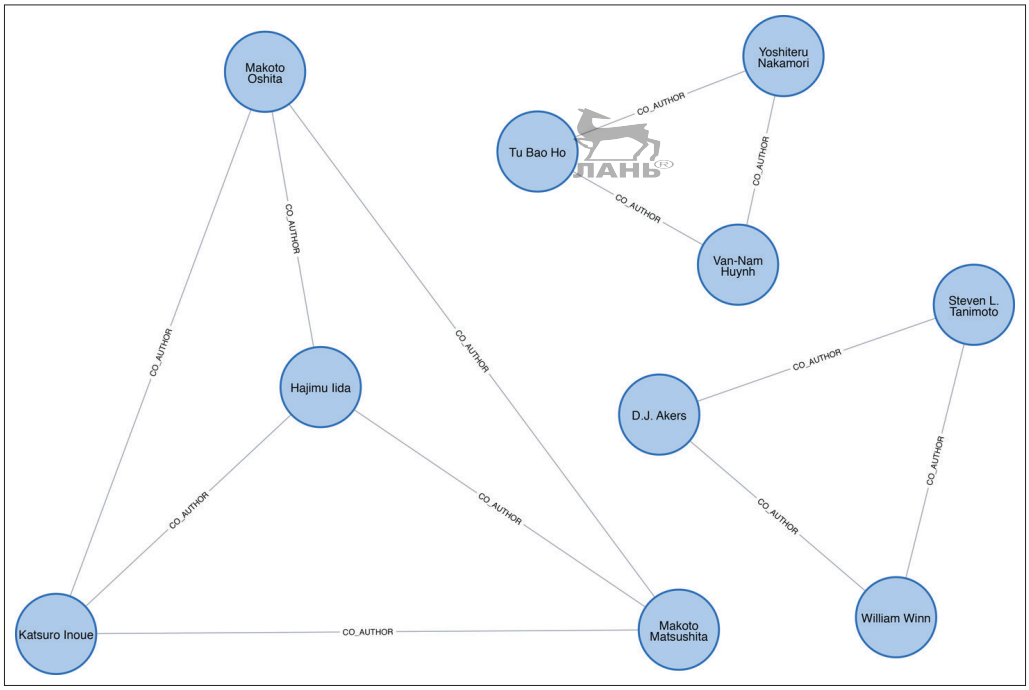


Рис. 8.5. Граф соавторства

Создание сбалансированных наборов данных для обучения и тестирования

Решая проблему предсказания связей, мы пытаемся предугадать возникновение *будущих* связей. Наш набор данных хорошо подходит для этой задачи, потому что мы можем воспользоваться датами публикации статей. Нужно выяснить, какой год будем использовать в качестве границы для разделения данных на выборки для обучения и тестирования. Мы будем обучать нашу модель на всех данных до этого года, а затем тестировать ее на связях, возникших позже.

Давайте начнем с выяснения, когда и сколько статей было опубликовано. Мы можем написать следующий запрос, чтобы получить количество статей, сгруппированных по годам:

```
query = """
MATCH (article:Article)
RETURN article.year AS year, count(*) AS count
ORDER BY year
"""
by_year = graph.run(query).to_data_frame()
```

Давайте представим результат в виде гистограммы при помощи следующего кода:

```
plt.style.use('fivethirtyeight')
ax = by_year.plot(kind='bar', x='year', y='count', legend=None, figsize=(15,8))
ax.xaxis.set_label_text("")
plt.tight_layout()
plt.show()
```

Диаграмма, сгенерированная этим кодом, показана на рис. 8.6.

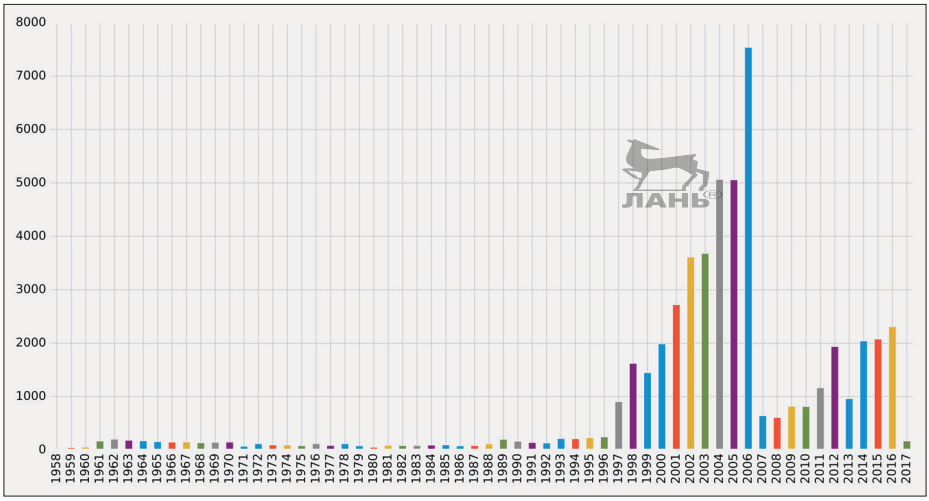


Рис. 8.6. Распределение количества статей по годам

До 1997 года было опубликовано очень мало статей, затем в период с 2001 по 2006 год наблюдался интенсивный рост, завершившийся резким падением, и далее наблюдается плавный рост с 2011 года (исключая 2013-й). Похоже, 2006 год мог бы стать хорошей границей для разделения наших данных на наборы для обучения и тестирования. Давайте проверим, сколько работ было опубликовано до этого года, сколько за этот год и сколько после него. Для этого достаточно простого запроса:

```
MATCH (article:Article)
RETURN article.year < 2006 AS training, count(*) AS count
```

Результат показан ниже, где `true` (истина) означает, что статья была опубликована до 2006 года:

training	count
false	21059
true	30897

Неплохо! 60% статей были опубликованы до 2006 года, и 40% – в течение или после него. Это довольно сбалансированный раздел данных для нашего обучения и тестирования.

Так что теперь, когда у нас есть хороший критерий разделения статей, давайте использовать его и для соавторства. Мы создадим связь CO_AUTHOR_EARLY между парами авторов, чье первое сотрудничество состоялось до 2006 года:

```
MATCH (a1)<-[:AUTHOR]-(paper)-[:AUTHOR]->(a2:Author)
WITH a1, a2, paper
ORDER BY a1, paper.year
WITH a1, a2, collect(paper)[0].year AS year, count(*) AS collaborations
WHERE year < 2006
MERGE (a1)-[coauthor:CO_AUTHOR_EARLY {year: year}]-(a2)
SET coauthor.collaborations = collaborations;
```

А затем создадим связь CO_AUTHOR_LATE между парами авторов, чье первое сотрудничество состоялось в течение или после 2006 года:

```
MATCH (a1)<-[:AUTHOR]-(paper)-[:AUTHOR]->(a2:Author)
WITH a1, a2, paper
ORDER BY a1, paper.year
WITH a1, a2, collect(paper)[0].year AS year, count(*) AS collaborations
WHERE year >= 2006
MERGE (a1)-[coauthor:CO_AUTHOR_LATE {year: year}]-(a2)
SET coauthor.collaborations = collaborations;
```

Прежде чем мы создадим наши обучающие и тестовые наборы, давайте проверим, сколько у нас пар вершин, которые связаны между собой. Следующий запрос найдет количество пар CO_AUTHOR_EARLY:

```
MATCH ()-[:CO_AUTHOR_EARLY]->()
RETURN count(*) AS count
```

Выполнение запроса вернет следующий результат:

count
81096

А этот запрос найдет количество пар CO_AUTHOR_LATE:

```
MATCH ()-[:CO_AUTHOR_LATE]->()
RETURN count(*) AS count
```

Выполнение запроса вернет следующий результат:

count
74128

Теперь мы готовы к созданию наших обучающих и тестовых наборов данных.

Балансировка и разделение данных

Пары вершин со связями `CO_AUTHOR_EARLY` и `CO_AUTHOR_LATE` между ними будут служить нашими положительными примерами, но для обучения нам также нужны отрицательные примеры. Большинство реальных сетей являются разреженными, с локальными сгустками связей, и наш граф не является исключением – количество пар, не имеющих связи, в нем намного больше, чем пар, у которых есть связь.

Если мы запросим количество связей `CO_AUTHOR_EARLY`, то обнаружим, что существует 45 018 авторов с таким типом связей, но между авторами установлено только 81 096 связей. Наши данные могут показаться сбалансированными, но на самом деле это далеко не так: потенциальное максимальное количество связей, которое может иметь наш граф, составляет (45 018 * 45 017) / 2 = 1 013 287 653, что означает наличие намного большего количества отрицательных примеров, т. е. пар без связи. Если мы используем для обучения нашей модели все отрицательные примеры, возникнет серьезная проблема дисбаланса класса. С формальной точки зрения модель может достичь чрезвычайно высокой точности, просто заявляя, что каждая пара вершин не имеет связи.

В своей работе[8] «Новые перспективы и методы в прогнозировании связей» Р. Лихтенвальтер, Дж. Люсье и Н. Чаула описывают несколько методов решения этой проблемы. Одним из таких подходов является создание отрицательных примеров путем поиска вершин в окрестном районе, с которыми мы в данный момент не связаны.

Мы сформируем отрицательные примеры, найдя пары узлов, которые представляют собой смесь между двумя и тремя переходами друг от друга, за исключением тех пар, которые уже имеют связи. Затем уменьшим выборку этих пар узлов, чтобы у нас было одинаковое количество положительных и отрицательных примеров.

У нас есть 314 248 пар вершин, которые не связаны между собой на расстоянии двух переходов. Если мы увеличим расстояние до трех переходов, у нас будет 967 677 пар вершин.

[8] *R. Lichtenwalter, J. Lussier, and N. Chawla*, «New Perspectives and Methods in Link Prediction», https://ntrda.me/2TrSg9K.

Для уменьшения количества отрицательных примеров напишем такую функцию:

```
def down_sample(df):
    copy = df.copy()
    zero = Counter(copy.label.values)[0]
    un = Counter(copy.label.values)[1]
    n = zero - un
    copy = copy.drop(copy[copy.label == 0].sample(n=n, random_state=1).index)
    return copy.sample(frac=1)
```


Эта функция определяет разницу между количеством положительных и отрицательных примеров, а затем делает выборку отрицательных примеров, чтобы получить равные числа. Затем мы воспользуемся следующим кодом для построения обучающего набора со сбалансированными положительными и отрицательными примерами:

```
train_existing_links = graph.run("""
MATCH (author:Author)-[:CO_AUTHOR_EARLY]->(other:Author)
RETURN id(author) AS node1, id(other) AS node2, 1 AS label
""").to_data_frame()

train_missing_links = graph.run("""
MATCH (author:Author)
WHERE (author)-[:CO_AUTHOR_EARLY]-()
MATCH (author)-[:CO_AUTHOR_EARLY*2..3]-(other)
WHERE not((author)-[:CO_AUTHOR_EARLY]-(other))
RETURN id(author) AS node1, id(other) AS node2, 0 AS label
""").to_data_frame()

train_missing_links = train_missing_links.drop_duplicates()
training_df = train_missing_links.append(train_existing_links, ignore_index=True)
training_df['label'] = training_df['label'].astype('category')
training_df = down_sample(training_df)
training_data = spark.createDataFrame(training_df)
```

Теперь у нас есть категориальный столбец `label`, где 1 указывает на наличие связи между парой вершин, а 0 указывает на отсутствие связи. Мы можем посмотреть на данные в нашем блоке DataFrame, запустив следующий код:

```
training_data.show(n=5)
```

node1	node2	label
10019	28091	1
10170	51476	1
10259	17140	0
10259	26047	1
10293	71349	1

Результаты показывают нам список пар вершин и наличие соавторства. Например, пара вершин 10019 и 28091 имеет метку 1, означающую совместную работу.

Теперь давайте выполним следующий код, чтобы проверить сводку содержимого DataFrame:

```
training_data.groupby("label").count().show()
```

Так выглядит результат:

label	count
0	81096
1	81096

Замечательно – мы создали обучающий набор с одинаковым количеством положительных и отрицательных образцов. Теперь нужно сделать то же самое для тестового набора. Следующий код создаст тестовый набор со сбалансированными положительными и отрицательными примерами:

```
test_existing_links = graph.run("""
MATCH (author:Author)-[:CO_AUTHOR_LATE]->(other:Author)
RETURN id(author) AS node1, id(other) AS node2, 1 AS label
""").to_data_frame()

test_missing_links = graph.run("""
MATCH (author:Author)
WHERE (author)-[:CO_AUTHOR_LATE]-()
MATCH (author)-[:CO_AUTHOR*2..3]-(other)
WHERE not((author)-[:CO_AUTHOR]-(other))
RETURN id(author) AS node1, id(other) AS node2, 0 AS label
""").to_data_frame()

test_missing_links = test_missing_links.drop_duplicates()
test_df = test_missing_links.append(test_existing_links, ignore_index=True)
test_df['label'] = test_df['label'].astype('category')
test_df = down_sample(test_df)
test_data = spark.createDataFrame(test_df)
```

Проверим содержимое DataFrame:

```
test_data.groupby("label").count().show()
```

и получим следующий результат:

label	count
0	74128
1	74128

Теперь, когда мы сбалансировали наборы данных для обучения и тестирования, давайте обсудим методы предсказания связей.

Как мы предсказываем недостающие связи

Нам нужно начать с предположений о том, какие элементы в данных означают, что два автора могут в будущем стать соавторами. Наша гипотеза будет варьироваться в зависимости от области и проблемы, но в данном случае мы считаем, что наиболее важные прогнозирующие признаки будут связаны с сообществами. Начнем с предположения, что следующие элементы увеличивают вероятность соавторства:

- больше соавторов в целом;
- потенциальные треугольные связи между авторами;
- авторы с большим количеством связей;
- авторы в одном сообществе;
- авторы в одном более локальном сообществе.

Мы будем строить графовые признаки на основе наших предположений и использовать их для обучения бинарного классификатора. *Бинарная классификация* (binary classification) – это разновидность задачи ML, которая заключается в прогнозировании, к какой из двух предопределенных групп принадлежит элемент на основании некого правила классификации. Мы используем классификатор для прогнозирования, будет ли пара авторов иметь связь или нет. В нашем примере значение 1 говорит о том, что связь есть (соавторство), а значение 0 – что связи нет (авторы не сотрудничают).

Мы реализуем наш бинарный классификатор как случайный лес на платформе Spark. *Случайный лес* (random forest) – это метод композиционного обучения для классификации, регрессии и других задач, как показано на рис. 8.7.

Наш классификатор случайного леса будет извлекать результаты из множества деревьев решений, которые мы обучаем, и затем использовать голосование для прогнозирования классификации – в нашем примере это прогноз наличия соавторства.

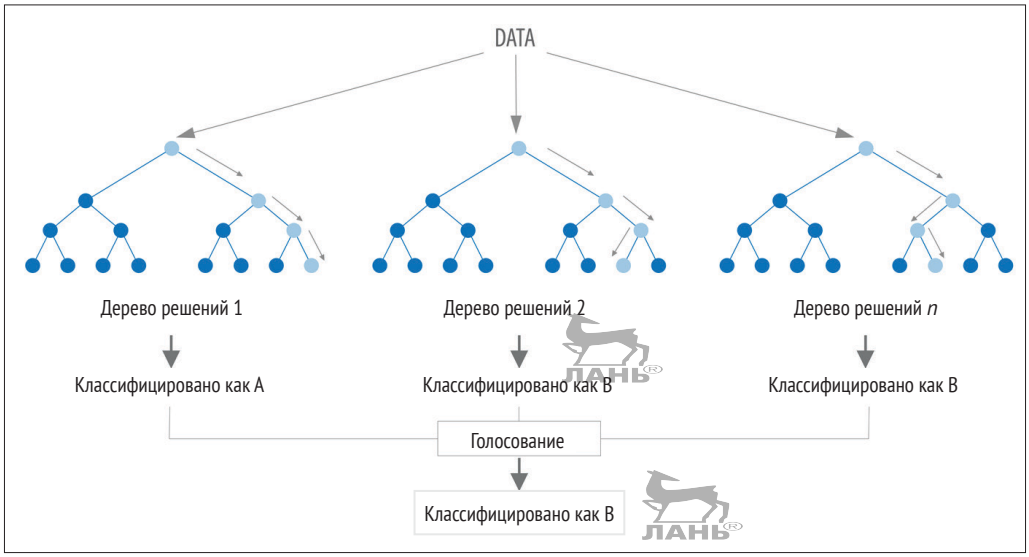


Рис. 8.7. Случайный лес строит коллекцию деревьев решений и затем агрегирует результаты по большинству голосов (классификация) или для получения среднего значения (регрессия)

Теперь давайте сформируем полный цикл машинного обучения.

Разработка полного цикла машинного обучения

Мы создадим наш цикл машинного обучения на основе классификатора случайного леса в Spark. Этот подход вполне уместен, так как наш набор данных будет состоять из сочетания сильных и слабых признаков. Хотя слабые признаки иногда могут быть полезны, метод случайного леса гарантирует, что мы не создадим модель, которая соответствует только обучающим данным.

Чтобы создать цикл ML, мы передадим список признаков как переменную `fields` – это те признаки, которые будет использовать наш классификатор. Классификатор ожидает получить эти признаки в виде одного столбца `features`, поэтому мы используем `VectorAssembler` для преобразования данных в требуемый формат.

Следующий код создает цикл машинного обучения и устанавливает наши параметры с помощью MLlib:

```
def create_pipeline(fields):
    assembler = VectorAssembler(inputCols=fields, outputCol="features")
    rf = RandomForestClassifier(labelCol="label", featuresCol="features",
                                numTrees=30, maxDepth=10)
    return Pipeline(stages=[assembler, rf])
```

`RandomForestClassifier` использует следующие параметры:

- `labelCol` – имя поля, содержащего переменную, которую мы хотим предсказать, т. е. имеет ли пара вершин связь;
- `featuresCol` – имя поля, содержащего переменные, которые будут использоваться для прогнозирования, есть ли у пары вершин связь;
- `numTrees` – количество деревьев решений, которые образуют случайный лес;
- `maxDepth` – максимальная глубина деревьев решений.

В данном случае мы выбрали количество деревьев решений и их глубину на основе ранее проведенных экспериментов. Мы можем считать упомянутые гиперпараметры настройками алгоритма, которые можно подбирать для оптимизации точности. Зачастую бывает сложно определить лучшее сочетание гиперпараметров с первого раза, и настройка модели обычно происходит методом проб и ошибок.

Итак, мы рассмотрели основы и настроили наш алгоритм, поэтому давайте займемся созданием предсказательной модели и оценкой ее точности.

Прогнозирование связей: основные признаки графа

Мы начнем с создания простой модели, которая пытается предсказать, будут ли два автора в будущем сотрудничать, на основе признаков, извлеченных из общих авторов, предпочтительного присоединения и совокупного соседства:

- *общие авторы* (common authors) – находит количество потенциальных треугольников, в которых задействованы два автора. Этот подход отражает идею о том, что два автора, имеющих общих соавторов, могут познакомиться и начать сотрудничать в будущем;
- *предпочтительное присоединение* (preferential attachment) – вычисляет оценку для каждой пары авторов путем умножения количества соавторов, которые есть у каждого из них. Идея заключается в том, что авторы чаще сотрудничают с кем-то, кто уже является соавтором многих работ;
- *совокупное соседство* (total union of neighbors) – находит общее число соавторов, которые есть у каждого автора, за вычетом дубликатов.

В Neo4j мы можем вычислить эти значения, используя запросы на языке Cypher. Следующая функция вычислит упомянутые метрики для учебного набора:

```
def apply_graphy_training_features(data):
    query = """
```

```
    UNWIND $pairs AS pair
    MATCH (p1) WHERE id(p1) = pair.node1
    MATCH (p2) WHERE id(p2) = pair.node2
    RETURN pair.node1 AS node1,
           pair.node2 AS node2,
           size([(p1)-[:CO_AUTHOR_EARLY]-(a)-
                     [:CO_AUTHOR_EARLY]-(p2) | a]) AS commonAuthors,
           size((p1)-[:CO_AUTHOR_EARLY]-()) * size((p2)-
                    [:CO_AUTHOR_EARLY]-()) AS prefAttachment,
           size(apoc.coll.toSet(
             [(p1)-[:CO_AUTHOR_EARLY]-(a) | id(a)] +
                   [(p2)-[:CO_AUTHOR_EARLY]-(a) | id(a)]
           )) AS totalNeighbors
    """
    pairs = [{"node1": row["node1"], "node2": row["node2"]}
                                  for row in data.collect()]
    features = spark.createDataFrame(graph.run(query,
                                  {"pairs": pairs}).to_data_frame())
    return data.join(features, ["node1", "node2"])
```

А эта функция вычислит те же значения для тестового набора:

```
def apply_graphy_test_features(data):
    query = """
    UNWIND $pairs AS pair
    MATCH (p1) WHERE id(p1) = pair.node1
    MATCH (p2) WHERE id(p2) = pair.node2
    RETURN pair.node1 AS node1,
           pair.node2 AS node2,
           size([(p1)-[:CO_AUTHOR]-(a)-[:CO_AUTHOR]-(p2) | a]) AS commonAuthors,
           size((p1)-[:CO_AUTHOR]-()) * size((p2)-[:CO_AUTHOR]-())
                                    AS prefAttachment,
           size(apoc.coll.toSet(
             [(p1)-[:CO_AUTHOR]-(a) | id(a)] + [(p2)-[:CO_AUTHOR]-(a) | id(a)]
           )) AS totalNeighbors
    """
    pairs = [{"node1": row["node1"], "node2": row["node2"]}
                        for row in data.collect()]
    features = spark.createDataFrame(graph.run(query,
                        {"pairs": pairs}).to_data_frame())
    return data.join(features, ["node1", "node2"])
```

Обе эти функции принимают DataFrame, который содержит пары вершин в столбцах `node1` и `node2`. Затем мы строим массив карт, содержащих эти пары, и вычисляем каждую из мер для каждой пары вершин.

Оператор UNWIND особенно полезен в этой главе для получения большого набора пар вершин и возврата всех их признаков в одном запросе.

Мы можем применить эти функции к обучающему и тестовому блоку DataFrames с помощью следующих строк кода:

```
training_data = apply_graphy_training_features(training_data)
test_data = apply_graphy_test_features(test_data)
```

Давайте исследуем данные в нашем обучающем наборе. Следующий код построит гистограмму частотности CommonAuthors:

```
plt.style.use('fivethirtyeight')
fig, axs = plt.subplots(1, 2, figsize=(18, 7), sharey=True)
charts = [(1, "have collaborated"), (0, "haven't collaborated")]

for index, chart in enumerate(charts):
    label, title = chart
    filtered = training_data.filter(training_data["label"] == label)
    common_authors = filtered.toPandas()["commonAuthors"]
    histogram = common_authors.value_counts().sort_index()
    histogram /= float(histogram.sum())
    histogram.plot(kind="bar", x='Common Authors', color="darkblue",
    ax=axs[index], title=f"Authors who {title} (label={label})")
    axs[index].xaxis.set_label_text("Common Authors")

plt.tight_layout()
plt.show()
```

Сгенерированные диаграммы показаны на рис. 8.8.

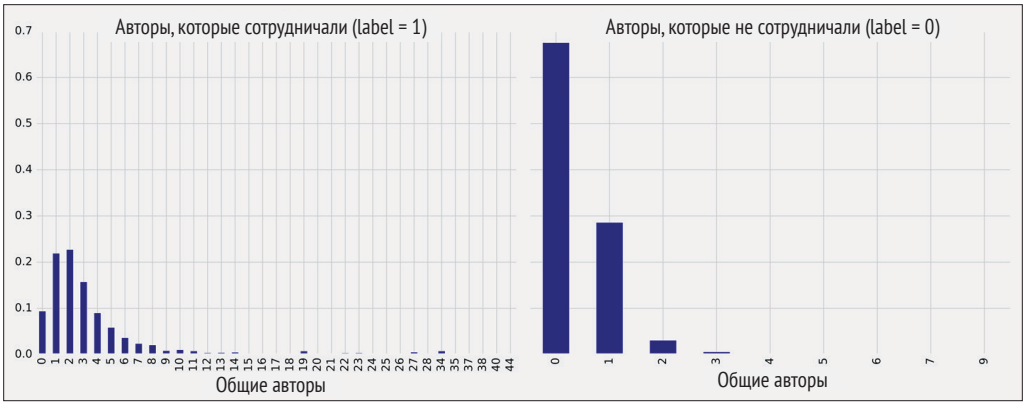


Рис. 8.8. Частотность параметра commonAuthors

Слева мы видим частотность commonAuthors, когда авторы сотрудничают, а справа – частотность commonAuthors, когда они не сотрудничают. Для тех, кто не сотрудничал (справа), максимальное число общих авторов равно 9, но 95% значений равны 1 или 0. Неудивительно, что люди, которые не сотрудничали непосредственно друг с другом в написании статей, в большинстве своем также не имеют других общих соавторов. Для тех, кто сотрудничал (левая сторона), 70% имеют менее пяти общих соавторов, однако наблюдается резкий скачок количества обладателей одного или двух соавторов.

Теперь мы хотим обучить модель прогнозировать отсутствующие связи. Этим занимается следующая функция:

```
def train_model(fields, training_data):
    pipeline = create_pipeline(fields)
    model = pipeline.fit(training_data)
    return model
```

Начнем с создания базовой модели, которая использует только признак commonAuthors. Мы можем создать эту модель, выполнив всего одну строку кода:

```
basic_model = train_model(["commonAuthors"], training_data)
```

Обучив нашу модель, давайте проверим, как она работает с некоторыми фиктивными данными. Следующий код выполняет оценку при разных значениях commonAuthors:

```
eval_df = spark.createDataFrame(
    [(0,), (1,), (2,), (10,), (100,)],
    ['commonAuthors'])

(basic_model.transform(eval_df)
 .select("commonAuthors", "probability", "prediction")
 .show(truncate=False))
```

Выполнение этого кода даст следующий результат:

commonAuthors	probability	prediction
0	[0.7540494940434322,0.24595050595656787]	0.0
1	[0.7540494940434322,0.24595050595656787]	0.0
2	[0.0536835525078107,0.9463164474921892]	1.0
10	[0.0536835525078107,0.9463164474921892]	1.0

Если значение commonAuthors меньше 2, то существует 75%-ная вероятность того, что между авторами не будет никакой связи, поэтому наша модель прогнозирует 0. Если значение commonAuthors 2 или более, существует

вероятность 94%, что между авторами возникнет связь, поэтому наша модель предсказывает 1.

Давайте теперь оценим нашу модель по тестовому набору. Хотя существует несколько способов оценить, насколько хорошо работает модель, большинство из них основаны на нескольких базовых предсказывающих метриках, как показано в табл. 8.1.

Таблица 8.1. Предсказывающие метрики

Величина	Формула	Описание
Достоверность (accuracy)	$\frac{TruePositives + TrueNegatives}{TotalPredictions}$	Доля прогнозов, которые наша модель получает правильно, или общее количество правильных прогнозов, деленное на общее количество прогнозов. Обратите внимание, что одна только правильность может вводить в заблуждение, особенно когда наши данные несбалансированы. Например, если у нас есть набор данных, содержащий 95 кошек и 5 собак и наша модель предсказывает, что каждое изображение является кошкой, мы получим 95%-ный показатель правильности, несмотря на отсутствие правильной идентификации хотя бы одной собаки
Точность (precision)	$\frac{TruePositives}{TruePositives + FalsePositives}$	Доля положительных ответов, которые являются правильными. Низкая точность оценки указывает на большее количество ложных срабатываний. Модель, которая не дает ложных срабатываний, имеет точность 1,0
Отклик (истинно положительные)	$\frac{TruePositives}{TruePositives + FalseNegatives}$	Доля фактических положительных ответов, которые определены правильно. Низкий отклик указывает на большее количество ложноотрицательных ответов. Модель, которая не дает ложных отрицаний, имеет отклик 1,0
Ложноположительные	$\frac{FalsePositives}{FalsePositives + TrueNegatives}$	Доля ошибочно позитивных заключений. Высокое значение указывает на большое количество ложных срабатываний
Кривая ROC (receiver operation characteristic)	График *X-Y*	Кривая ROC представляет собой график отклика (истинного положительного показателя) по отношению к ложноположительному показателю при различных пороговых значениях классификации. Площадь под кривой AUC (area under curve) соответствует двумерной площади под кривой ROC от координат *X-Y* (0,0) до (1,1)

Для оценки наших моделей мы будем использовать кривые достоверности, точности, отзыва и ROC. Достоверность – это грубая мера, поэтому мы сосредоточимся на повышении нашей общей точности и отклика. Мы будем использовать кривые ROC для сравнения того, как отдельные признаки изменяют прогнозные показатели.

В зависимости от наших целей можно решить использовать разные меры. Например, мы можем стараться исключить все ложноотрицательные показатели обнаружения болезни, но не хотим подталкивать предсказания к полностью положительному результату. Можно установить несколько порогов для разных моделей, которые передают свои результаты для вторичной проверки на вероятность ложных результатов.

Снижение порогов классификации приводит к более обширным положительным результатам, таким образом увеличивая число как ложно-, так и истинно положительных срабатываний.

Для вычисления прогнозных показателей будем использовать следующую функцию:

```
def evaluate_model(model, test_data):
    # Запускаем модель на тестовом наборе.
    predictions = model.transform(test_data)

    # Вычисляем количество истинно положительных, ложноположительных
    # и ложноотрицательных прогнозов.
    tp = predictions[(predictions.label == 1) &
                     (predictions.prediction == 1)].count()
    fp = predictions[(predictions.label == 0) &
                     (predictions.prediction == 1)].count()
    fn = predictions[(predictions.label == 1) &
                     (predictions.prediction == 0)].count()

    # Вычисляем отклик и точность
    recall = float(tp) / (tp + fn)
    precision = float(tp) / (tp + fp)

    # Вычисляем достоверность (accuracy), используя инструмент оценки
    # бинарной классификации библиотеки MLLib
    accuracy = BinaryClassificationEvaluator().evaluate(predictions)

    # Вычисляем коэффициенты ложноположительных и истинно положительных
    # прогнозов при помощи функции sklearn.
```

```
    labels = [row["label"] for row in predictions.select("label").collect()]
    preds = [row["probability"][1] for row in predictions.select
                ("probability").collect()]
    fpr, tpr, threshold = roc_curve(labels, preds)
    roc_auc = auc(fpr, tpr)

    return { "fpr": fpr, "tpr": tpr, "roc_auc": roc_auc, "accuracy": accuracy,
             "recall": recall, "precision": precision }
```

Затем напишем функцию для отображения результатов в более удобном для использования формате:

```
def display_results(results):
    results = {k: v for k, v in results.items() if k not in
                         ["fpr", "tpr", "roc_auc"]}
    return pd.DataFrame({"Measure": list(results.keys()),
                         "Score": list(results.values())})
```

Мы можем вызвать эту функцию и отобразить результаты при помощи двух строк кода:

```
basic_results = evaluate_model(basic_model, test_data)
display_results(basic_results)
```

У нас получились следующие предсказательные характеристики для модели поиска соавторов:

measure	score
accuracy	0.864457
recall	0.753278
precision	0.968670

Это неплохое начало, учитывая, что мы прогнозируем будущее сотрудничество, основываясь только на количестве общих авторов для каждой пары. Тем не менее мы видим более полную картину, если рассматриваем эти характеристики в контексте друг друга. Например, эта модель имеет точность 0,968670, что означает, что она очень хорошо предсказывает *существование* связей. Тем не менее отклик 0,753278 означает, что модель не очень хорошо предсказывает, когда связи *не существуют*.

Мы также можем построить кривую ROC (соотношение истинных положительных и ложноположительных результатов), используя следующие функции:

```
def create_roc_plot():
    plt.style.use('classic')
    fig = plt.figure(figsize=(13, 8))
```

```
    plt.xlim([0, 1])
    plt.ylim([0, 1])
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')
    plt.rc('axes', prop_cycle=(cycler('color',
                   ['r', 'g', 'b', 'c', 'm', 'y', 'k'])))
    plt.plot([0, 1], [0, 1], linestyle='--', label='Random score
                   (AUC = 0.50)')
    return plt, fig

def add_curve(plt, title, fpr, tpr, roc):
    plt.plot(fpr, tpr, label=f"{title} (AUC = {roc:0.2})")
```

Эти функции можно вызвать, например, так:

```
plt, fig = create_roc_plot()

add_curve(plt, "Common Authors",
          basic_results["fpr"], basic_results["tpr"], basic_results["roc_auc"])

plt.legend(loc='lower right')
plt.show()
```

Кривая ROC для нашей базовой модели представлена на рис. 8.9. Текущая модель обеспечивает нам значение 0,86 площади под кривой (AUC). И хотя это дает нам одну общую прогностическую меру, нам нужна диаграмма (или другие меры), чтобы оценить, соответствует ли результат нашей цели. На рис. 8.9 мы видим, что, когда мы приближаемся к 80%-му истинному положительному уровню (отклик), наш ложноположительный уровень достигает около 20%. Это может быть проблематично в некоторых сценариях, таких как обнаружение мошенничества, где ложные срабатывания обходятся дорого.

Теперь давайте воспользуемся другими графовыми признаками, чтобы посмотреть, сможем ли мы улучшить наши прогнозы. И вновь, прежде чем начать обучение модели, давайте изучим распределение данных. Мы можем получить описательную статистику для каждого из наших признаков:

```
(training_data.filter(training_data["label"]==1)
 .describe()
 .select("summary", "commonAuthors", "prefAttachment", "totalNeighbors")
 .show())

(training_data.filter(training_data["label"]==0)
 .describe()
 .select("summary", "commonAuthors", "prefAttachment", "totalNeighbors")
 .show())
```

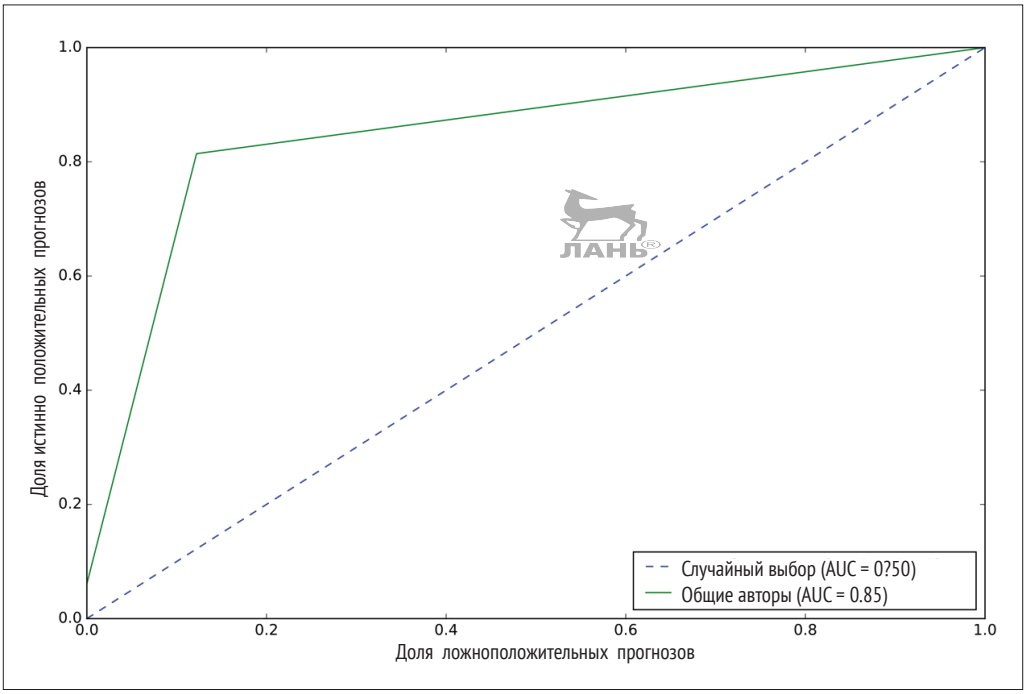


Рис. 8.9. Кривая ROC для базовой модели

Результаты выполнения этих фрагментов кода показаны в следующих таблицах:

summary	commonAuthors	prefAttachment	totalNeighbors
count	81096	81096	81096
mean	3.5959233501035808	69.93537289138798	10.082408503502021
stddev	4.715942231635516	171.47092255919472	8.44109970920685
min	0	1	2
max	44	3150	90

summary	commonAuthors	prefAttachment	totalNeighbors
count	81096	81096	81096
mean	0.37666469369635985	48.18137762651672	12.97586810693499
stddev	0.6194576095461857	94.92635344980489	10.082991078685803
min	0	1	1
max	9	1849	89

Признаки с бóльшими различиями между наличием связи (соавторство) и отсутствием связи (без соавтора) наверняка будут более прогнозирующими, поскольку этот разрыв больше выражен. Среднее значение `prefAttachment` выше для авторов, которые сотрудничали, по сравнению с теми, кто не сотрудничал. Эта разница еще более существенна для обычных авторов. Мы видим, что нет большой разницы в значениях `totalNeighbors`, и, вероятно, этот признак не будет очень прогнозирующим. Кроме того, интересно выглядят большое стандартное отклонение, а также минимальное и максимальное значения для предпочтительного присоединения. Это свойства, которые мы могли бы ожидать от сетей малого мира с выраженными концентраторами (суперконнекторами). Теперь давайте обучим новую модель, добавив новые признаки – предпочтительное присоединение и совокупное соседство, – при помощи следующего кода:

```
fields = ["commonAuthors", "prefAttachment", "totalNeighbors"]
graphy_model = train_model(fields, training_data)
```

А теперь оценим модель и отобразим результаты:

```
graphy_results = evaluate_model(graphy_model, test_data)
display_results(graphy_results)
```

Предсказательные меры для новой графовой модели выглядят так:

measure	score
accuracy	0.978351
recall	0.924226
precision	0.943795

Наша достоверность и отклик значительно возросли, но точность немного снизилась, и мы все еще неправильно классифицируем около 8% ссылок. Давайте построим кривую ROC и сравним нашу базовую и графовую модели, выполнив следующий код:

```
plt, fig = create_roc_plot()

add_curve(plt, "Common Authors",
          basic_results["fpr"], basic_results["tpr"],
                                basic_results["roc_auc"])

add_curve(plt, "Graphy",
          graphy_results["fpr"], graphy_results["tpr"],
                                 graphy_results["roc_auc"])

plt.legend(loc='lower right')
plt.show()
```

Полученнный график изображен на рис. 8.10.

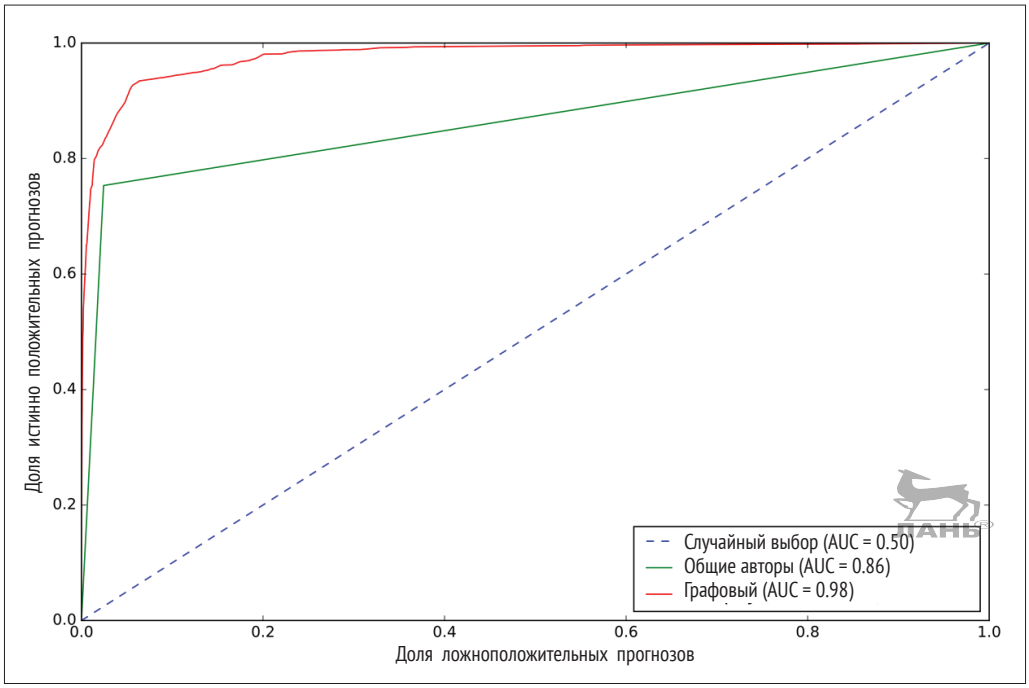


Рис. 8.10. Кривая ROC

В целом похоже, что мы движемся в правильном направлении, и полезно визуализировать сравнение моделей, чтобы понять, как разные признаки влияют на наши результаты.

Теперь, когда у нас есть более одного признака, нужно оценить, какие признаки имеют наибольшее значение. Мы будем использовать меру *важности признака* (feature importance), чтобы измерить влияние различных признаков на прогноз нашей модели. Это позволяет нам оценить влияние, оказываемое на результаты различными алгоритмами и статистикой.

Чтобы вычислить важность признаков, алгоритм случайного леса в Spark усредняет уменьшение засоренности по всем деревьям в лесу. *Засоренность* (impurity) – это частота, с которой случайно назначенные метки оказываются неверны.

Ранжирование признаков по степени важности всегда нормируется к 1. Если мы оцениваем один признак, его важность равна 1,0, поскольку он оказывает 100%-ное влияние на модель.

Следующая функция создает диаграмму, показывающую наиболее влиятельные признаки:

```
def plot_feature_importance(fields, feature_importances):
    df = pd.DataFrame({"Feature": fields, "Importance": feature_importances})
    df = df.sort_values("Importance", ascending=False)
    ax = df.plot(kind='bar', x='Feature', y='Importance', legend=None)
    ax.xaxis.set_label_text("")
    plt.tight_layout()
    plt.show()
```

Теперь вызовем следующую функцию:

```
rf_model = graphy_model.stages[-1]
plot_feature_importance(fields, rf_model.featureImportances)
```

Результат выполнения функции изображен на рис. 8.11.

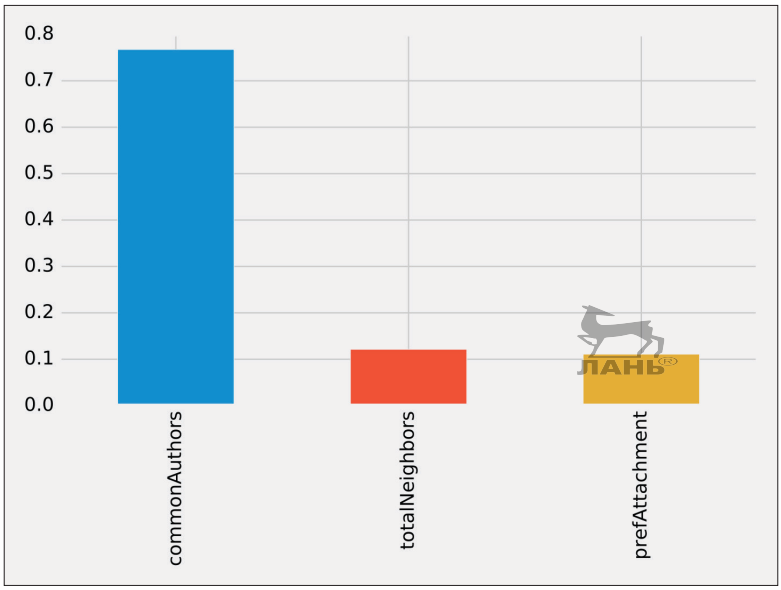


Рис. 8.11. Диаграмма важности признаков

Из трех признаков, которые мы использовали до сих пор, наиболее важным с большим отрывом является признак `commonAuthors`.

Чтобы понять, как создаются наши прогностические модели, мы можем визуализировать одно из деревьев решений в нашем случайном лесу с помощью библиотеки `spark-tree-plotting`. Следующий код генерирует файл GraphViz:

```
from spark_tree_plotting import export_graphviz
dot_string = export_graphviz(rf_model.trees[0],
```

```
    featureNames=fields, categoryNames=[], classNames=["True", "False"],
    filled=True, roundedCorners=True, roundLeaves=True)

with open("/tmp/rf.dot", "w") as file:
    file.write(dot_string)
```

Затем можем сгенерировать визуальное представление этого файла, выполнив следующую команду из терминала:

```
dot -Tpdf /tmp/rf.dot -o /tmp/rf.pdf
```

Результат выполнения этой команды представлен на рис. 8.12.

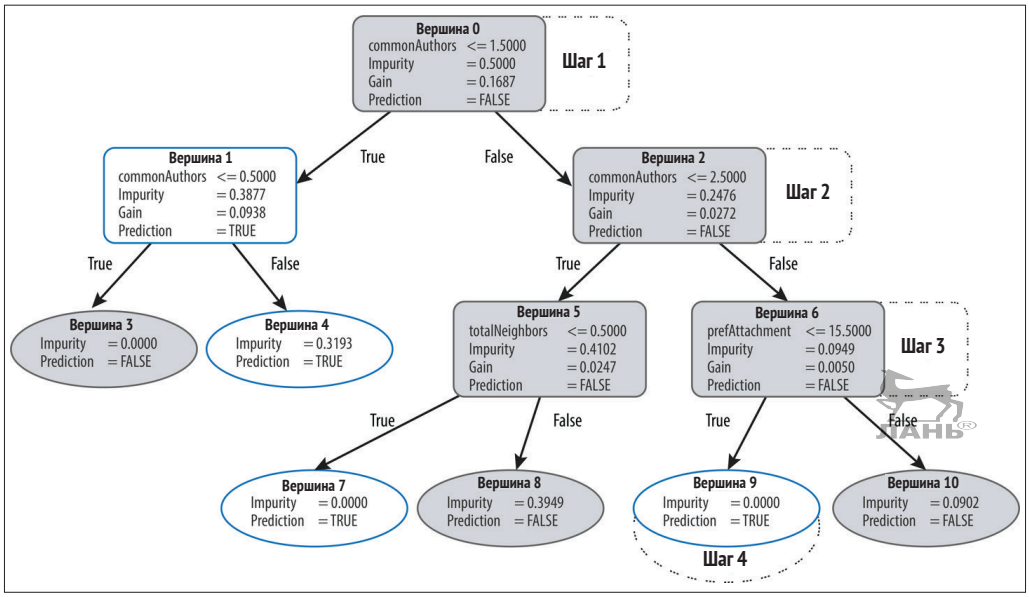


Рис. 8.12. Визуализация дерева решений

Представьте, что мы используем это дерево решений, чтобы предсказать, связана ли пара узлов, исходя из следующих признаков:

commonAuthors	prefAttachment	totalNeighbors
10	12	5

При создании прогноза наш случайный лес проходит через несколько шагов:

1. Начинаем с вершины 0, где у нас не выполняется условие `commonAuthors<=1.5`, поэтому следуем по ветви `False` до вершины 2.
2. Здесь значение `commonAuthors` превышает `2.5`, поэтому мы следуем по ветви `False` до вершины 6.

3. У нас есть оценка `prefAttachment` меньше `15.5`, которая приводит по ветви `True` к вершине 9.
4. Вершина 9 является листовой (конечной) вершиной в этом дереве решений, что означает, что не нужно больше проверять условия, – значение `Prediction` (т. е. `True`) в этом узле является предсказанием дерева решений.
5. Наконец, случайный лес оценивает прогнозируемый элемент по совокупности этих деревьев решений и выдает свой прогноз на основе наиболее популярных результатов.

Теперь давайте обсудим добавление дополнительных признаков графа.

Прогнозирование связей: треугольники и коэффициент кластеризации

Рекомендательные решения часто основаны на какой-либо форме треугольной метрики, поэтому давайте посмотрим, помогут ли они в нашем примере. Мы можем вычислить количество треугольников, частью которых является вершина, и ее коэффициент кластеризации, выполнив следующий запрос:

```
CALL algo.triangleCount('Author', 'CO_AUTHOR_EARLY', { write:true,
  writeProperty:'trianglesTrain', clusteringCoefficientProperty:
                'coefficientTrain'});

CALL algo.triangleCount('Author', 'CO_AUTHOR', { write:true,
  writeProperty:'trianglesTest', clusteringCoefficientProperty:
                'coefficientTest'});
```

Следующая функция добавит эти признаки в наши блоки данных DataFrames:

```
def apply_triangles_features(data, triangles_prop, coefficient_prop):
    query = """
    UNWIND $pairs AS pair
    MATCH (p1) WHERE id(p1) = pair.node1
    MATCH (p2) WHERE id(p2) = pair.node2
    RETURN pair.node1 AS node1,
           pair.node2 AS node2,
           apoc.coll.min([p1[$trianglesProp], p2[$trianglesProp]])
                                              AS minTriangles,
           apoc.coll.max([p1[$trianglesProp], p2[$trianglesProp]])
                                              AS maxTriangles,
           apoc.coll.min([p1[$coefficientProp], p2[$coefficientProp]])
                                              AS minCoefficient,
```

```
            apoc.coll.max([p1[$coefficientProp], p2[$coefficientProp]])
                                                    AS maxCoefficient
    """
    params = {
        "pairs": [{"node1": row["node1"], "node2": row["node2"]}
                            for row in data.collect()],
        "trianglesProp": triangles_prop,
        "coefficientProp": coefficient_prop
    }

    features = spark.createDataFrame(graph.run(query, params).to_data_frame())
    return data.join(features, ["node1", "node2"])
```

Обратите внимание, что мы использовали префиксы `min` и `max` для наших алгоритмов подсчета треугольников и коэффициента кластеризации. Нужен способ предотвратить обучение нашей модели на основе порядка, в котором авторы в парах передаются из нашего неориентированного графа. Для этого мы разделили эти признаки по авторам с минимальным и максимальным количеством.

Можно применить эту функцию к нашим обучающим и тестовым наборам данных с помощью следующего кода:

```
training_data = apply_triangles_features(training_data,
                                         "trianglesTrain", "coefficientTrain")
test_data = apply_triangles_features(test_data,
                                         "trianglesTest", "coefficientTest")
```

Теперь запустите следующий код, чтобы показать описательную статистику для каждого из наших признаков:

```
(training_data.filter(training_data["label"]==1)
 .describe()
 .select("summary", "minTriangles", "maxTriangles",
                    "minCoefficient", "maxCoefficient")
 .show())

(training_data.filter(training_data["label"]==0)
 .describe()
 .select("summary", "minTriangles", "maxTriangles", "minCoefficient",
                                                    "maxCoefficient")
 .show())
```

Результаты выполнения этих блоков кода показаны в следующих таблицах:

summary	minTriangles	maxTriangles	minCoefficient	maxCoefficient
count	81096	81096	81096	81096
mean	19.478260333431983	27.73590559337082	0.5703773654487051	0.8453786164620439
stddev	65.7615282768483	74.01896188921927	0.3614610553659958	0.2939681857356519
min	0	0	0.0	0.0
max	622	785	1.0	1.0

summary	minTriangles	maxTriangles	minCoefficient	maxCoefficient
count	81096	81096	81096	81096
mean	5.754661142349808	35.651980368945445	0.49048921333297446	0.860283935358397
stddev	20.639236521699	85.82843448272624	0.3684138346533951	0.2578219623967906
min	0	0	0.0	0.0
max	617	785	1.0	1.0

Сравнивая эти таблицы, обратите внимание, что разница между данными о наличии соавторства и его отсутствии не так велика. Это может означать, что использованные признаки не являются прогнозирующими. Мы можем обучить другую модель, запустив следующий код:

```
fields = ["commonAuthors", "prefAttachment", "totalNeighbors",
          "minTriangles", "maxTriangles", "minCoefficient", "maxCoefficient"]
triangle_model = train_model(fields, training_data)
```

А теперь давайте оценим модель и покажем результаты:

```
triangle_results = evaluate_model(triangle_model, test_data)
display_results(triangle_results)
```

Предсказательные меры для модели с треугольниками показаны в таблице ниже:

measure	score
accuracy	0.992924
recall	0.965384
precision	0.958582

Наши прогностические показатели значительно улучшаются с добавлением каждого нового признака в предыдущую модель. Давайте добавим

нашу модель с треугольниками к диаграмме кривой ROC при помощи следующего кода:

```
plt, fig = create_roc_plot()

add_curve(plt, "Common Authors",
          basic_results["fpr"], basic_results["tpr"], basic_results["roc_auc"])

add_curve(plt, "Graphy",
          graphy_results["fpr"], graphy_results["tpr"],
                                 graphy_results["roc_auc"])

add_curve(plt, "Triangles",
          triangle_results["fpr"], triangle_results["tpr"],
                                   triangle_results["roc_auc"])

plt.legend(loc='lower right')
plt.show()
```

Мы получим график, показанный на рис. 8.13.

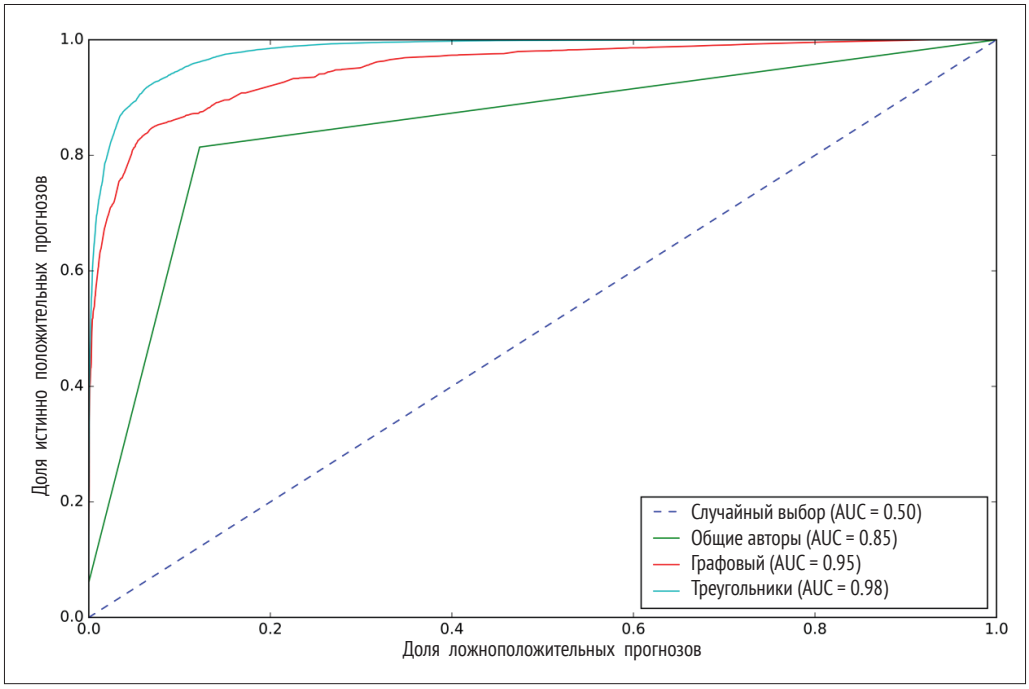


Рис. 8.13. Кривая ROC для модели с треугольниками

Наши модели в целом улучшились, и мы находимся в 90%-ной области предсказательных характеристик. Сейчас настал самый трудный этап раз-

работки модели, потому что самые очевидные улучшения достигнуты, но есть все еще место для совершенствования. Давайте посмотрим, как изменились важные признаки:

```
rf_model = triangle_model.stages[-1]
plot_feature_importance(fields, rf_model.featureImportances)
```

Результаты выполнения этого кода можно увидеть на рис. 8.14. Признак `common authors` по-прежнему оказывает наибольшее влияние на нашу модель. Возможно, нам нужно обратить внимание на новые области и посмотреть, что произойдет, когда мы добавим информацию о наличии сообществ.

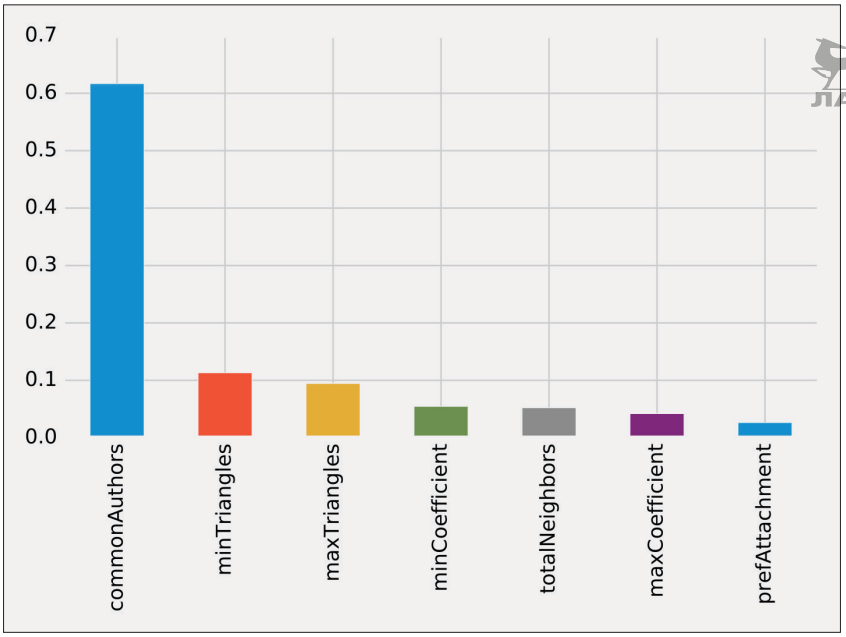


Рис. 8.14. Значимость признака – модель с треугольниками

Прогнозирование связей: выделение сообществ

Мы предполагаем, что вершины, которые находятся в одном сообществе, с большей вероятностью установят связь между собой, если они этого еще не сделали. Более того, мы считаем, что чем теснее сообщество, тем более вероятны связи.

Мы начнем с расчета более крупных сообществ, используя алгоритм распространения меток в Neo4j. С этой целью выполним следующий запрос, который сохранит сообщество в свойстве `partitionTrain` для обучающего набора и `partitionTest` для тестового набора:

```
CALL algo.labelPropagation("Author", "CO_AUTHOR_EARLY", "BOTH",
  {partitionProperty: "partitionTrain"});

CALL algo.labelPropagation("Author", "CO_AUTHOR", "BOTH",
  {partitionProperty: "partitionTest"});
```

Мы также вычислим более мелкозернистые группы, используя Лувенский алгоритм. Этот алгоритм возвращает промежуточные кластеры, и мы будем хранить наименьший из этих кластеров в свойстве `louvainTrain` для обучающего набора и `louvainTest` для тестового набора:


```
CALL algo.louvain.stream("Author", "CO_AUTHOR_EARLY",
                         {includeIntermediateCommunities:true})
YIELD nodeId, community, communities
WITH algo.getNodeById(nodeId) AS node, communities[0] AS smallestCommunity
SET node.louvainTrain = smallestCommunity;

CALL algo.louvain.stream("Author", "CO_AUTHOR",
                         {includeIntermediateCommunities:true})
YIELD nodeId, community, communities
WITH algo.getNodeById(nodeId) AS node, communities[0] AS smallestCommunity
SET node.louvainTest = smallestCommunity;
```

Теперь создадим следующую функцию для возврата значений из этих алгоритмов:


```
def apply_community_features(data, partition_prop, louvain_prop):
    query = """
    UNWIND $pairs AS pair
    MATCH (p1) WHERE id(p1) = pair.node1
    MATCH (p2) WHERE id(p2) = pair.node2
    RETURN pair.node1 AS node1,
           pair.node2 AS node2,
           CASE WHEN p1[$partitionProp] = p2[$partitionProp] THEN
                     1 ELSE 0 END AS samePartition,
           CASE WHEN p1[$louvainProp] = p2[$louvainProp] THEN
                     1 ELSE 0 END AS sameLouvain
    """
    params = {
        "pairs": [{"node1": row["node1"], "node2": row["node2"]} for
                           row in data.collect()],
        "partitionProp": partition_prop,
        "louvainProp": louvain_prop
    }
    features = spark.createDataFrame(graph.run(query, params).to_data_frame())
    return data.join(features, ["node1", "node2"])
```

Мы можем применить эту функцию к обучающим и тестовым блокам DataFrames в Spark с помощью кода:

```
training_data = apply_community_features(training_data,
                                         "partitionTrain", "louvainTrain")
test_data = apply_community_features(test_data, "partitionTest", "louvainTest")
```

И наконец, можно запустить следующий код, чтобы увидеть, принадлежат ли пары вершин к одному сообществу:

```
plt.style.use('fivethirtyeight')
fig, axs = plt.subplots(1, 2, figsize=(18, 7), sharey=True)
charts = [(1, "have collaborated"), (0, "haven't collaborated")]

for index, chart in enumerate(charts):
    label, title = chart
    filtered = training_data.filter(training_data["label"] == label)
    values = (filtered.withColumn('samePartition',
              F.when(F.col("samePartition") == 0, "False")
                                  .otherwise("True"))
              .groupby("samePartition")
              .agg(F.count("label").alias("count"))
              .select("samePartition", "count")
              .toPandas())
    values.set_index("samePartition", drop=True, inplace=True)
    values.plot(kind="bar", ax=axs[index], legend=None,
                title=f"Authors who {title} (label={label})")
    axs[index].xaxis.set_label_text("Same Partition")
plt.tight_layout()
plt.show()
```

Результат выполнения кода показан на рис. 8.15.

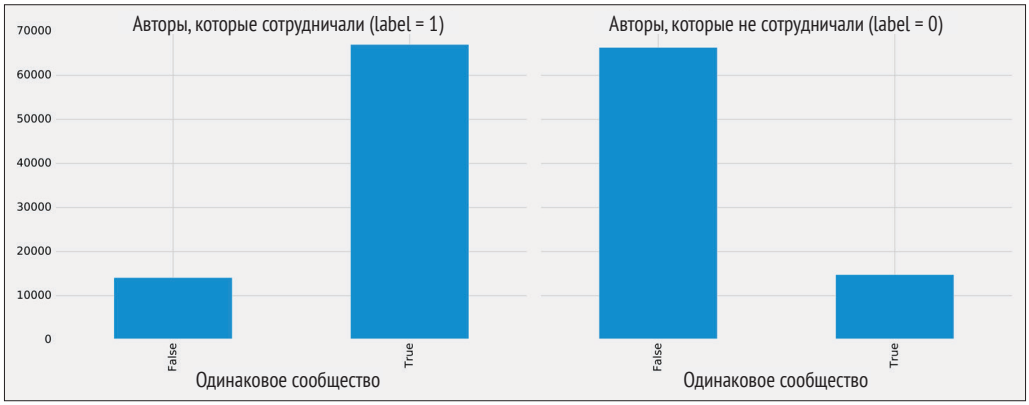


Рис. 8.15. Одинаковые сообщества

Похоже, что этот признак может быть довольно прогнозирующим – авторы, которые сотрудничали, с большей вероятностью окажутся в одном разделе, чем те, кто этого не сделал. Мы можем проделать то же самое для кластеров Лувенского алгоритма, запустив следующий код:

```
plt.style.use('fivethirtyeight')
fig, axs = plt.subplots(1, 2, figsize=(18, 7), sharey=True)
charts = [(1, "have collaborated"), (0, "haven't collaborated")]

for index, chart in enumerate(charts):
    label, title = chart
    filtered = training_data.filter(training_data["label"] == label)
    values = (filtered.withColumn('sameLouvain',
              F.when(F.col("sameLouvain") == 0, "False")
                               .otherwise("True"))
              .groupby("sameLouvain")
              .agg(F.count("label").alias("count"))
              .select("sameLouvain", "count")
              .toPandas())
    values.set_index("sameLouvain", drop=True, inplace=True)
    values.plot(kind="bar", ax=axs[index], legend=None,
                title=f"Authors who {title} (label={label})")
    axs[index].xaxis.set_label_text("Same Louvain")
    plt.tight_layout()
    plt.show()
```

Результат выполнения кода показан на рис. 8.16.

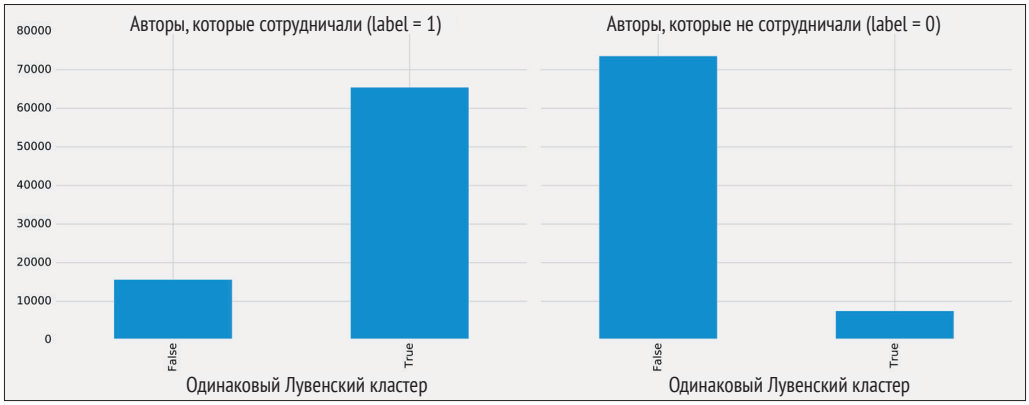


Рис. 8.16. Кластеры, полученные с помощью Лувенского алгоритма

Похоже, что этот признак также может быть достаточно прогнозирующим – авторы, которые сотрудничали, скорее всего, будут находиться в одном кластере.

Мы можем обучить другую модель, запустив следующий код:

```
fields = ["commonAuthors", "prefAttachment", "totalNeighbors",
          "minTriangles", "maxTriangles", "minCoefficient", "maxCoefficient",
          "samePartition", "sameLouvain"]
community_model = train_model(fields, training_data)
```

А теперь давайте оценим модель и отобразим результаты:

```
community_results = evaluate_model(community_model, test_data)
display_results(community_results)
```

Предсказательные меры для модели с сообществами:

measure	score
accuracy	0.995771
recall	0.957088
precision	0.978674

Некоторые из наших показателей улучшились, поэтому для визуального сравнения построим кривую ROC для всех наших моделей, выполнив следующий код:

```
plt, fig = create_roc_plot()

add_curve(plt, "Common Authors",
          basic_results["fpr"], basic_results["tpr"], basic_results["roc_auc"])

add_curve(plt, "Graphy",
          graphy_results["fpr"], graphy_results["tpr"],
          graphy_results["roc_auc"])

add_curve(plt, "Triangles",
          triangle_results["fpr"], triangle_results["tpr"],
          triangle_results["roc_auc"])

add_curve(plt, "Community",
          community_results["fpr"], community_results["tpr"],
          community_results["roc_auc"])

plt.legend(loc='lower right')
plt.show()
```

Результат выполнения кода показан на рис. 8.17.

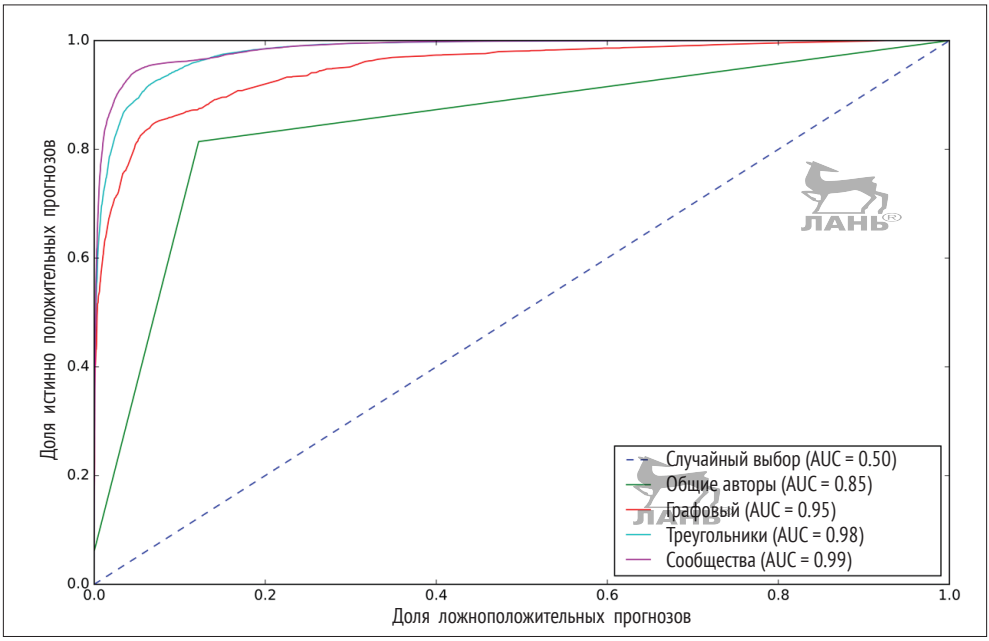


Рис. 8.17. Кривая ROC с учетом сообществ

После добавления признака сообщества наблюдается достаточно заметное улучшение, поэтому давайте посмотрим, какие из признаков являются наиболее важными сейчас:

```
rf_model = community_model.stages[-1]
plot_feature_importance(fields, rf_model.featureImportances)
```

Результаты выполнения этой функции можно увидеть на рис. 8.18.

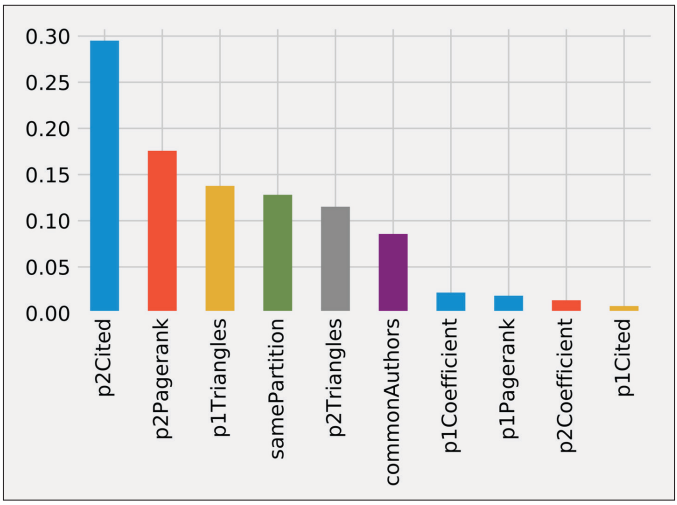


Рис. 8.18. Значимость признаков после добавления модели с сообществами

Хотя модель общих авторов в целом очень важна, желательно избегать чрезмерно доминирующего элемента, который может исказить прогнозы на новых данных. Алгоритмы выделения сообществ оказали большое влияние на нашу последнюю модель со всеми включенными признаками, и это помогает завершить наш прогнозный проект.

Мы видели в примерах, что простые основанные на графе признаки – хорошее начало, и затем, по мере добавления графовых и графово-алгоритмических признаков, постепенно возрастают предсказательные способности нашей модели. Теперь у нас есть хорошая сбалансированная модель для прогнозирования соавторских связей.

Использование графов для извлечения связанных признаков может значительно улучшить наши прогнозы. Наилучшие графовые признаки и алгоритмы различаются в зависимости от атрибутов данных, включая предметную область сети и форму графа. Мы рекомендуем сначала рассмотреть элементы прогнозирования в ваших данных и проверить гипотезы с различными типами связанных признаков, прежде чем выполнять тонкую настройку.

Задания для читателя

Существуют разные области для исследования и способы построения других моделей. Вот несколько идей для дальнейшего изучения:

- насколько хорошо работает наша модель на данных конференций, которые мы не использовали?
- что происходит, когда мы удаляем некоторые признаки при тестировании на новых данных?
- влияет ли на прогнозы разделение по годам между обучением и тестированием?
- текущий набор данных также содержит связи между публикациями. Можем ли мы использовать эти данные для создания различных признаков или прогнозирования будущих связей?

Заключение

В этой главе мы рассмотрели использование графовых признаков и алгоритмов для улучшения моделей машинного обучения. Мы изучили несколько базовых понятий, а затем разобрали подробный пример интеграции Neo4j и Apache Spark для предсказания связей. Мы продемонстрировали, как оценивать модели классификатора на основе случайного леса и добавлять различные типы связанных признаков для улучшения модели.

Итог книги

В этой книге мы освоили основы теории графов, а также платформы обработки и анализа; рассмотрели множество практических примеров использования графовых алгоритмов в Apache Spark и Neo4j. Мы закончили книгу изучением того, как графы улучшают машинное обучение.

Графовые алгоритмы являются движущей силой анализа реальных систем – от предотвращения мошенничества и оптимальной сетевой маршрутизации до прогнозирования распространения гриппа. Мы надеемся, что вы присоединитесь к нам и разработаете свои собственные уникальные решения, использующие преимущества современных высокосвязанных данных.

Приложение A

Дополнительная информация и ресурсы

В этом приложении мы кратко изложим дополнительные сведения, которые могут пригодиться читателям. Будут рассмотрены другие типы алгоритмов, иной способ импорта данных в Neo4j и другая библиотека процедур. Здесь также упомянуты некоторые ресурсы для поиска наборов данных и помощи в обучении и освоении графовых платформ.

Дополнительные алгоритмы

Для работы с графовыми данными придумано много разных алгоритмов. В этой книге мы сосредоточились на алгоритмах, которые предназначены для классических графов и наиболее полезны для разработчиков приложений. Некоторые алгоритмы, такие как раскраска областей и эвристика, были опущены, поскольку они либо представляют чисто академический интерес, либо могут быть легко реализованы при помощи других алгоритмов.

Некоторые алгоритмы, такие как выделение сообществ на основе границ, интересны, но еще не реализованы в Neo4j или Apache Spark. Мы ожидаем, что список графовых алгоритмов, используемых в обеих платформах, будет увеличиваться по мере роста популярности анализа графов.

Есть также категории алгоритмов, которые используются с графами, но не являются строго графовыми по своей природе. Например, в главе 8 мы рассмотрели несколько алгоритмов, используемых в контексте машинного обучения. Еще одна область, на которую следует обратить внимание, – это алгоритмы подобия, часто применяемые к рекомендациям и предсказанию связей. *Алгоритмы подобия* (similarity algorithm) определяют, какие вершины больше всего похожи друг на друга, используя различные методы для сравнения таких элементов, как атрибуты вершин.

Массовый импорт данных Neo4j и Yelp

Импорт данных в Neo4j с использованием языка запросов Cypher использует транзакционный подход. Рисунок П.1 иллюстрирует этот процесс в общем виде.

Рис. П.1. Импорт данных с применением запросов Cypher

Хотя этот метод хорошо работает для инкрементной загрузки данных или массовой загрузки до 10 млн записей, при импорте исходных массивов данных лучшим выбором является инструмент импорта Neo4j. Этот инструмент создает файлы хранилища напрямую, минуя журнал транзакций, как показано на рис. П.2.

Рис. П.2. Использование инструмента импорта Neo4j

Инструмент импорта Neo4j обрабатывает файлы в формате CSV и ожидает, что у этих файлов будут определенные заголовки. На рис. П.3 показан пример файлов CSV, которые могут быть обработаны инструментом.

Вершины (узлы)

id:ID(User)	name
1234	Bob
1235	Alice
1236	Erika

id:ID(Review)	text	stars
678	Awesome	3
679	Mediocre	2
680	Really bad	1

Ребра (связи)

:START_ID(User)	:END_ID(Review)
1234	678
1235	679
1236	680

Рис. П.3. Формат файлов CSV, которые обрабатывает импорт Neo4j

Размер набора данных Yelp означает, что инструмент импорта Neo4j – наш лучший выбор. Данные представлены в формате JSON, поэтому сначала нам нужно преобразовать их в формат, ожидаемый инструментом импорта Neo4j. На рис. П.4 показан пример файла JSON, который нам нужно преобразовать в CSV.

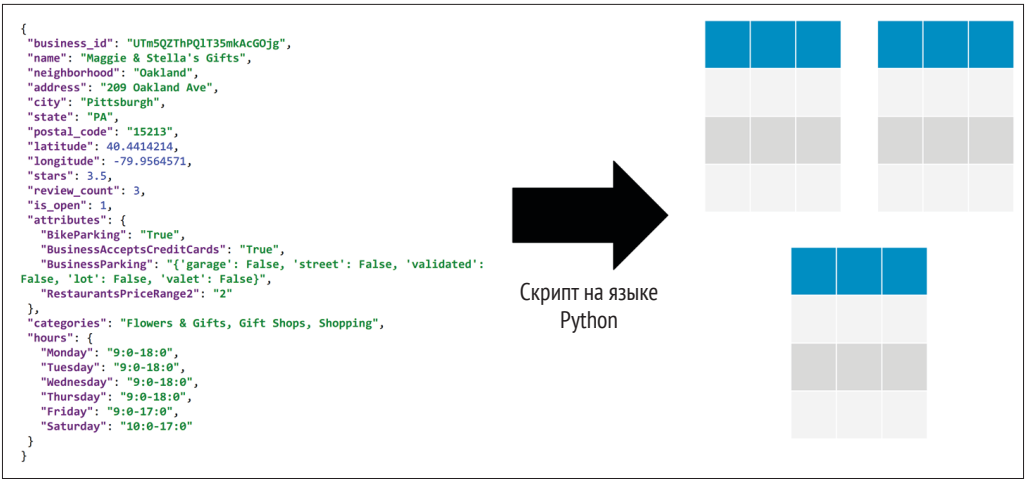


Рис. П.4. Преобразование JSON в CSV

Используя Python, мы можем создать простой скрипт для преобразования данных в файл CSV. Как только мы преобразовали данные в этот формат, можем импортировать их в Neo4j. Подробные инструкции, объясняющие, как это сделать, находятся в файловом архиве книги.

APOC и другие инструменты Neo4j

APOC (Awesome procedures on Cypher) – это библиотека, которая содержит более 450 процедур и функций, помогающих выполнять общие задачи, такие как интеграция данных, очистка и преобразование данных, а также справочная система. APOC входит в состав стандартного дистрибутива Neo4j.

Neo4j также имеет другие инструменты, которые можно использовать вместе с библиотекой графовых алгоритмов, например приложение «песочница» для знакомства с алгоритмами без кода. Эти инструменты можно найти на сайте для разработчиков графовых алгоритмов по адресу https://neo4j.com/developer/graph-algorithms/.

Поиск наборов данных

Найти набор графических данных, который соответствует целям или гипотезам тестирования, бывает непросто. В дополнение к изучению иссле-

довательских работ рассмотрите возможность использования наборов сетевых данных:

- Stanford Network Analysis Project (SNAP) включает в себя несколько наборов данных, а также соответствующие документы и руководства по использованию: https://snap.stanford.edu/index.html;
- Colorado Index of Complex Networks (ICON) – это поисковый индекс наборов сетевых данных исследовательского качества из различных областей сетевой науки: https://icon.colorado.edu/;
- Koblenz Network Collection (KONECT) включает большие наборы сетевых данных различных типов для проведения исследований в области сетевых наук http://konect.uni-koblenz.de/.

Большинство наборов данных нуждаются в некоторой предварительной обработке для преобразования в более полезный формат.

Помощь в освоении платформ Apache Spark и Neo4j

В интернете существует множество онлайн-ресурсов для желающих освоить платформы Apache Spark и Neo4j. Если у вас есть конкретные вопросы, мы рекомендуем вам обратиться к соответствующим сообществам:

- по общим вопросам Spark подпишитесь на users@spark.apache.org на странице сообщества Spark по адресу https://spark.apache.org/community.html;
- для вопросов по использованию GraphFrames используйте трекер GitHub по адресу https://github.com/graphframes/graphframes/issues;
- по всем вопросам использования Neo4j (в том числе по поводу графовых алгоритмов) посетите веб-сайт сообщества пользователей Neo4j по адресу https://community.neo4j.com/.

Дополнительные курсы

В интернете есть много отличных ресурсов, посвященных основам анализа графов. Поиск курсов или книг по графовым алгоритмам, сетевым наукам и анализу сетей откроет перед вами множество вариантов. Вот несколько отличных примеров онлайн-курсов:

- курс «Прикладной анализ социальных сетей на языке Python» на обучающем портале Coursera https://www.coursera.org/learn/python-social-network-analysis;

- курс лекций Леонида Жукова «Анализ социальной сети» на YouTube https://www.youtube.com/channel/UCqwvCUUnfyL_MdA_uQPZF_A;
- Стэнфордский курс «Анализ сетей» включает видео-лекции, списки литературы и другие ресурсы http://web.stanford.edu/class/cs224w/;
- сайт https://www.complexityexplorer.org/ предлагает онлайн-курсы по *теории сложных систем* (complexity science).

Об авторах

Марк Нидхем (Mark Needham) – знаток графов и инженер по связанным данным в Neo4j. Он старается помочь пользователям освоить графы и Neo4j, находя продвинутые решения сложных проблем с данными. Марк имеет большой опыт работы с графовыми данными, ранее он участвовал в создании системы причинно-следственной кластеризации Neo4j. Он пишет о своем опыте работы с графами в популярном блоге по адресу https://markhneedham.com/blog/ и в Твиттере: @markhneedham.

Эми Ходлер (Amy Hodler) – преданный энтузиаст сетевых наук, менеджер программ по искусственному интеллекту и анализу графов в Neo4j. Она продвигает использование анализа графов для выявления структур в реальных сетях и прогнозирования поведенческой динамики. Эми помогает командам применять новые подходы и находить новые возможности в таких компаниях, как EDS, Microsoft, Hewlett-Packard (HP), Hitachi IoT и Cray Inc. Эми сочетает любовь к науке и искусству, увлекаясь исследованиями сложных процессов и теорией графов. Она пишет в Твиттере: @amyhodler.

Об изображении на обложке

На обложке этой книги изображен *европейский садовый паук* (Araneus diadematus), распространенный в Европе, а также в Северной Америке, куда его случайно завезли европейские поселенцы.

Европейский садовый паук имеет длину менее дюйма и пестрый коричневый окрас с бледными отметинами. Некоторые из пятен расположены таким образом, что они образуют на спине небольшой крест, откуда возникло общее название этой разновидности пауков – паук-крестовик. Эти пауки равномерно распространены по всему ареалу и чаще всего заметны в конце лета, когда становятся взрослыми и начинают плести свои сети.

Европейские садовые пауки – это кругопряды, т. е. они плетут круговую паутину, в которую ловят мелких насекомых. Паутина часто используется и обновляется ночью, чтобы обеспечить максимальную эффективность. Когда паук оставляет паутину вне поля зрения, одна из его ног опирается на «сигнальную линию», соединенную с паутиной. Вибрация сигнальной нити предупреждает паука о наличии борющейся добычи. Паук стремительно возвращается на паутину, чтобы укусить свою жертву и ввести в нее специальные ферменты, облегчающие переваривание. Когда паутину беспокоят хищники или иное внешнее воздействие, европейские садовые пауки встряхивают паутину ногами, а затем спускаются на землю на нити своего шелка. Когда опасность проходит, паук использует эту нить, чтобы вернуться на паутину.

Садовые пауки живут один год – после вылупления весной новое поколение созревает в течение лета, а затем приступает к спариванию. Самцы приближаются к самкам с осторожностью, поскольку самки иногда убивают и поедают самцов. После спаривания самка паука плетет плотный шелковый кокон для своих яиц перед тем, как умереть осенью.

Эти хорошо изученные пауки широко распространены и адаптированы к местам обитания человека. В 1973 году две самки садового паука по имени Арабелла и Анита участвовали в эксперименте на борту орбитального аппарата Skylab NASA для проверки влияния невесомости на конструкцию паутины. После начального периода адаптации к невесомости Арабелла построила сначала частичную сеть, а затем полностью сформированную круглую паутину.

Многие из животных на обложках книг O'Reilly находятся под угрозой исчезновения; все они важны для мира.

Изображение на обложке – цветная иллюстрация Карен Монтгомери (Karen Montgomery) на основе черно-белой гравюры от Meyers Kleines Lexicon.

Предметный указатель

Б

В

Г

Д

Книги издательства «ДМК ПРЕСС» можно купить оптом и в розницу в книготорговой компании «Галактика» (представляет интересы издательств «ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Адрес книготорговой компании «Галактика»: **г. Москва, пр. Андропова, 38**

Тел.: **+7(499) 782-38-89**.

Электронный почта: **books@alians-kniga.ru**.

Марк Нидхем
Эми Ходлер

Графовые алгоритмы

Практическая реализация
на платформах Apache Spark и Neo4j

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Перевод с английского *Яценков В. С.*
Корректор *Абросимова Л. А.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Печать цифровая. Усл. печ. л. 22,59.
Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com