
ДМК ПРЕСС
издательство

Ю. Лесковец, А. Раджараман, Дж. Ульман

# Анализ больших наборов данных

Юре Лесковец, Ананд Раджараман, Джеффри Д. Ульман

# Анализ больших наборов данных

# Mining of Massive Datasets

Jure Leskovec
Stanford University

Anand Rajaraman
Milliway Labs

Jeffrey D. Ullman
Stanford University.

CAMBRIDGE
UNIVERSITY PRESS

# Анализ больших наборов данных

Юре Лесковец

Ананд Раджараман

Джеффри Д. Ульман

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Москва, 2016

**УДК 004.6**
**ББК 32.972**
**Л50**

Л50 Юре Лесковец, Ананд Раджараман, Джеффри Д. Ульман
Анализ больших наборов данных. / Пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2016. – 498 с.: ил.

**ISBN 978-5-97060-190-7**

Эта книга написана ведущими специалистами в области технологий баз данных и веба. Благодаря популярности интернет-торговли появилось много чрезвычайно объемных баз данных, для извлечения информации из которых нужно применять методы добычи данных (data mining).

В книге описываются алгоритмы, которые реально использовались для решения важнейших задач добычи данных и могут быть с успехом применены даже к очень большим наборам данных. Изложение начинается с рассмотрения технологии MapReduce – важного средства распараллеливания алгоритмов. Излагаются алгоритмы хэширования с учетом близости и потоковой обработки данных, которые поступают слишком быстро для тщательного анализа. В последующих главах рассматривается идея показателя PageRank, нахождение частых предметных наборов и кластеризация. Во второе издание включен дополнительный материал о социальных сетях, машинном обучении и понижении размерности.

Издание будет в равной мере полезна студентам и программистам-практикам.

Original English language edition published by Cambridge University Press, 132 Avenue of the Americas, New York, NY 10013-2473, USA. Copyright © 2010, 2011, 2012, 2013, 2014 Anand Rajaraman, Jure Leskovec, Jeffrey D. Ullman. Russian-language edition copyright © 2015 by DMK Press. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-118-62986-4 (англ.)

ISBN 978-5-97060-190-7 (рус.)

© 2010, 2011, 2012, 2013, 2014 Anand Rajaraman, Jure Leskovec, Jeffrey D. Ullman
© Оформление, издание, ДМК Пресс, 2016

# ОГЛАВЛЕНИЕ

## ГЛАВА 2. MapReduce и новый программный стек ..... 39

# ПРЕДИСЛОВИЕ

В основу этой книги положен материал односеместрового курса, который Ананд Раджараман и Джефф Ульман в течение нескольких лет читали в Стэнфордском университете. Курс CS345A под названием «Добыча данных в вебе» задумывался как спецкурс для аспирантов, но оказался доступным и полезным также старшекурсникам. Когда в Стэнфорд пришел преподавать Юре Лесковец, мы существенно изменили организацию материала. Он начал читать новый курс CS224W по анализу сетей и расширил материал курса CS345A, который получил номер CS246. Втроем авторы также подготовили курс CS341, посвященный крупномасштабному проекту в области добычи данных. В своем теперешнем виде книга содержит материал всех трех курсов.

## О чем эта книга

В самых общих словах, эта книга о добыче данных. Но акцент сделан на анализе данных очень большого объема, не помещающихся в оперативную память. Поэтому многие примеры относятся к вебу или к данным, полученным из веба. Кроме того, в книге принят алгоритмический подход: добыча данных – это применение алгоритмов к данным, а не использование данных для «обучения» той или иной машины. Ниже перечислены основные рассматриваемые темы.

1. Распределенные файловые системы и технология распределения-редукции (map-reduce) как средство создания параллельных алгоритмов, успешно справляющихся с очень большими объемами данных.
2. Поиск по сходству, в том числе такие важнейшие алгоритмы, как MinHash и хэширование с учетом близости (locality sensitive hashing).
3. Обработка потоков данных и специализированные алгоритмы для работы с данными, которые поступают настолько быстро, что либо обрабатываются немедленно, либо теряются.
4. Принципы работы поисковых систем, в том числе алгоритм Google PageRank, распознавание ссылочного спама и метод авторитетных и хаб-документов.
5. Частые предметные наборы, в том числе поиск ассоциативных правил, анализ корзины, алгоритм Apriori и его усовершенствованные варианты.
6. Алгоритмы кластеризации очень больших многомерных наборов данных.

7. Две важные для веб-приложений задачи: управление рекламой и рекомендательные системы.
8. Алгоритмы анализа структуры очень больших графов, в особенности графов социальных сетей.
9. Методы получения важных свойств большого набора данных с помощью понижения размерности, в том числе сингулярное разложение и латентно-семантическое индексирование.
10. Алгоритмы машинного обучения, применимые к очень большим наборам данных, в том числе перцептроны, метод опорных векторов и градиентный спуск.

# Требования к читателю

Для полного понимания изложенного в книге материала мы рекомендуем:

1. Прослушать вводный курс по системам баз данных, включая основы SQL и сопутствующих систем программирования.
2. Иметь знания о структурах данных, алгоритмах и дискретной математике в объеме второго курса университета.
3. Иметь знания о программных системах, программной инженерии и языках программирования в объеме второго курса университета.

# Упражнения

В книге много упражнений, они есть почти в каждом разделе. Более трудные упражнения или их части отмечены восклицательным знаком, а самые трудные – двумя восклицательными знаками.

# Поддержка в вебе

Слайды, домашние задания, проектные требования и экзаменационные задачи из курсов, примыкающих к этой книге, можно найти по адресу `http://www.mmds.org`.

# Автоматизированные домашние задания

На основе этой книги составлены автоматизированные упражнения с применением системы проверочных вопросов Gradiance, доступной по адресу `www.gradiance.com/services`. Студенты могут стать членами открытой группы, создав на этом сайте учетную запись и присоединившись к группе с кодом `1EDD8A1D`. Преподаватели также могут воспользоваться этим сайтом, для этого нужно создать учетную запись и отправить сообщение на адрес `support@gradiance.com`,

указав в нем свой логин, название учебного заведения и запрос на право использования материалов к книге (MMDS).

# Благодарности

Мы благодарны Фото Афрати (Foto Afrati), Аруну Маратхи (Arun Marathe) и Року Сосику (Rok Sosic) за критическое прочтение рукописи.

Об ошибках также сообщали Раджив Абрахам (Rajiv Abraham), Апурв Агарвал (Apoorv Agarwal), Арис Анагностопулос (Aris Anagnostopoulos), Атилла Сонер Балкир (Atilla Soner Balkir), Арно Бельтуаль (Arnaud Belletoile), Робин Беннетт (Robin Bennett), Сьюзан Бьянкани (Susan Biancani), Амитабх Чаудхари (Amitabh Chaudhary), Леланд Чен (Leland Chen), Анастасиос Гунарис (Anastasios Gounaris), Шрей Гупта (Shrey Gupta), Валид Хамеид (Waleed Hameid), Саман Харати-заде (Saman Haratizadeh), Лаклан Канг (Lachlan Kang), Эд Кнорр (Ed Knorr), Хэй Вун Квак (Haewoon Kwak), Эллис Лау (Ellis Lau), Грег Ли (Greg Lee), Этан Лозано (Ethan Lozano), Ю Нань Люо (Yunan Luo), Майкл Махоуни (Michael Mahoney), Джастин Мейер (Justin Meyer), Брайант Москон (Bryant Moscon), Брэд Пенофф (Brad Penoff), Филипс Коко Прасетийо (Philips Kokoh Prasetyo), Ки Ге (Qi Ge), Рич Сейтер (Rich Seiter), Хитэш Шетти (Hitesh Shetty), Ангад Сингх (Angad Singh), Сандип Срипада (Sandeep Sripada), Дэннис Сидхарта (Dennis Sidharta), Кшиштоф Стенсел (Krzysztof Stencel), Марк Сторус (Mark Storus), Рошан Сумбалай (Roshan Sumbaly), Зак Тэйлор (Zack Taylor), Тим Триш мл. (Tim Triche Jr.), Вань Бин (Wang Bin), Вэнь Цзен Бин (Weng Zhen-Bin), Роберт Уэст (Robert West), Оскар Ву (Oscar Wu), Се Ке (Xie Ke), Николас Чжао (Nicolas Zhao) и Чжу Цзинь Бо (Zhou Jingbo). Разумеется, все оставшиеся незамеченными ошибки – наша вина.

*Ю. Л.*
*А. Р.*
*Дж. Д. У.*
Пало-Альто, Калифорния
март 2014

# ГЛАВА 1. Добыча данных

В этой вводной главе мы опишем, в чем состоит сущность добычи данных, и обсудим, как добыча данных трактуется в различных дисциплинах, которые вносят свой вклад в эту область. Мы рассмотрим «принцип Бонферрони», предупреждающий об опасностях чрезмерного увлечения добычей данных. В этой же главе мы кратко упомянем некоторые идеи, которые, хотя сами и не относятся к добыче данных, но полезны для понимания ряда важных идей, относящихся к этой тематике. Мы имеем в виду метрику важности слов TF.IDF, поведение хэш-функций и индексов, а также некоторые тождества, содержащие число e, основание натуральных логарифмов. Наконец, мы расскажем о темах, рассматриваемых в этой книге.

## 1.1. Что такое добыча данных?

Многие разделяют определение «добычи данных» как выявление «моделей» данных. Однако под моделью можно понимать разные вещи. Ниже описываются наиболее важные направления моделирования.

### *1.1.1. Статистическое моделирование*

Первыми термин «добыча данных» ввели в обиход специалисты по математической статистике. Первоначально словосочетание «data mining» (добыча данных) или «data dredging» (вычерпывание данных) имело несколько пренебрежительный оттенок и обозначало попытки извлечь информацию, которая явно не присутствовала в данных. В разделе 1.2 демонстрируются различные ошибки, которые могут возникнуть, если пытаться извлечь то, чего в данных на самом деле нет. В наши дни термин «добыча данных» употребляется в положительном смысле. Теперь статистики рассматривают добычу данных как средство построения статистической модели, т. е. закона, в соответствии с которым распределены видимые данные.

**Пример 1.1.** Пусть данными будет множество чисел. Эти данные намного проще тех, что подвергаются добыче, но для примера вполне подойдут. Статистик может предположить, что данные имеют гауссово распределение и по известным формулам вычислить наиболее вероятные параметры этого распределения.

Среднее и стандартное отклонение полностью определяют гауссово распределение и потому могут служить моделью данных.

## 1.1.2. Машинное обучение

Некоторые считают, что добыча данных и машинное обучение – синонимы. Безусловно, для добычи данных иногда используются алгоритмы, применяемые в машинном обучении. Специалисты по машинному обучению используют данные как обучающий набор и на них обучают алгоритм того или иного вида, например: байесовские сети, метод опорных векторов, решающие деревья, скрытые марковские модели и т. п.

В некоторых ситуациях использование данных подобным образом имеет смысл. В частности, машинное обучение дает хороший результат, когда мы плохо представляем себе, что искать в данных. Например, совсем неясно, из-за каких особенностей одним людям фильм нравится, а другим – нет. Поэтому принявшие «вызов Netflix» – изобрести алгоритм, который предсказывал бы оценку фильма пользователями на основе выборки из их прошлых ответов, – с большим успехом применили алгоритмы машинного обучения. Мы обсудим простую форму алгоритма такого типа в разделе 9.4.

С другой стороны, машинное обучение не приносит успеха в ситуациях, когда цели добычи данных можно описать более конкретно. Интересный пример – попытка компании WhizBang! Labs[1] использовать методы машинного обучения для поиска резюме, которые люди размещают в сети. У нее не получилось добиться результатов, лучших, чем дают вручную составленные алгоритмы, которые ищут очевидные слова и фразы, встречающиеся в типичном резюме. Всякий, кто читал или писал резюме, довольно отчетливо представляет, что в нем содержится, поэтому как выглядит веб-страница, содержащая резюме, – никакая не тайна. Потому-то применение машинного обучение не дало выигрыша по сравнению с составленным в лоб алгоритмом распознавания резюме.

## 1.1.3. Вычислительные подходы к моделированию

Сравнительно недавно на добычу данных стали смотреть как на алгоритмическую задачу. В этом случае модель данных – это просто ответ на сложный запрос к данным. Например, если дано множество чисел, как в примере 1.1, то можно было бы вычислить их среднее и стандартное отклонение. Отметим, что эти значения необязательно являются параметрами гауссова распределения, которое лучше всего аппроксимирует данные, хотя при достаточно большом наборе данных они почти наверняка будут близки к ним.

Есть много подходов к моделированию данных. Мы уже упомянули одну возможность: построить статистический процесс, с помощью которого данные могли

[1] Эта компания пыталась использовать методы машинного обучения для анализа очень большого объема данных и наняла для этого много высококлассных специалистов. К сожалению, выжить ей не удалось.

быть сгенерированы. Большинство прочих подходов к моделированию можно отнести к одной из двух категорий.

1. Краткое и приближенное обобщение данных или
2. Извлечение из данных наиболее существенных признаков с отбрасыванием всего остального.

В следующих разделах мы исследуем оба подхода.

## 1.1.4. Обобщение

Одна из самых интересных форм обобщения – идея алгоритма PageRank, так успешно примененная Google; мы будем рассматривать ее в главе 5. При такой форме добычи данных вся сложная структура веба сводится к одному числу для каждой страницы. Несколько упрощая, это число, «ранг страницы» (PageRank), можно описать как вероятность того, что пользователь, случайно обходящий граф, окажется на этой странице в любой заданный момент времени. Замечательное свойство такого ранжирования заключается том, что оно очень хорошо отражает «важность» страницы – в какой мере типичный пользователь поисковой системы хотел бы видеть данную страницу в ответе на свой запрос.

Еще один важный вид обобщения – кластеризация – будет рассмотрен в главе 7. В этом случае данные рассматриваются как точки в многомерном пространстве. Те точки, которые в некотором смысле «близки», помещаются в один кластер. Сами кластеры также обобщаются, например, путем указания центроида кластера и среднего расстояния от центроида до всех точек. Совокупность обобщенных характеристик кластеров становится обобщением всего набора данных.

**Пример 1.2.** Знаменитый пример применения кластеризации для решения задачи имел место много лет назад в Лондоне, когда никаких компьютеров еще не было[2]. Врач Джон Сноу, сражаясь со вспышкой холеры, нанес места проживания заболевших на карту города. На рис. 1.1 показана упрощенная иллюстрация этой процедуры.

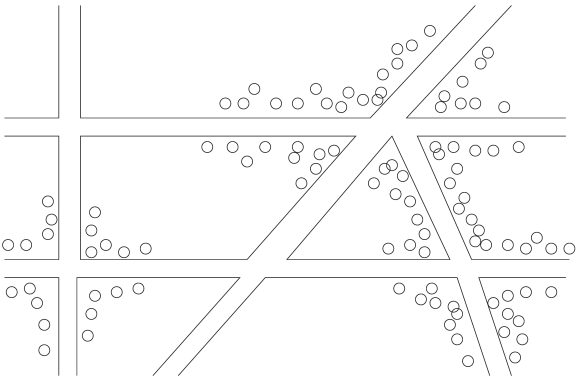


**Рис. 1.1.** Случаи холеры на карте Лондона

[2] См. http://en.wikipedia.org/wiki/1854 Broad Street cholera outbreak

Как видно, образовалось несколько кластеров в районе перекрестков. На этих перекрестках находились зараженные водоразборные колонки; жившие поблизости от них заболели, те же, кто жил рядом с незараженными колонками, остались здоровы. Не будь возможности кластеризовать данные, причина холеры осталась бы невыясненной.

### 1.1.5. Выделение признаков

В типичной модели на основе признаков ищутся экстремальные примеры некоторого явления, и данные представляются с помощью этих примеров. Если вы знакомы с *байесовскими сетями*, одной из ветвей машинного обучения, которая в этой книге не рассматривается, то знаете, что в них сложные связи между объектами представляются с помощью отыскания самых сильных статистических зависимостей и использования только их для представления всех статистических связей. Мы изучим следующие важные формы выделения признаков из больших наборов данных.

1. *Частые предметные наборы.* Эта модель имеет смысл, когда данные состоят из «корзин», содержащих небольшие наборы предметов, как, например, в задаче об анализе корзин покупок, обсуждаемой в главе 6. Мы ищем небольшие наборы предметов, которые встречаются вместе во многих корзинах, и считаем эти «частые предметные наборы» искомой характеристикой данных. Первоначально такой вид добычи данных применялся к настоящим корзинам покупок: поиску предметов, например гамбургер и кетчуп, которые люди покупают вместе в небольшой лавке или в супермаркете.
2. *Похожие предметы.* Часто данные имеют вид коллекции наборов, а цель состоит в том, чтобы найти пары наборов, в которых относительно много общих элементов. Например, покупателей в интернет-магазине типа Amazon можно рассматривать как наборы купленных ими товаров. Чтобы предложить покупателю еще что-нибудь, что могло бы ему понравиться, Amazon может поискать «похожих» покупателей и порекомендовать товары, которые покупали многие из них. Этот процесс называется «коллаборативной фильтрацией». Если бы все покупатели были целеустремленными, т. е. покупали бы только одну вещь, то могла бы сработать кластеризация покупателей. Но обычно покупателей интересуют разные вещи, поэтому полезнее для каждого покупателя найти небольшое число покупателей со схожими вкусами и представить данные такими связями. Проблему сходства мы будем обсуждать в главе 3.

## 1.2. Статистические пределы добычи данных

Типичная задача добычи данных – обнаружение необычных событий, скрытых в массивном объеме данных. В этом разделе мы рассмотрим эту проблему и заодно

«принцип Бонферрони» – предостережение против излишне ревностных попыток добыть данные.

## 1.2.1. Тотальное владение информацией

В 2002 году администрация Буша выдвинула план – подвергнуть анализу все данные, до которых можно дотянуться, в том числе чеки, оплаченные кредитной картой, данные о регистрации в гостиницах, данные о поездках и многие иные виды информации, – с целью отслеживания террористической деятельности. Эта идея, естественно, вызвала недовольство у поборников защиты частной жизни, и в итоге весь проект, названный TIA, или *Total Information Awareness* (*тотальное владение информацией*), был похоронен Конгрессом, хотя не исключено, что он все же существует под другим именем. В этой книге мы не собираемся обсуждать трудную проблему поиска компромисса между безопасностью и конфиденциальностью. Однако в связи с проектом TIA или подобной системой возникает ряд технических вопросов касательно практической осуществимости и реалистичности предположений.

Многие задавались вопросом: если исследовать так много данных, пытаясь найти следы деятельности, характерной для террористов, то не получится ли, что мы найдем много совершенно невинных действий – или даже незаконных, но не относящихся к терроризму, – и человеку придется свести знакомство с полицией, а, может, и не просто знакомство? Здесь все зависит от того, насколько узко определена интересующая нас деятельность. Статистики сталкивались с многообразными проявлениями этой проблемы и выдвинули теорию, начатки которой мы изложим в следующем разделе.

## 1.2.2. Принцип Бонферрони

Пусть имеются какие-то данные, и мы ищем в них события определенного вида. Можно ожидать, что такие событие встретятся, даже если данные выбраны абсолютно случайно, а количество событий будет расти вместе с объемом данных. Эти события «фиктивные» в том смысле, что у них нет никакой причины, помимо случайности данных, а в случайных данных всегда встретится какое-то количество необычных признаков, которые, хотя и выглядят значимыми, на самом деле таковыми не являются. Теорема математической статистики, известная под названием *поправка Бонферрони*, дает статистически корректный способ избежать большинства таких ложноположительных ответов на поисковый запрос. Не вдаваясь в технические детали, мы предложим ее неформальный вариант, *принцип Бонферрони*, который поможет избежать трактовки случайных фактов как реальных. Вычислите ожидаемое число искомых событий в предположении, что данные случайны. Если это число существенно больше количества реальных событий, которые вы надеетесь обнаружить, то следует ожидать, что почти все найденные события фиктивные, т. е. являются статистическими артефактами, а не свидетельством в пользу того, что вы ищете. Это наблюдение и есть неформальный принцип Бонферрони.

В случае поиска террористов, когда мы ожидаем, что сколько-то террористов действуют в любой момент времени, принцип Бонферрони гласит, что обнаружить террористов можно, только выискивая события настолько редкие, что в случайных данных их появление крайне маловероятно. Развернутый пример мы приведем в следующем разделе.

### 1.2.3. Пример применения принципа Бонферрони

Допустим, мы полагаем, что где-то действуют «злоумышленники», и хотим их обнаружить. Допустим также, что есть основания полагать, что злоумышленники периодически встречаются в гостинице, чтобы спланировать свой злой умысел. Сделаем следующие предположения о размере задачи:

1. Есть миллиард людей, среди которых могут быть злоумышленники.
2. Любой человек останавливается в гостинице один день из 100.
3. Гостиница вмещает 100 человек. Следовательно, 100 000 гостиниц будет достаточно, чтобы разместить 1 % от миллиарда людей, которые останавливаются в гостинице в каждый конкретный день.
4. Мы изучаем данные о регистрации в гостиницах за 1000 дней.

Чтобы найти в этих данных злоумышленников, мы будем искать людей, которые в два разных дня останавливались в одной и той же гостинице. Допустим, однако, что в действительности никаких злоумышленников нет. То есть все ведут себя случайным образом, с вероятностью 0,01 решая в данный день остановиться в какой-то гостинице и при этом случайно выбирая одну из $10^5$ гостиниц. Найдем ли мы пары людей, которые выглядят как злоумышленники? Можно выполнить простое вычисление. Вероятность того, что два произвольных человека решат остановиться в гостинице в данный день, составляет 0,0001. Вероятность того, что они остановятся в одной и той же гостинице в один и тот же день равна $10^{-9}$. Вероятность, что они остановятся в одной и той же гостинице в два разных дня, равна квадрату этого числа, т. е. $10^{-18}$. Отметим, что выбранные в эти дни гостиницы могут быть разными.

Теперь надо посчитать, сколько событий указывают на злой умысел. Под «событием» здесь понимается пара людей и пара дней такие, что оба человека в каждый из этих двух дней останавливались в одной и той же гостинице. Чтобы упростить вычисления, заметим, что для больших $n$ $\binom{n}{2}$ приблизительно равно $n^2/2$. Таким образом, количество пар людей равно $\binom{10^9}{2} = 5 \times 10^{17}$. Количество пар дней равно $\binom{1000}{2} = 5 \times 10^5$. Ожидаемое число событий, выглядящих как злоумышление, равно произведению количества пар людей на количество пар дней и на вероятность того, что пара людей и пара дней демонстрируют искомое поведение. Это число равно

$$5 \times 10^{17} \times 5 \times 10^5 \times 10^{-18} = 250\,000.$$

То есть четверть миллиона людей будут казаться злоумышленниками, даже если не являются таковыми.

Теперь предположим, что в действительности существует 10 пар злоумышленников. Полиции придется проверить четверть миллиона других пар, чтобы найти настоящих злоумышленников. Мало того что это означает вторжение в частную жизнь полумиллиона ни в чем неповинных людей, так еще и объем работы настолько велик, что такой подход к поиску злоумышленников практически неосуществим.

### 1.2.4. Упражнения к разделу 1.2

**Упражнение 1.2.1.** Используя сведения из раздела 1.2.3, найдите количество подозрительных пар, если внести в данные следующие изменения (сохранив все прочие числа)?

(*а*) Увеличить количество дней наблюдения до 2000.

(*б*) Увеличить количество наблюдаемых людей до 2 миллиардов (а количество гостиниц соответственно до 200 000).

(*с*) Считать двух человек подозрительными, если они останавливались в одной и той же гостинице в три разных дня.

! **Упражнение 1.2.2.** Предположим, что у нас есть информация о покупках 100 миллионов людей в супермаркетах. Каждый человек заходит в супермаркет 100 раз в год и покупает 10 из 1000 предлагаемых там товаров. Мы думаем, что двое террористов в какой-то день на протяжении года купят в точности один и тот же набор предметов (быть может, компонентов бомбы). Если мы будем искать пары людей, купивших одинаковые наборы предметов, то можно ли ожидать, что найденные люди действительно террористы[3]?

## 1.3. Кое-какие полезные сведения

В этом разделе содержится краткое введение в темы, с которыми вы, возможно, знакомились на других курсах. Все эти сведения будут полезны при изучении добычи данных.

1. Мера важности слов TF.IDF.
2. Хэш-функции и их применение.
3. Внешняя память (диск) и ее влияние на время работы алгоритмов.
4. Основание натуральных логарифмов *e* и тождества, содержащие эту константу.
5. Степенные зависимости.

### 1.3.1. Важность слов в документах

В нескольких приложениях добычи данных мы столкнемся с проблемой классификации документов (последовательностей слов) по тематике. Как правило, тема определяется путем поиска специальных слов, которые характеризуют относя-

[3] То есть наша гипотеза состоит в том, что террористы наверняка купят набор из десяти одинаковых предметов в какой-то день на протяжении года. Мы не хотим обсуждать вопрос о том, характерно ли такое поведение для настоящих террористов.

щиеся к ней документы. Например, в статьях о бейсболе будут часто встречаться слова «мяч», «бита», «бросок», «пробежка» и т. д. После того как документы отнесены к теме бейсбола, нетрудно заметить, что подобные слова встречаются в них аномально часто. Но пока классификация не произведена, выделить эти слова как характеристические невозможно.

Таким образом, классификация часто начинается с изучения документов и отыскания в них важных слов. Первая гипотеза может состоять в том, что слова, которые чаще всего встречаются в документе, и есть самые важные. Но в данном случае интуиция подсказывает ответ, прямо противоположный истинному положению дел. Самыми частыми, конечно же, будут наиболее употребительные слова типа «the» и «and», которые помогают выразить мысль, но сами по себе не несут никакого смысла. И действительно, несколько сотен наиболее употребительных слов английского языка (они называются *стоп-словами*), обычно исключаются из документов еще до попытки классификации.

На самом деле, индикаторами темы являются относительно редко встречающиеся слова. Но не все редкие слова одинаково полезны в качестве индикаторов. Некоторые слова, например «notwithstanding» (несмотря на) или «albeit» (пусть даже), хоть редко встречаются в коллекции документов, но ничего полезного не сообщают. С другой стороны, слово «chukker» (период в игре в поло), пожалуй, встречается не менее редко, но подсказывает, что данный документ посвящен игре в поло. Разница между значимыми и незначимыми редкими словами определяется концентрацией полезных слов в немногих документах. То есть присутствие в документе слова типа «albeit» не повышает вероятность его повторного появления. Но если в статье один раз встречается слово «chukker», то весьма вероятно, что оно входит в составе словосочетания «first chukker» (первый период), еще раз в «second chukker» (второй период) и т. д. То есть, если слово вообще встречается, то с большой вероятностью оно встретится несколько раз.

Формальная мера концентрации данного слова в относительно небольшом количестве документов называется TF.IDF (частота терма, помноженная на обратную частоту документа). Обычно она вычисляется следующим образом. Пусть есть коллекция из $N$ документов. Обозначим $f_{ij}$ *частоту* (количество вхождений) терма (слова) $i$ в документ $j$ и определим *частоту терма* $TF_{ij}$ такой формулой:

$$TF_{ij} = \frac{f_{ij}}{\max_k f_{kj}}.$$

Иначе говоря, частота терма $i$ в документе $j$ равна величине $f_{ij}$, нормированной путем деления на максимальное количество вхождений этого терма (возможно, после исключения стоп-слов) в один и тот же документ. Следовательно, у самого часто встречающего терма в документе $j$ TF будет равно 1, а у всех остальных меньше.

IDF терма определяется следующим образом. Пусть терм $i$ встречается в $n_i$ документах из коллекции, содержащей всего $N$ документов. Тогда $IDF_i = \log_2(N/n_i)$. Оценка TF.IDF для терма $i$ в документе $j$ определяется как $TF_{ij} \times IDF_i$. Именно тер-

мы с наибольшей оценкой TF.IDF часто наилучшим образом характеризуют тему документа.

**Пример 1.3.** Предположим, что репозиторий содержит $2^{20} = 1\,048\,576$ документов. Пусть слово $w$ встречается в $2^{10} = 1024$ документов. Тогда $IDF_w = \log_2(2^{20}/2^{10}) = \log_2(2^{10}) = 10$. Рассмотрим документ $j$, в котором терм $w$ встречается 20 раз, и пусть это максимальное количество вхождений одного слова (возможно, после исключения стоп-слов). Тогда $TF_{wj} = 1$ и оценка TF.IDF терма $w$ в документе $j$ равна 10.

Предположим, что в документе $k$ слово $w$ встречается один раз, тогда как максимальное количество вхождений одного слова в этом документе равно 20. Тогда $TF_{wk} = 1/20$, а оценка TF.IDF для $w$ в документе $k$ равна 1/2.

## 1.3.2. Хэш-функции

Вы, вероятно, слышали о хэш-таблицах и, возможно, использовали их в Java-классах или других подобных пакетах. Хэш-функции, лежащие в основе хэш-таблиц, находят важное применение и во многих алгоритмах добычи данных, в которых хэш-таблицы принимают необычную форму. В этом разделе мы рассмотрим основные понятия.

Прежде всего, хэш-функция $h$ принимает *ключ хэширования* в качестве аргумента и возвращает *номер ячейки* (*bucket*). Номер ячейки – это целое число, обычно в диапазоне от 0 до $B - 1$, где $B$ – количество ячеек. Тип ключа хэширования может быть любым. На интуитивном уровне хэш-функция «рандомизирует» ключи хэширования. Точнее, если ключи хэширования случайным образом выбираются из разумной совокупности возможных ключей, то $h$ поместит в каждую из $B$ ячеек примерно одинаковое количество ключей. Это было бы невозможно, если, к примеру, размер совокупности ключей хэширования меньше $B$. Такая совокупность не считается «разумной». Однако есть немало более тонких причин, по которым хэш-функция может не давать приблизительно равномерного распределения по ячейкам.

**Пример 1.4.** Допустим, что ключи хэширования – положительные целые числа. Простая и употребительная хэш-функция – $h(x) = x \bmod B$ – возвращает остаток от деления $x$ на $B$. Она неплохо работает, если совокупность ключей хэширования – множество всех положительных целых чисел. Тогда доля ключей, попавших в каждую ячейку, составит $1/B$. Но предположим, что наша совокупность содержит только четные числа и пусть $B = 10$. Тогда значениями $h(x)$ могут быть только ячейки с номерами 0, 2, 4, 6, 8, так что поведение хэш-функции заведомо не случайно. С другой стороны, если взять $B = 11$, то окажется, что доля четных чисел в каждой из 11 ячеек равна 1/11, т. е. хэш-функция работает очень хорошо.

Обобщая пример 1.4, можно сказать, что если ключами хэширования являются целые числа, то при выборе в качестве $B$ числа, имеющего общий множитель со

всеми возможными ключами (или хотя бы с их большинством), распределение по ячейкам будет неравномерным. Поэтому обычно в качестве $B$ берут простое число. При таком выборе снижаются шансы неслучайного поведения, хотя по-прежнему необходимо рассмотреть случай, когда все ключи делятся на $B$. Разумеется, есть много других типов хэш-функций, не зависящих от арифметики по модулю. Мы не станем пытаться систематизировать их здесь, но приведем несколько источников в списке литературы.

А что, если ключи хэширования не являются целыми числами? Вообще говоря, у любого типа данных имеется значение, состоящее из битов, а последовательность битов всегда можно интерпретировать как целое число. Однако существуют простые правила, позволяющие преобразовать наиболее употребительные типы в целые числа. Например, если ключ хэширования – строка, то мы можем преобразовать каждый символ в его значение в кодировке ASCII или Unicode, которое можно интерпретировать как небольшое целое число. Затем, перед делением на $B$, эти числа складываются. Если $B$ меньше типичной суммы кодов символов для некоторой совокупности строк, то распределение по ячейкам будет близко в равномерному. Если же $B$ больше, то мы можем разбить все символы строки на несколько групп. Затем конкатенируем коды символов в одной группе и рассматриваем результат как одно целое число. Складываем все получившиеся таким образом числа и делим на $B$, как и раньше. Например, если $B$ порядка миллиарда, т. е. $2^{30}$, то группировка символов по четыре даст 32-разрядные целые числа. Их суммы распределяются по миллиарду ячеек приблизительно равномерно.

Эта идея рекурсивно обобщается на более сложные типы данных.

- Если тип является записью, каждая компонента которой имеет свой тип, то рекурсивно преобразовать значение каждой компоненты в целое число, применяя алгоритм, соответствующий типу компоненты. Сложить целые числа, получившиеся для всех компонент, и преобразовать сумму в номер ячейки, разделив ее на $B$.
- Если тип является массивом, множеством или коллекцией элементов одного и того же типа, то преобразовать значения его элементов в целые числа, сложить результаты и разделить сумму на $B$.

## 1.3.3. Индексы

*Индекс* – это структура данных, которая позволяет эффективно находить объект по значениям одного или нескольких его элементов. Наиболее типична ситуация, когда объекты являются записями, а индекс строится по одному из полей каждой записи. Если известно значение $v$, то индекс позволяет найти все записи, в которых это поле имеет такое значение. Например, мы можем располагать файлом, содержащим тройки (имя, адрес, телефон), и построить индекс по полю «телефон». Зная номер телефона, мы можем с помощью индекса быстро найти одну или несколько записей с таким номером.

Есть много способов реализации индексов, и мы не собираемся приводить обзор на эту тему. В списке литературы имеются ссылки на источники для дальнейшего изучения. Однако отметим, что хэш-таблица – один из простых методов построения индекса. Одно или несколько полей, по которым строится индекс, образует ключ хэширования, подаваемый на вход хэш-функции. Хэш-функция применяется к ключу хэширования записи, а сама запись помещается в ячейку с номером, возвращенным хэш-функцией. Ячейка может представлять собой список записей в оперативной памяти или, скажем, блок на диске.

Впоследствии, имея значение ключа хэширования, мы можем хэшировать его, вычислить соответствующую ячейку и производить поиск только в этой ячейке, т. е. среди записей с указанным значением ключа хэширования. Если выбрать количество ячеек $B$ примерно того же порядка, что и количество записей в файле, то в каждой ячейке окажется сравнительно мало записей, и поиск по ячейке будет занимать мало времени.

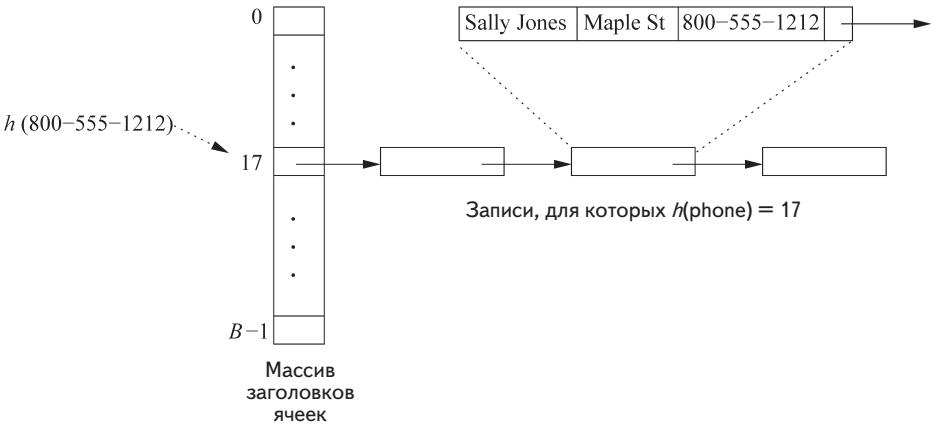



**Рис. 1.2.** Использование хэш-таблицы в качестве индекса; номер ячейки определяется хэш-кодом телефона, а сама запись помещается в ячейку, соответствующую хранящемуся в ней телефону

**Пример 1.5.** На рис. 1.2 показано, как в оперативной памяти может выглядеть индекс записей, содержащих имя, адрес и телефон. Здесь индекс построен по номеру телефона, а ячейки представляют собой связанные списки. Телефон 800-555-1212 хэшируется в ячейку 17. Существует массив *заголовков ячеек*, $i$-ый элемент которого является началом связанного списка для ячейки с номером $i$. На рисунке показан один развернутый элемент списка. Он содержит запись с полями имени, адреса и телефона. В этой конкретной записи хранится телефон 800-555-1212. В других записях той же ячейки телефон может быть таким же или отличающимся. Мы знаем лишь, что хэш-код телефона в любой записи из этой ячейки равен 17.

## 1.3.4. Внешняя память

При работе с большими данными важно понимать, насколько различается время вычислений в случаях, когда данные хранятся на диске и в оперативной памяти. Мы могли бы много чего сказать по поводу физических характеристик диска, но не будем увлекаться и дадим возможность интересующемуся читателю самому изучить рекомендуемую в конце главы литературу.

Диски организованы в виде совокупности *блоков* – минимальных единиц, используемых операционной системой для перемещения данных между диском и оперативной памятью. Так, в Windows размер блока составляет 64 КБ (т. е. $2^{16} = 65\,536$ байтов). Для *доступа* к диску (переместить головку считывания-записи к нужной дорожке и дождаться, пока под ней окажется нужный блок) и считывания блока требуется примерно 10 миллисекунд. Эта задержка по меньшей мере на пять порядков (в $10^5$ раз) превышает время считывания слова из оперативной памяти, т. е. если нам всего-то и нужно, что получить несколько байтов, то преимущества хранения данных в оперативной памяти не вызывают ни малейших сомнений. На самом деле, если мы хотим сделать что-то очень простое с каждым байтом в дисковом блоке, например рассматривать блок как ячейку хэш-таблицы и искать в этой ячейке записи с нужным значением ключа хэширования, то время, необходимое для перемещения блока с диска в оперативную память, окажется намного больше времени вычислений.

Если сделать так, чтобы связанные между собой данные располагались на одном *цилиндре* (множество блоков, отстоящих на одно и то же расстояние от центра диска, вследствие чего для чтения другого блока из того же цилиндра не нужно перемещать головку), то прочитать все блоки из этого цилиндра в оперативную память можно меньше, чем за 10 мс на блок. Можно считать, что диск не способен передавать данные в память со скоростью более ста миллионов байтов в секунду, как бы они ни были организованы. Если размер набора данных составляет мегабайт, то никакой проблемы нет. Но набор размером порядка сотен гигабайтов или терабайт сложно даже прочитать, не говоря уже о том, чтобы сделать нечто полезное.

## 1.3.5. Основание натуральных логарифмов

У числа $e = 2.7182818\ldots$ есть целый ряд полезных свойств. В частности, $e$ является пределом последовательности $\left(1+\frac{1}{x}\right)^x$ при $x$ стремящемся к бесконечности. Значения этого выражения для $x = 1, 2, 3, 4$ приблизительно равны 2, 2.25, 2.37, 2.44, так что нетрудно поверить, что предел этой последовательности близок к 2.72.

Простые алгебраические преобразования позволяют получить аппроксимации многих, на первый взгляд, сложных выражений. Возьмем, к примеру, выражение $(1+a)^b$, где $a$ мало. Его можно переписать в виде $(1+a)^{(1/a)(ab)}$. Если теперь подставить $a = 1/x$ и $1/a = x$, то получим

$$\left(1+\frac{1}{x}\right)^{x(ab)}, \text{ или } \left(\left(1+\frac{1}{x}\right)^{x}\right)^{ab}.$$

Так как $a$, по предположению, мало, то $x$ велико, следовательно, подвыражение $\left(1+\frac{1}{x}\right)^{x}$ будет близко к своему пределу $e$. Таким образом, $(1 + a)^b$ можно аппроксимировать выражением $e^{ab}$.

Аналогичные тождества имеют место для отрицательных $a$. То есть при $x$ стремящемся к бесконечности предел $\left(1-\frac{1}{x}\right)^{x}$ равен $1/e$. Отсюда следует, что аппроксимация $(1 + a)^b = e^{ab}$ справедлива и тогда, когда $a$ – малое отрицательное число. По-другому то же самое можно выразить, сказав, что $(1 - a)^b$ приближенно равно $e^{-ab}$, когда $a$ мало, а $b$ велико.

Другие полезные аппроксимации вытекают из разложения $e^x$ в ряд Тейлора: $e^x = \sum_{i=0}^{\infty} x^i / i!$, или $e^x = 1 + x + x^2/2 + x^3/6 + x^4/24 + \ldots$ . При больших $x$ этот ряд сходится медленно, но все же сходится, потому что $n!$ растет быстрее $x^n$ при любой константе $x$. Однако при малых $x$, все равно положительных или отрицательных, ряд сходится быстро и для получения хорошей аппроксимации достаточно всего нескольких членов.

**Пример 1.6.** Пусть $x = 1/2$. Тогда $e^{1/2} = 1+\frac{1}{2}+\frac{1}{8}+\frac{1}{48}+\frac{1}{348}+\ldots$ или приближенно $e^{1/2} = 1.64844$.

Пусть $x = -1$. Тогда $e^{-1} = 1 - 1+\frac{1}{2}+\frac{1}{6}+\frac{1}{24}+\frac{1}{120}+\frac{1}{720}+\frac{1}{5040}+\ldots$ или приближенно $e^{-1} = 0.36786$.

## 1.3.6. Степенные зависимости

Многие явления описываются уравнениями, в которых две переменные связаны *степенной зависимостью*, т. е. между логарифмами этих переменных существует линейная зависимость. На рис. 1.3 показан пример такой зависимости. Если $x$ – горизонтальная ось, а $y$ – вертикальная, то эта зависимость описывается формулой $\log_{10} y = 6 – 2 \log_{10} x$.

**Пример 1.7.** Мы могли бы заняться исследованием продаж книг на сайте Amazon.com и считать, что $x$ представляет ранг книги по продажам. Тогда $y$ – количество продаж книги, стоящей на $x$-ом месте по продажам, за некоторый период времени. Из графика на рис. 1.3 следует, что для книги, занявшей первое место, продано 1 000 000 экземпляров, для книги на 10-м месте – 10 000 экземпляров, на 100-м месте – 100 экземпляров и т. д. для всех рангов в этом диапазоне и вне его. Делать вывод о том, что для книги с рангом 1000 продана лишь часть экземпляра, было бы слишком смело – на

самом деле, мы ожидаем, что для рангов, существенно больших 1000, зависимость выйдет на плато.

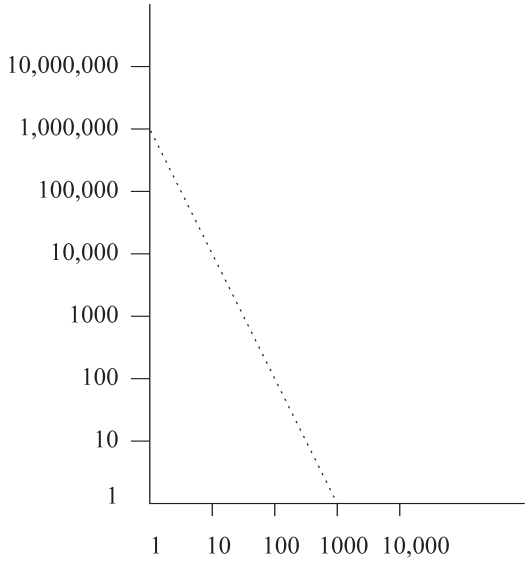



**Рис. 1.3.** Степенная зависимость с угловым коэффициентом –2

### Эффект Матфея

Часто существование степенных зависимостей, в которых показатель степени больше 1, объясняют *эффектом Матфея*. В Евангелии от Матфея есть слова о том, что «...всякому имеющему дастся и приумножится...». Такое поведение характерно для многих явлений: если значение некоторого свойства велико, то оно и далее продолжает увеличиваться. Например, если на веб-страницу ведет много ссылок, то пользователи будут находить ее с большей вероятностью, а, значит, и ставить на нее ссылки со своих страниц. Другой пример: если книга хорошо продается в Amazon, то, скорее всего, она будет рекламироваться пользователям, когда они заходят на сайт. И кто-то решит купить эту книгу, тем самым еще больше увеличив ее продажи.

В общем случае степенная зависимость между $x$ и $y$ описывается формулой $\log y = b + a \log x$. Если возвести основание логарифма (которое на самом деле ни на что не влияет), скажем $e$, в степень, равную левой и правой части этого уравнения, то получим $y = e^b e^{a \log x} = e^b x^a$. Поскольку $e^b$ – просто «некая константа», мы можем заменить ее константой $c$. Таким образом, степенная зависимость может быть записана в виде $y = cx^a$ при некоторых константах $a$ и $c$.

**Пример 1.8.** На рис. 1.3 мы видим, что если $x = 1$, то $y = 10^6$, а если $x = 1000$, то $y = 1$. Сделав первую подстановку, найдем, что $10^6 = c$. Вторая подстановка дает $1 = c(1000)^a$. Поскольку мы уже знаем, что $c = 10^6$, то из второго уравнения получаем $1 = 10^6(1000)^a$, откуда $a = -2$. Следовательно, зависимость, показанная на рис. 1.3, имеет вид $y = 10^6x^{-2}$ или $y = 10^6/x^2$.

В этой книге мы встретим много явлений, описываемых степенной зависимостью. Вот лишь некоторые примеры.

1. *Степени вершин в графе веба*. Упорядочим все страницы по количеству ведущих на страницу ссылок. Пусть $x$ – позиция страницы при таком упорядочении, а $y$ – количество ссылок, ведущих на $x$-ую страницу. Тогда $y$ – функция от $x$, очень похожая на ту, что изображена на рис. 1.3. Показатель степени $a$ в этом случае чуть больше –2; было показано, что он ближе к –2.1.
2. *Продажи товаров*. Упорядочим товары, скажем книги на сайте Amazon.com, по количеству продаж за прошлый год. Пусть $y$ – количество продаж книги, занимающей $x$-ое место в рейтинге популярности. И снова функция $y(x)$ похожа на рис. 1.3. Мы обсудим, какие следствия вытекают из такого распределения продаж, в разделе 9.1.2, где займемся проблемой «длинных хвостов».
3. *Размеры веб-сайтов*. Подсчитаем количество страниц веб-сайта и упорядочим все сайты по этому показателю. Пусть $y$ – количество страниц $x$-ого сайта. Функция $y(x)$ снова представляет собой степенную зависимость.
4. *Закон Ципфа*. Эта степенная зависимость первоначально относилась к частоте слов в коллекции документов. Если упорядочить слова по частоте и обозначить $y$ количество вхождений $x$-ого по порядку слова, то получится степенная зависимость, хотя и гораздо более пологая, чем на рис. 1.3. Ципф заметил, что $y = cx^{-1/2}$. Интересно, что данные других видов тоже описываются этим законом. Например, если упорядочить штаты США по численности населения и обозначить $y$ численность населения в $x$-ом по порядку штате, что $x$ и $y$ приближенно подчиняются закону Ципфа.

## 1.3.7. Упражнения к разделу 1.3

**Упражнение 1.3.1.** Допустим, что имеется репозиторий, содержащий 10 миллионов документов. Какова (с точностью до ближайшего целого) величина IDF для слова, встречающегося в (*а*) 40 документах, (*б*) 10 000 документах?

**Упражнение 1.3.2.** Допустим, что имеется репозиторий, содержащий 10 миллионов документов, и слово $w$ встречается в 320 из них. В некотором документе $d$ максимальное количество одного вхождений слова равно 15. Какова приблизительно оценка TF.IDF для $w$, если это слово встречается (*а*) один раз, (*б*) пять раз?

**! Упражнение 1.3.3.** Предположим, что ключи хэширования выбираются из совокупности всех неотрицательных целых чисел, кратных некоторой констан-

те $c$, и что взята хэш-функция $h(x) = x \bmod 15$. Для каких значений $c$ эта хэш-функция годится, т. е. распределяет случайно выбранные ключи хэширования по ячейкам приблизительно равномерно?

**Упражнение 1.3.4.** Дайте аппроксимации следующих чисел в терминах $e$:

$$(а)\ (1.01)^{500};\quad (б)\ (1.05)^{1000};\quad (в)\ (0.9)^{40}.$$

**Упражнение 1.3.5.** С помощью разложения $e^x$ в ряд Тейлора вычислите следующие числа с точностью до трех знаков после запятой: (*а*) $e^{1/10}$; (*б*) $e^{-1/10}$; (*в*) $e^2$.

# 1.4. План книги

В этом разделе мы коротко расскажем, о чем пойдет речь в последующих главах. Глава 2 не относится к добыче данных как таковой. В ней дается введение в методику MapReduce, позволяющую распараллелить вычисления в облаке (массиве взаимосвязанных процессоров). Есть основания полагать, что облачные вычисления, и MapReduce в частности, станут обычным способом анализа очень больших объемов данных. Через все остальные главы красной нитью проходит исследование возможности применения MapReduce к реализации описываемых алгоритмов.

Глава 3 посвящена поиску похожих объектов. Отправной точкой будет предположение о том, что объекты представлены множествами элементов, и множества считаются похожими, если доля общих элементов велика. Объясняются основополагающие алгоритмы MinHash и хэширование с учетом близости. У этих методов много применений, и зачастую они дают на удивление эффективные решения задач, которые для больших наборов данных поначалу казались нерешаемыми.

В главе 4 мы рассматриваем данные в форме потока. Разница между потоком и базой данных заключается в том, что данные, поступающие из потока, теряются, если не обработать их немедленно. Важные примеры – поток запросов к поисковой системе или поток кликов на популярном веб-сайте. В этой главе мы увидим несколько неожиданных применений хэширования, благодаря которым удается управлять потоковыми данными.

Глава 5 посвящена всего одному приложению: вычислению PageRank. Именно это вычисление позволило Google выделиться из ряда других поисковых систем, и до сих пор оно является неотъемлемой частью механизма, с помощью которого поисковая система узнает, какие страницы хотел бы видеть пользователь. Обобщения PageRank важны также для борьбы со спамом (для чего употребляется эвфемизм «поисковая оптимизация»), и мы рассмотрим самые последние достижения на этом пути.

Далее, в главе 6 содержится введение в модель корзины покупок и канонические задачи, связанные с ассоциативными правилами и поиском частых предметных наборов. В модели корзины покупок данные состоят из большой коллекции корзин, в каждой из которых находится небольшой набор предметов. Мы опишем

несколько алгоритмов, способных найти все частые пары предметов, т. е. такие пары предметов, которые встречаются вместе во многих корзинах. Еще одна серия алгоритмов полезна для эффективного нахождения большинства частых предметных наборов, содержащих больше двух предметов.

В главе 7 изучается задача кластеризации. Мы предполагаем, что имеется набор объектов с метрикой, определяющей, насколько близки или далеки два объекта. Цель состоит в том, чтобы разбить большой набор данных на поднаборы (кластеры), состоящие из объектов, которые расположены близко друг к другу, но далеко от объектов, принадлежащих другим кластерам.

Глава 8 посвящена Интернет-рекламе и сопутствующим вычислительным задачам. Мы вводим понятие онлайнового алгоритма – такого, который должен дать хороший ответ немедленно, а не ждать, пока станет доступен весь набор данных. В этой же главе обсуждается важная идея коэффициента конкурентоспособности; это отношение гарантированного качества онлайнового алгоритма к качеству оптимального алгоритма, который может видеть все данные перед принятием решения. Эти идеи применяются для синтеза хороших алгоритмов, которые принимают участие в торгах за право показа рекламного объявления в ответ на запрос, поступивший поисковой системе.

Глава 9 посвящена рекомендательным системам. Многие веб-приложения рекомендуют пользователям то, что им может понравиться. Один такой пример – задача компании Netflix: придумать эффективный алгоритм, предсказывающий, какие фильмы понравятся пользователю, другой – задача Amazon: предложить пользователю товар, исходя из информации о том, что ему было бы интересно купить. К построению таких систем есть два основных подхода. Мы можем охарактеризовать объекты признаками, например известными актерами, играющими в фильме, и рекомендовать объекты с такими признаками, которые пользователю заведомо нравятся. Или же мы можем выбрать других пользователей, чьи предпочтения похожи на предпочтения данного пользователя, и посмотреть, что им нравится (этот метод называется коллаборативной фильтрацией).

В главе 10 мы изучаем социальные сети и алгоритмы их анализа. Канонический пример социальной сети – граф друзей в Facebook, в котором вершинами являются люди, и две вершины соединены ребром, если соответствующие люди – друзья. Ориентированные графы, например графы читателей в Twitter, тоже можно рассматривать как социальные сети. Типичная задача – найти «сообщества», т. е. небольшие множества узлов, которые связывает необычно много ребер. Для социальных сетей можно решать и общие задачи теории графов: вычисление транзитивного замыкания или диаметра графа. Но они осложняются из-за гигантского размера типичной сети.

В главе 11 рассматривается понижение размерности. Дана очень большая матрица, обычно разреженная. Можно считать, что матрица представляет связи между сущностями двух видов, например оценки, поставленные фильмам зрителями. Интуитивно представляется, что должно существовать небольшое число

концепций – гораздо меньше, чем зрителей или фильмов, – которые объясняют, почему определенным зрителям нравятся определенные фильмы. Мы предложим несколько алгоритмов, позволяющих упростить матрицы путем их разложения в произведение матриц, гораздо меньших по одному из двух измерений. Одна матрица соотносит сущности одного вида с небольшим числом концепций, а другая – концепции с сущностями другого вида. Если все сделано правильно, то произведение меньших матриц будет очень мало отличаться от исходной.

Наконец, в главе 12 обсуждаются алгоритмы машинного обучения на очень больших наборах данных. Здесь рассматриваются перцептроны, метод опорных векторов, нахождение моделей методом градиентного спуска и модели ближайших соседей.

# 1.5. Резюме

- *Добыча данных.* Этим термином называют процесс извлечения полезных моделей из данных. Иногда модель представляет собой обобщенное представление данных, а иногда – набор экстремальных признаков данных.
- *Принцип Бонферрони.* Если мы хотим считать интересным признаком нечто такое, что вполне может часто встречаться в случайных данных, то не можем полагаться на значимость таких признаков. Это наблюдение ограничивает возможность выделения признаков, которые на практике встречаются недостаточно редко.
- *TF.IDF.* Мера, называемая TF.IDF, позволяет выделить из коллекции документов слова, полезные для определения темы каждого документа. Слово имеет высокую оценку TF.IDF в данном документе, если оно встречается в относительно небольшом числе документов (в том числе в этом), но если уж встречается, то обычно много раз.
- *Хэш-функции.* Хэш-функция отображает ключи хэширования некоторого типа на целочисленные номера ячеек. Хорошая хэш-функция распределяет все возможные значения ключей хэширования по ячейкам приблизительно равномерно. Областью определения хэш-функции может быть любой тип.
- *Индексы.* Индекс – это структура данных, которая позволяет эффективно хранить и извлекать записи, если известны значения одного или нескольких полей. Хэширование – один из способов построения индекса.
- *Хранение на диске.* Если данные приходится хранить на диске (во внешней памяти), то время доступа многократно возрастает по сравнению с хранением в оперативной памяти. Если объем данных велик, то важно, чтобы алгоритмы стремились держать необходимые данные в оперативной памяти.
- *Степенные зависимости.* Многие явления описываются зависимостью вида $y = cx^a$ с некоторым показателем степени $a$, часто близким к $-2$.

В качестве примеров можно назвать продажи $x$-й по популярности книги или количество ссылок, ведущих на $x$-ю по популярности страницу.

# 1.6. Список литературы

Работа [7] – ясное введение в основы добычи данных. В работе [2] добыча данных рассматривается главным образом с точки зрения машинного обучения и статистики.

О построении хэш-функций и хэш-таблиц см. [4]. Подробности, касающиеся меры TF.IDF и других вопросов обработки документов, можно найти в [5]. В книге [3] есть много информации об управлении индексами, хэш-таблицами и данными на диске.

Степенные зависимости в контексте веба исследовались в работе [1]. Эффект Матфея впервые описан в [6].

1. A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, J. Weiner «Graph structure in the web», *Computer Networks* 33:1–6, pp. 309–320, 2000.
2. M. M. Gaber, *Scientific Data Mining and Knowledge Discovery – Principles and Foundations*, Springer, New York, 2010.
3. H. Garcia-Molina, J. D. Ullman, J. Widom, *Database Systems: The Complete Book* Second Edition, Prentice-Hall, Upper Saddle River, NJ, 2009.
4. D. E. Knuth, *The Art of Computer Programming* Vol. 3 (*Sorting and Searching*), Second Edition, Addison-Wesley, Upper Saddle River, NJ, 1998.
5. C. P. Manning, P. Raghavan, H. Schütze, *Introduction to Information Retrieval*, Cambridge Univ. Press, 2008.
6. R. K. Merton «The Matthew effect in science», *Science* 159:3810, pp. 56–63, Jan. 5, 1968.
7. P.-N. Tan, M. Steinbach, V. Kumar, *Introduction to Data Mining*, Addison-Wesley, Upper Saddle River, NJ, 2005.

# ГЛАВА 2. MapReduce и новый программный стек

От современных приложений добычи данных, которые часто называют также анализом «больших данных», требуется способность быстро обрабатывать огромные массивы данных. Во многих таких приложениях структура данных регулярна и существует масса возможностей распараллеливания. Приведем два важных примера.

1. Ранжирование веб-страниц по релевантности, для чего требуется многократно выполнять операции умножения матрицы на вектор, причем речь идет о размерностях порядка миллиардов.
2. Поиск «друзей» на сайтах социальных сетей. Речь идет о графах с сотнями миллионов вершин и многими миллиардами ребер.

Для подобных приложений был разработан новый программный стек. Источником параллелизма в таких программных системах является не «суперкомпьютер», а «вычислительный кластер» – большая совокупность стандартного оборудования общего назначения, включая самые обычные процессоры («вычислительные узлы»), соединенные между собой Ethernet-кабелями и недорогими коммутаторами. Программный стек начинается с файловой системы нового вида, которая называется «распределенной файловой системой»; элементарные единицы хранения в ней гораздо больше, чем дисковые блоки в традиционных операционных системах. Распределенная файловая система также обеспечивает репликацию данных или резервирование для защиты от частых сбоев носителей, неизбежных, когда данные распределены по тысячам дешевых вычислительных узлов.

Поверх таких файловых систем разработано много различных систем программирования более высокого уровня. И главной частью нового программного стека является система *MapReduce*. Реализации MapReduce позволяют эффективно производить типичные операции обработки больших данных в вычислительных кластерах, не опасаясь отказов оборудования.

Системы MapReduce быстро развиваются и расширяются. Сегодня стали обыденностью системы программирования еще более высокого уровня, позволяющие создавать MapReduce-программы, в том числе с применением той или иной реа-

лизации языка SQL. Но, как выясняется, технология MapReduce – полезный, но простой частный случай более общих и перспективных идей. Мы включили в эту главу обсуждение обобщений MapReduce, сначала на системы, поддерживающие ацикличные рабочие процессы, а затем на системы, в которых реализованы рекурсивные алгоритмы.

Последней темой этой главы станет проектирование хороших алгоритмов MapReduce. Зачастую это сильно отличается от проектирования хороших параллельных алгоритмов, предназначенных для работы на суперкомпьютере. При проектировании алгоритмов MapReduce нередко обнаруживается, что эффективность обусловлена, прежде всего, коммуникационной стоимостью. Мы изучим эту стоимость и поймем, как она влияет на выбор наиболее эффективных алгоритмов MapReduce. Для нескольких типичных приложений MapReduce мы сможем предложить семейства алгоритмов с оптимальным компромиссом между стоимостью коммуникаций и уровнем параллелизма.

# 2.1. Распределенные файловые системы

Как правило, вычисления производятся на одном процессоре со своей оперативной памятью, кэшем и локальным диском (*вычислительный узел*). В прошлом приложения, которым требовалась параллельная обработка, например крупные научные программы, исполнялись на специализированных параллельных компьютерах, оснащенных большим числом процессоров и нестандартным оборудованием. Но широкое распространение крупномасштабных веб-служб привело к тому, что все больше и больше вычислений выполняются в центрах с тысячами вычислительных узлов, работающих более-менее независимо. В таких центрах вычислительные узлы представляют собой стандартное оборудование, которое гораздо дешевле специализированных параллельных суперкомпьютеров.

Новые вычислительные средства дали толчок развитию нового поколения программных систем. В этих системах широко используются возможности распараллеливания, но в то же время решены проблемы надежности, возникающие, когда вычислительное оборудование состоит из тысяч независимых компонентов, каждый из которых в любой момент может отказать. В этом разделе мы обсудим характеристики таких вычислительных центров и специализированных файловых систем, разработанных с учетом их возможностей.

## 2.1.1. Физическая организация вычислительных узлов

Новая архитектура параллельных вычислений, которую иногда называют *кластерными вычислениями*, устроена следующим образом. Вычислительные узлы собираются в *стойки*, от 8 до 64 узлов на одну стойку. Узлы, находящиеся в одной стойке, соединены сетью, обычно гигабитной сетью Ethernet. Стоек может быть

много, и они соединены между собой сетью другого уровня или коммутатором. Полоса пропускания межстоечной сети несколько шире, чем внутристоечной, но с учетом количества пар узлов, которым может потребоваться межстоечная связь, ширина этой полосы может оказаться существенной. На рис. 2.1 показана архитектура крупномасштабной вычислительной системы. Конечно, и стоек, и вычислительных узлов в стойке может быть гораздо больше.

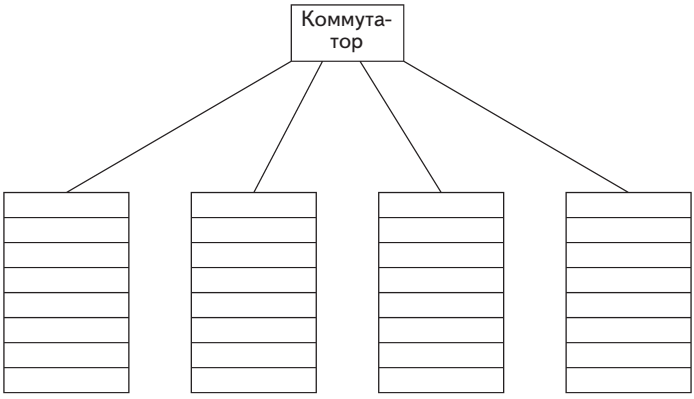



**Рис. 2.1.** Вычислительные узлы собраны в стойки, а стойки связаны между собой коммутатором

Компоненты отказывают – от этого факта никуда не деться. И чем больше в системе компонентов, например вычислительных узлов и межсоединений, тем чаще какой-то компонент выходит из строя. Для такой системы, как на рис. 2.1, основными видами отказов являются утрата одного узла (например, из-за поломки диска) или утрата целой стойки (например, когда внутристоечная сеть работает, но сеть, связывающая стойку с внешним миром, вышла из строя).

Иногда важное вычисление, выполняемое на тысячах узлов, продолжается несколько минут или даже часов. Если бы нам приходилось прерывать и заново запускать такое вычисление после отказа каждого компонента, то оно никогда бы не завершилось. К решению этой проблемы следует подходить с двух сторон.

1. Файлы необходимо хранить с резервированием. Если не продублировать файл на нескольких узлах, то при выходе узла из строя все хранившиеся на нем файлы окажутся недоступны, пока узел не будет заменен. Если не было сделано резервных копий файлов, а диск сломался, то эти файлы пропадут навсегда. Управление файлами мы обсудим в разделе 2.1.2.
2. Вычисление следует разбить на задачи, так чтобы любую задачу, которая не смогла дойти до конца, можно было перезапустить, не затрагивая при этом другие задачи. Такой стратегии придерживается система программирования MapReduce, с которой мы познакомимся в разделе 2.2.

### 2.1.2. Организация больших файловых систем

В кластерных вычислениях файлы должны выглядеть и вести себя иначе, чем в традиционных файловых системах на автономных компьютерах. Новая файловая система, которую часто называют *распределенной файловой системой*, или *DFS* (хотя в прошлом этот термин имел другой смысл), обычно используется следующим образом.

- Размер файлов может быть очень большим, достигая даже терабайтов. Для небольших файлов использовать DFS не имеет смысла.
- Файлы редко обновляются. Как правило, из них читаются данные в ходе некоторого вычисления и, возможно, время от времени добавляются новые данные в конец. Например, система бронирования авиабилетов не подходит для DFS, даже если объем данных очень велик, потому что данные часто изменяются.

Файлы разбиваются на *порции*, обычно размером 64 мегабайта. Порции реплицируются, например, на три разных вычислительных узла. Узлы, содержащие копии одной порции, следует размещать в разных стойках, чтобы не потерять все копии в случае отказа стойки. Размер порции и коэффициент репликации обычно задает пользователь.

Для поиска всех порций файла существует еще один небольшой файл, называемый *главным узлом*, или *узлом имен* для данного файла. Сам главный узел тоже реплицируется, а каталог файловой системы в целом знает, где находятся его копии. Каталог также можно реплицировать, и все клиенты DFS, знают, где найти копии каталога.

**Реализации DFS**

На практике встречается несколько распределенных файловых систем описанного выше типа, в том числе:

1. *Файловая система Google* (*GFS*), родоначальница всего класса.
2. *Распределенная файловая система Hadoop* (*HDFS*), DFS с открытым исходным кодом, используемая совместно с Hadoop – одной из реализаций технологии MapReduce (см. раздел 2.2). Распространяется фондом Apache Software Foundation.
3. *CloudStore*, DFS с открытым исходным кодом, первоначально разработанная компанией Kosmix.

## 2.2. MapReduce

*MapReduce* – это технология вычислений, реализованная в нескольких системах, в том числе в Google (где она называется просто MapReduce) и в популярной си-

стеме с открытым исходным кодом Hadoop, которую можно получить вместе с файловой системой HDFS от фонда Apache Foundation. Реализацию MapReduce можно использовать для отказоустойчивого управления различными крупномасштабными вычислениями. Вам нужно написать всего две функции, *Map* и *Reduce*, а система управляет параллельным выполнением и координацией задач, исполняющих Map и Reduce, учитывая возможность сбоя любой задачи. Вкратце, процесс вычислений в MapReduce выглядит так:

1. Есть несколько задач Map (распределителей), каждая из которых получает одну или несколько порций файла из распределенной файловой системы. Распределители преобразуют порцию в последовательность пар *ключ-значение*. Как именно из входных данных порождаются эти пары, определяет функция Map, написанная пользователем.
2. Пары ключ-значение от каждого распределителя собираются *главным контроллером* и сортируются по ключу. Затем ключи раздаются задачам Reduce (редукторам), так что все пары с одинаковым ключом попадают одному и тому же редуктору.
3. Редукторы обрабатывают по одному ключу за раз и каким-то образом комбинируют значения, ассоциированные с этим ключом. Способ комбинирования определяется функцией Reduce, написанной пользователем.

Эта схема вычислений показана на рис. 2.2.

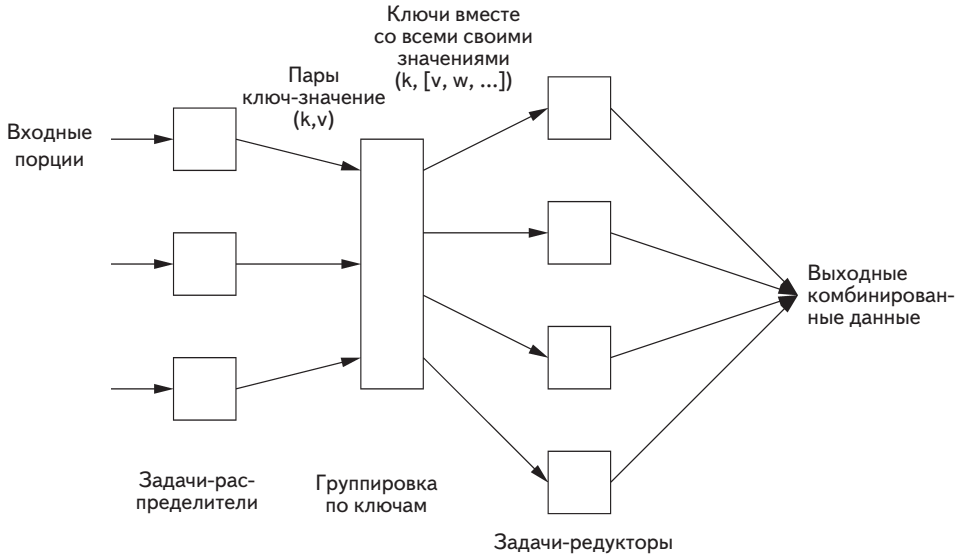



**Рис. 2.2.** Схема вычислений по технологии MapReduce

### 2.2.1. Задачи-распределители

Входные файлы для распределителя рассматриваются как собрания *элементов* произвольного типа, например, кортеж или документ. Порция – это коллекция элементов, и никакой элемент не может входить в две порции. Технически все входные данные распределителей и результаты редукторов представлены в виде пар ключ-значение, но обычно ключи входных элементов несущественны и игнорируются. Требование представлять входные и выходные данные именно в таком виде вызвано желанием поддержать композицию нескольких процессов MapReduce.

Функция Map принимает в качестве аргумента один входной элемент и порождает нуль или более пар ключ-значение. Типы ключей и значений могут быть произвольными. Кроме того, ключи не являются «ключами» в общепринятом смысле: они не обязаны быть уникальными. Распределитель вполне может породить несколько пар ключ-значение с одинаковым ключом – и даже из одного элемента.

**Пример 2.1.** Проиллюстрируем MapReduce-вычисление на уже ставшем стандартным примере: подсчет количества вхождений каждого слова в коллекцию документов. В данном случае входной файл – это репозиторий документов, а каждый документ является элементом. Функция Map получает ключи типа String (слова) и целочисленные значения. Распределитель читает документ и разбивает его на последовательность слов $w_1, w_2, \ldots, w_n$. Затем он порождает последовательность пар ключ-значение, в которых значение всегда равно 1. Таким образом, на выходе распределителя для данного документа получается последовательность пар

$$(w_1, 1), (w_2, 1), \ldots, (w_n, 1)$$

Отметим, что обычно один распределитель обрабатывает много документов – все документы из одной или нескольких порций. Следовательно, на выходе он порождает не только последовательность для одного документа. Заметим также, что если слово $w$ встречается $m$ раз во всех документах, поступивших данному процессу, то на его выходе пара $(w, 1)$ будет встречаться $m$ раз. Можно было бы (мы обсудим это в разделе 2.2.4) объединить эти $m$ пар в одну пару $(w, m)$, но это допустимо лишь потому, что, как мы увидим ниже, редукторы применяют к значениям ассоциативную и коммутативную операцию сложения.

### 2.2.2. Группировка по ключу

После того как все задачи-распределители успешно завершились, пары ключ-значение группируются по ключу, и значения, ассоциированные с каждым ключом, собираются в список. Эта группировка выполняется системой и не зависит от того, что делают функции Map и Reduce. Процесс главного контроллера знает, сколько запущено задач-редукторов, допустим, $r$. Обычно значение $r$ задает пользователь. Располагая этой информацией, главный контроллер выбирает хэш-

функцию, которая применяется к ключам и возвращает номер ячейки в диапазоне от 0 до $r - 1$. Каждый ключ, порожденный распределителем, хэшируется, а соответствующая ему пара ключ-значение записывается в один из $r$ локальных файлов. Каждый файл передается одной задаче-редуктору[1].

Чтобы произвести группировку по ключам и раздать их редукторам, главный контроллер объединяет файлы, созданные распределителями и предназначенные конкретному редуктору, после чего подает результирующий файл на вход процессу в виде последовательности пар ключ-список значений. То есть для каждого ключа $k$ на вход редуктору, обрабатывающему этот ключ, подается пара вида $(k, [v_1, v_2, \ldots, v_n])$, где $(k, v_1)$, $(k, v_2)$, ... , $(k, v_n)$ – все пары с ключом $k$, порожденные всеми распределителями.

## 2.2.3. Задачи-редукторы

Аргументом функции Reduce является пара, состоящая из ключа и списка ассоциированных с ним значений. На выходе функция Reduce возвращает последовательность из нуля или более пар ключ-значение. Типы этих ключей и значений могут отличаться от тех, что распределители подготовили для редукторов, хотя зачастую они совпадают. Мы будем называть *редукцией* акт применения функции Reduce к одному ключу и списку ассоциированных с ним значений.

Задача-редуктор получает один или более ключей и соответствующих списков значений, а, значит, выполняет одну или несколько редукцией. Результаты всех редукторов объединяются в один файл. Редукцию можно поручить меньшему числу задач-редукторов, если хэшировать ключи и ассоциировать по одному редуктору с каждой ячейкой хэш-таблицы.

> **Пример 2.2.** Продолжим пример с подсчетом слов. Функция Reduce просто складывает все значения. Результатом редукции является слово и сумма. Следовательно, результатом всех редукторов является последовательность пар $(w,m)$, где $w$ – слово, которое встречается хотя бы один раз во входных документах, а $m$ – общее число вхождений $w$ во все документы.

## 2.2.4. Комбинаторы

Иногда функция Reduce является коммутативной и ассоциативной. В таком случае значения можно комбинировать в любом порядке, результат от этого не зависит. Сложение в примере 2.2 – пример коммутативной и ассоциативной операции. Как бы ни группировать список чисел $v_1, v_2, \ldots, v_n$, сумма будет одинакова.

Если функция Reduce коммутативна и ассоциативна, то часть работы редукторов можно перенести в распределители. Например, распределители из приме-

[1] При желании пользователь может написать свою хэш-функцию или применить другой способ назначения ключей редукторам. Но в любом случае каждый ключ должен быть назначен одному и только одному редуктору.

ра 2.1 вместо порождения многочисленных пар ($w$, 1), ($w$, 1), ... могли бы применить функцию Reduce внутри себя и только потом отдавать результат для группировки и агрегирования. Тогда все такие пары ключ-значение были бы заменены одной парой с ключом $w$ и значением, равным сумме всех единиц в исходных парах. Следовательно, пары с ключом $w$, порождаемые одной задачей-распределителем, были бы заменены парой ($w$,$m$), где $m$ – количество вхождений слова $w$ во все документы, обработанные данным распределителем. Отметим, что группировать, агрегировать и передавать результат редукторам все равно нужно, потому что, как правило, каждый распределитель выведет одну пару с данным ключом $w$.

**Редукторы, вычислительные узлы и асимметрия**

Чтобы достичь максимального уровня параллелизма, мы могли бы завести отдельную задачу-редуктор для каждой операции редукции, т. е. одного ключа и ассоциированного с ним списка значений. Кроме того, мы могли бы выполнять каждый редуктор на отдельном вычислительном узле, тогда все они будут работать параллельно. Но обычно этот план не оптимален. Во-первых, с каждой создаваемой задачей сопряжены накладные расходы, поэтому лучше, чтобы количество редукторов было меньше количества различных ключей. Кроме того, количество ключей зачастую гораздо больше количества имеющихся узлов, так что создание гигантского числа редукторов не даст никакого выигрыша.

Во-вторых, списки значений для разных ключей часто сильно различаются по длине, так что разные операции редукции будут занимать разное время. Если поручить каждую операцию редукции отдельной задаче-редуктору, то сами эти задачи будут демонстрировать *асимметрию* – значительный разброс времени выполнения. Эффект асимметрии можно уменьшить, если использовать меньше редукторов, чем имеется операций редукции. Если ключи распределяются между редукторами равномерно, то можно ожидать некоторого усреднения общего времени, затрачиваемого каждым редуктором. Асимметрию можно еще уменьшить, если создать больше редукторов, чем имеется вычислительных узлов. Тогда долгая задача-редуктор будет полностью занимать узел, а более короткие смогут последовательно выполняться на одном узле.

## 2.2.5. Детали выполнения MapReduce

Рассмотрим подробнее, как выполняется программа, написанная по технологии MapReduce. На рис. 2.3 схематически представлено взаимодействие процессов, задач и файлов. Пользуясь библиотекой, предоставляемой системой MapReduce, например Hadoop, программа пользователя запускает процесс главного контроллера (мастер) и несколько *рабочих процессов* на разных вычислительных узлах. Обычно рабочий процесс исполняет либо задачи-распределители, либо задачи-редукторы, но не те и другие одновременно.

У мастера много обязанностей. Одна из них – создавать распределители и редукторы, причем количество тех и других определяется пользовательской программой. Мастер назначает этим задачам рабочие процессы. Имеет смысл создавать по одному распределителю на каждую порцию входного файла (или файлов), но количество редукторов обычно меньше. Причина такого ограничения в том, что каждый распределитель создает по одному промежуточному файлу для каждого редуктора, и если редукторов слишком много, то и промежуточные файлы размножаются в неимоверном количестве.

Мастер отслеживает состояние каждого распределителя и редуктора (простаивает, выполняется конкретным рабочим процессом или завершен). Рабочий процесс уведомляет мастера о завершении задачи, после чего мастер назначает ему новую задачу.

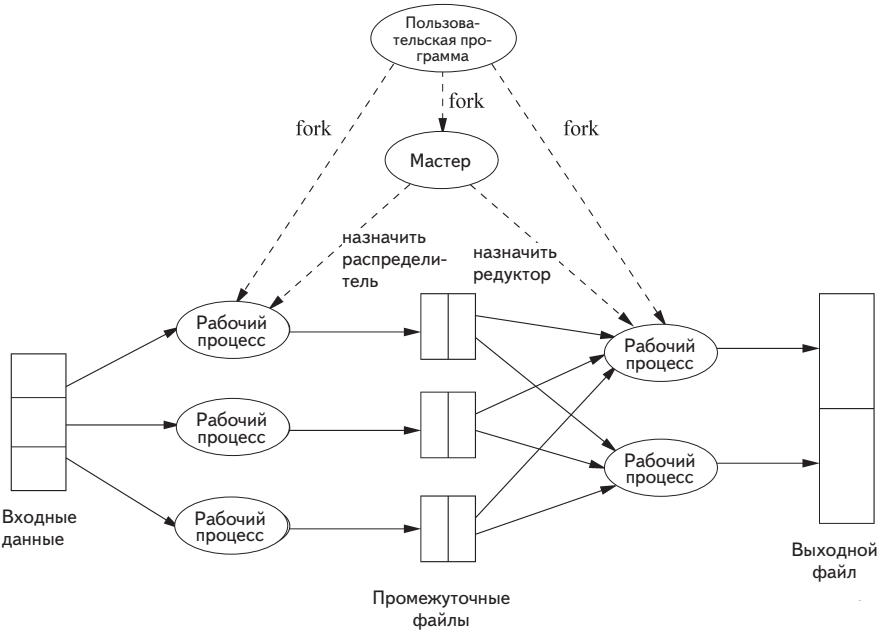



**Рис. 2.3.** Схема выполнения MapReduce-программы

Каждому распределителю назначается одна или несколько порций входного файла (или файлов), после чего в нем исполняется код, написанный пользователем. Распределитель создает по одному файлу для каждого редуктора на локальном диске рабочего процесса, исполняющего данный распределитель. Мастер получает уведомления о месте нахождения и размере каждого такого файла и о редукторе, которому он предназначен. После того как мастер назначит задаче-редуктору рабочий процесс, этой задаче передаются все созданные для нее файлы. Редуктор выполняет написанный пользователем код и выводит результат в файл, находящийся в окружающей распределенной файловой системе.

### 2.2.6. Обработка отказов узлов

Худшее, что может произойти, – отказ вычислительного узла, на котором работает мастер. В таком случае придется перезапустить все приложение MapReduce. Но это единственный узел, который может «уложить» весь процесс; все прочие отказы обрабатываются мастером, так что приложение рано или поздно завершится.

Предположим, что отказывает узел, на котором находится рабочий процесс распределителя. Этот отказ будет замечен мастером, потому что тот периодически прозванивает рабочие процессы. Все задачи-распределители, назначенные этому рабочему процессу, придется выполнить заново, даже если они уже завершились. Завершенные задачи нужно перезапускать, потому что их результаты, предназначенные редукторам, находятся на том же узле и, стало быть, недоступны редукторам. Мастер сбрасывает состояние каждого такого распределителя в «простаивает» и назначает их первому освободившемуся рабочему процессу. Мастер также должен проинформировать каждый редуктор об изменении местоположения входных файлов от этого распределителя.

Обработать отказ узла редуктора проще. Мастер просто устанавливает состояние задач-редукторов, выполнявшихся на этом узле, в «простаивает». Впоследствии им будет назначен другой рабочий процесс.

### 2.2.7. Упражнения к разделу 2.2

**Упражнение 2.2.1.** Допустим, что мы выполняем описанную в этом разделе MapReduce-программу подсчета слов для большого репозитория, например копии всего веба. Будем использовать 100 распределителей и сколько-то редукторов.

(*а*) Предположим, что комбинаторы в распределителях не используются. Следует ли ожидать значительного разброса времени обработки списков значений различными редукторами? Объясните свой ответ.

(*б*) Если поручить операции редукции небольшому числу редукторов, скажем 10, то следует ли ожидать значительного разброса? А что, если вместо этого использовать 10 000 редукторов?

**!** (*с*) Предположим, что есть 100 распределителей и используется комбинатор. Следует ли ожидать значительного разброса? Объясните свой ответ.

## 2.3. Алгоритмы, в которых используется MapReduce

Технология MapReduce – не панацея для решения всех задач, она пригодна даже не для каждой задачи, которая могла бы получить выигрыш от распараллеливания на многих вычислительных узлах. Как было отмечено в разделе 2.1.2, само окружение в виде распределенной файловой системы имеет смысл, только если файлы очень велики и редко изменяются. Поэтому не следует ожидать, что DFS

или какая-то реализация MapReduce помогут управлять розничной торговлей через Интернет, хотя такой гигант, как Amazon.com и задействует тысячи узлов для обработки веб-запросов. Причина в том, что в Amazon основные операции с данными – это поиск товаров, регистрация продаж и т. д., то есть процессы, в которых объем вычислений сравнительно невелик, но зато нужно обновлять базу данных[2]. С другой стороны, Amazon мог бы использовать MapReduce для выполнения некоторых аналитических запросов к большим данным, например, чтобы для каждого пользователя найти, какие пользователи покупают примерно то же, что и он.

Исходная задача, для решения которой в Google была создана реализация MapReduce, заключалась в выполнении умножения очень больших матриц на векторы, что необходимо для вычисления PageRank (см. главу 5). Мы увидим, что умножение матрицы на вектор и перемножение двух матриц – операции, которые прекрасно укладываются в схему MapReduce. Другой важный класс – операции реляционной алгебры, и мы рассмотрим, как их выполнить с помощью MapReduce.

## 2.3.1. Умножение матрицы на вектор с применением MapReduce

Пусть имеется матрица $M$ размерности $n \times n$. Обозначим $m_{ij}$ элемент на пересечении строки $i$ и столбца $j$. Пусть имеется также вектор $\mathbf{v}$ длины $n$, обозначим $v_j$ его $j$-ый элемент. Тогда произведением матрицы $M$ на вектор $\mathbf{v}$ будет вектор $x$ длины $n$, $i$-ый элемент которого $x_i$ определяется по формуле

$$x_i = \sum_{j=1}^{n} m_{ij} v_j .$$

Если $n = 100$, то для этого вычисления не нужны ни DFS, ни MapReduce. Но такого рода вычисления составляют основу ранжирования веб-страниц, которое производят все поисковые системы, а это значит, что порядок $n$ – десятки миллиардов[3]. Предположим сначала, что $n$ велико, но не настолько, чтобы вектор $\mathbf{v}$ не мог уместиться в оперативной памяти. В таком случае этот вектор будет доступен всем распределителям.

Матрица $M$ и вектор $\mathbf{v}$ будут храниться в файлах в DFS. Предположим, что координаты каждого элемента матрицы – строка и столбец – можно определить: либо по его позиции в файле, либо потому что они явно хранятся в виде тройки $(i, j, m_{ij})$. Предположим еще, что позицию элемента $v_j$ в векторе можно определить аналогичным способом.

- **Функция Map.** Функция Map применяется к одному элементу $M$. Однако если $\mathbf{v}$ еще не находится в оперативной памяти узла, на котором выполня-

[2] Напомним, что даже если вы просто смотрите, ничего не покупая, Amazon запоминает, какими товарами вы интересовались.

[3] В этом случае матрица разрежена, в среднем в каждой строке от 10 до 15 ненулевых элементов, поскольку она представляет ссылки в вебе, и элемент $m_{ij}$ отличен от нуля, если существует ссылка со страницы $j$ на страницу $i$. Отметим, что не существует способа сохранить плотную матрицу размерности $10^{10} \times 10^{10}$, потому что в ней было бы $10^{20}$ элементов.

ется распределитель, то он сначала считывается – полностью – а затем становится доступен всем клиентам функции Map, выполняемой этим распределителем. Каждый распределитель работает с порцией матрицы $M$. Для каждого элементы матрицы $m_{ij}$ он порождает пару $(i, m_{ij}v_j)$. Таким образом, все члены суммы, составляющей компонент $x_i$ произведения матрицы на вектор, получают один и тот же ключ $i$.

- **Функция Reduce.** Функция Reduce просто складывает все значения, ассоциированные с данным ключом $i$. В результате получается пара $(i, x_i)$.

## 2.3.2. Если вектор v не помещается в оперативной памяти

Но может случиться, что вектор **v** настолько велик, что не помещается целиком в оперативной памяти. Не требуется, чтобы весь вектор **v** находился в оперативной памяти узла, но в таком случае потребуется очень много операций доступа к диску для перемещения участков вектора в память, чтобы их можно было умножить на элементы матрицы. Можно подойти к задаче по-другому: разбить матрицу на вертикальные *полосы* равной ширины, а вектор на такое же число горизонтальных полос одинаковой высоты. Наша цель – выбрать такое число полос, чтобы часть вектора, высекаемая каждой полосой, помещалась в оперативной памяти узла. На рис. 2.4 показано, как выглядит такое разбиение, если матрица и вектор разбиты на пять полос.

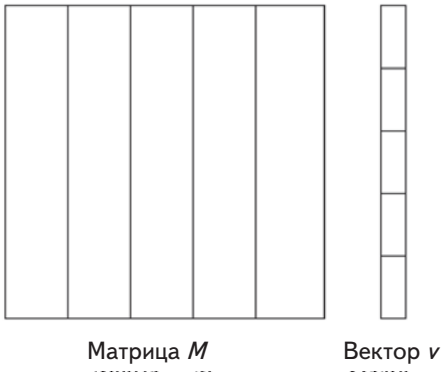



**Рис. 2.4.** Разбиение матрицы и вектора на пять полос

$i$-ая полоса матрица умножается только на компоненты из $i$-ой полосы вектора. Следовательно, мы можем поместить в отдельный файл одну полосу матрицы и так же поступить с вектором. Каждому распределителю назначается порция из какой-то одной полосы матрицы, и он получает соответствующую полосу вектора. После этого распределители и редукторы могут работать точно так же, как если бы распределитель получал целый вектор.

Мы еще вернемся к умножению матрицы на вектор в разделе 5.2. Но там из-за требований конкретного приложения (вычисления PageRank) будет дополни-

тельное ограничение: результирующий вектор должен быть разбит на полосы точно так же, как входной, чтобы результат можно было подать на вход следующей итерации умножения. Мы увидим, что оптимально разбивать матрицу $M$ не на полосы, а на квадратные блоки.

## 2.3.3. Операции реляционной алгебры

Над большими данными существует ряд операций, применяемых в запросах к базе данных. Во многих традиционных приложениях баз данных выбирается относительно немного данных, хотя вся база может быть очень велика. Например, можно запросить баланс одного счета в банке. Для таких запросов применять MapReduce бессмысленно.

Но есть много операций, которые легко описать в терминах типичных примитивов запроса, даже если сами запросы не выполняются в системе управления базами данных. Поэтому неплохой отправной точкой для изучения приложений MapReduce станет рассмотрение стандартных операций над отношениями. Предполагается, что вы знакомы с системами баз данных, языком запросов SQL и реляционной моделью, но на всякий случай напомним, что *отношение* – это таблица с заголовками столбцов, которые называются *атрибутами*. Строки отношения называются кортежами, а множество атрибутов отношения – его *схемой*. Часто отношение с именем $R$ и атрибутами $A_1, A_2, ..., A_n$ записывают в виде $R(A_1, A_2, ..., A_n)$.

**Пример 2.3.** На рис. 2.5 показана часть отношения *Links*, описывающего структуру веба. У него два атрибута: *From* и *To*. Строка, или кортеж отношения – это пара URL, такая что с первого URL ведет хотя бы одна ссылка на второй. Например, пара (*url1*, *url2*) в первой строке на рис. 2.5 означает, что со страницы *url1* ведет ссылка на страницу *url2*. Мы показали только четыре кортежа, но настоящее отношение для веба, или та его часть, которая хранится в типичной поисковой системе, содержит миллиарды кортежей.

| *From* | *To* |
|---|---|
| url1 | url2 |
| url1 | url3 |
| url2 | url3 |
| url2 | url4 |
| ... | ... |

**Рис. 2.5.** Отношение *Links* состоит из множества пар URL-адресов, таких, что с первого ведет хотя бы одна ссылка на второй

Отношение, пусть даже очень большое, можно сохранить в файле в распределенной файловой системе. Элементами этого файла будут кортежи отношения.

Существует несколько стандартных операций над отношениями (в совокупности они называются *реляционной алгеброй*), которые используются при реализации запросов. Сами запросы обычно записываются на языке SQL. Мы обсудим следующие операции реляционной алгебры.

1. *Выборка*. Применить условие $C$ к каждому кортежу отношения и вывести только те, которые удовлетворяют $C$. Результат выборки обозначается $\sigma_C(R)$.
2. *Проекция*. Для некоторого подмножества $S$ атрибутов отношения вывести из каждого кортежа только компоненты, соответствующие атрибутам из $S$. Результат проекции обозначается $\pi_S(R)$.
3. *Объединение*, *пересечение* и *разность*. Эти хорошо известные теоретико-множественные операции применяются к множествам кортежей двух отношений, имеющих одинаковую схему. В SQL существуют также варианты операций над множествами с дубликатами (мультимножествами), определения которых интуитивно не вполне очевидны, но мы не будем вдаваться в эти детали.
4. *Естественное соединение*. Пусть даны два отношения. Сравним все пары кортежей, по одному из каждого отношения. Если кортежи согласованы по всем атрибутам, общим для обеих схем, то выведем кортеж, который содержит компоненты, соответствующим всем атрибутам обеих схем, не повторяя общие атрибуты. Если кортежи не согласованы хотя бы по одному общему атрибуту, не выводим для такой пары кортежей ничего. Естественное соединение отношений $R$ и $S$ обозначается $R \bowtie S$. Мы будем обсуждать только естественное соединение с помощью MapReduce, но *эквисоединения* (соединения, в которых условие согласованности кортежей подразумевает сравнение на равенство атрибутов из двух отношений, не обязательно имеющих одинаковые имена) можно выполнить точно так же. Мы проиллюстрируем это в примере 2.4.
5. *Группировка и агрегирование*[4]. Пусть дано отношение $R$. Разобьем его кортежи на группы в соответствии со значениями в некотором наборе атрибутов $G$, называемых *группировочными атрибутами*. Затем для каждой группы агрегируем значения некоторых других атрибутов. Обычно допускаются операции агрегирования SUM, COUNT, AVG, MIN и MAX с очевидной семантикой. Отметим, что для выполнения MIN и MAX необходимо, чтобы тип агрегируемых атрибутов допускал сравнение (например, строки и числа), а для выполнения SUM и AVG тип должен допускать арифметические операции. Операция группировки и агрегирования на отношении $R$ обозначается $\gamma_X(R)$, где $X$ – список элементов, каждый из которых является либо

   (*а*) группировочным атрибутом, либо
   (*б*) выражением $\theta(A)$, где $\theta$ – одна из пяти операций агрегирования, например SUM, а $A$ – атрибут, не совпадающий ни с одним из группировочных.

   Результатом этой операции – является множество, содержащее один кортеж для каждой группы. В этом кортеже имеется по одному компоненту

[4] В некоторых описаниях реляционной алгебры эти операции не включаются, и на самом деле они не входили в оригинальное определение этой алгебры. Однако они настолько важны в SQL, что в современные трактовки реляционной алгебры включены.

для каждого группировочного атрибута со значением, общим для всех кортежей данной группы. Кроме того, в нем есть по одному компоненту для каждого агрегата со значением, полученным путем агрегирования по данной группе. Иллюстрация будет приведена в примере 2.5.

**Пример 2.4.** Попробуем найти пути длины 2 в вебе, воспользовавшись отношением *Links* на рис. 2.5. То есть мы хотим найти тройки URL-адресов ($u$,$v$,$w$) такие, что существует ссылка, ведущая с $u$ на $v$, и ссылка, ведущая с $v$ на $w$. По существу, мы хотим выполнить естественное соединение *Links* с самим собой, но сначала должны притвориться, что это два отношения с разными схемами, так чтобы желаемую связь можно было описать как естественное соединение. Итак, представим, что существует две копии отношения *Links*, $L1(U1,U2)$ и $L2(U2,U3)$. Если теперь вычислить $L1 \bowtie L2$, то мы получим именно то, что нужно. Иначе говоря, для каждого кортежа $t1$ из $L1$ (т. е. каждого кортежа *Links*) и каждого кортежа $t2$ из $L2$ (еще одного кортежа *Links*, возможно, того же самого) смотрим, совпадают ли значения компоненты $U2$. Отметим, что это вторая компонента $t1$ и первая компонента $t2$. Если эти две компоненты равны, то помещаем в результат кортеж со схемой ($U1$,$U2$,$U3$). Этот кортеж включает первую компоненту $t1$, вторую компоненту $t1$ (которая должна быть равна первой компоненте $t2$) и вторую компоненту $t2$.

Возможно, нам нужен не весь путь длины 2, а только пары ($u$,$w$) URL-адресов такие, что существует по меньшей мере один путь из $u$ в $w$ длины 2. В таком случае мы можем спроецировать результирующее отношение, опустив средние компоненты, т. е. вычислить $\pi_{U1,U3}(L1 \bowtie L2)$.

**Пример 2.5.** Предположим, что на сайте социальной сети имеется отношение

Friends(User, Friend).

Оно состоит из пар ($a$, $b$) таких, что $b$ является другом $a$. Требуется получить сводные данные о количестве друзей у всех членов сети. Это можно сделать с помощью группировки и агрегирования, а именно:

$$\gamma_{\text{User,COUNT(Friend)}}(\text{Friends}).$$

Эта операция группирует кортежи по значению первой компоненты, так что для каждого пользователя создается одна группа. Затем для каждой группы вычисляется количество друзей данного пользователя. В результате получится один кортеж для каждой группы вида (Sally, 300) – если у пользователя «Sally» 300 друзей.

## 2.3.4. Вычисление выборки с помощью MapReduce

Для выборки не нужна вся мощь MapReduce. Эту операцию удобнее выполнить целиком на стадии распределения, хотя можно было бы и целиком на стадии редукции. Ниже представлена MapReduce-реализация операции выборки $\sigma_C(R)$.

- **Функция Map.** Для каждого кортежа $t$, принадлежащего $R$, проверить, удовлетворяет ли он условию $C$. Если да, породить пару $(t, t)$, в которой и ключ, и значение равны $t$.
- **Функция Reduce.** Это тождественная функция. Она просто копирует каждую полученную пару в результат.

Отметим, что результат, строго говоря, не является отношением, т. к. содержит пары ключ-значение. Однако из него можно получить отношение, взяв только компоненту значения (или только ключа).

## 2.3.5. Вычисление проекции с помощью MapReduce

Проекция вычисляется аналогично выборке, но поскольку после проецирования результат может содержать одинаковые кортежи, то функция Reduce должна удалить дубликаты. Вычислить $\pi_s(R)$ можно следующим образом.

- **Функция Map.** Для каждого кортежа $t$, принадлежащего $R$, построить кортеж $t'$, исключив из $t$ компоненты, соответствующие атрибутам, которые не входят в $S$. Вывести пару $(t', t')$.
- **Функция Reduce.** Для каждого ключа $t'$, порожденного любым распределителем, может существовать одна или несколько пар $(t', t')$. Функция Reduce преобразует $(t', [t', t', ..., t'])$ в $(t', t')$, т. е. оставляет ровно одну пару $(t', t')$ для ключа $t'$.

Заметим, что операция Reduce сводится к удалению дубликатов. Эта операция ассоциативна и коммутативна, поэтому комбинатор, ассоциированный с каждым распределителем, мог бы удалить локальные дубликаты. Однако редукторы все равно нужны, чтобы удалять одинаковые кортежи, поступившие от разных распределителей.

## 2.3.6. Вычисление объединения, пересечения и разности с помощью MapReduce

Сначала рассмотрим объединение двух отношений. Предположим, что отношения $R$ и $S$ имеют одинаковые схемы. Распределителям будут назначены порции из R либо из $S$ – неважно, откуда именно. Распределитель должен только передать полученные на входе кортежи редукторам в виде пар ключ-значение. А редуктору останется только удалить дубликаты, как для проекции.

- **Функция Map.** Преобразовать каждый входной кортеж $t$ в пару $(t, t)$.
- **Функция Reduce.** С каждым ключом $t$ будет ассоциировано одно или два значения. В любом случае вывести пару $(t, t)$.

Чтобы вычислить пересечение, мы можем воспользоваться той же самой функцией Map. Но функция Reduce должна порождать кортеж, только если он встречается в обоих отношениях. Если с ключом $t$ ассоциирован список из двух значений

$[t, t]$, то редуктор для $t$ должен породить пару $(t, t)$. Если же с ключом ассоциирован список из одного значения $[t]$, то либо в $R$, либо в $S$ кортеж $t$ отсутствует, поэтому включать для него кортеж в пересечение не нужно.

- **Функция Map.** Преобразовать каждый входной кортеж $t$ в пару $(t, t)$.
- **Функция Reduce.** Если с ключом $t$ ассоциирован список $[t, t]$, то породить пару $(t, t)$. Иначе не порождать ничего.

Вычислить разность $R - S$ чуть сложнее. Кортеж $t$ входит в разность, только если он принадлежит $R$, но не принадлежит $S$. Функция Map может передавать кортежи из $R$ и $S$ насквозь, но должна информировать функцию Reduce, откуда именно пришел кортеж. Таким образом, в качестве значения ключа $t$ будет фигурировать отношение. Ниже приведена спецификация обеих функций.

- **Функция Map.** Для каждого кортежа $t$, принадлежащего $R$, породить пару $(t, R)$, а для каждого кортежа $t$, принадлежащего $S$, – пару $(t, S)$. Конечно, мы хотим в качестве значение указывать имя $R$ или $S$ (а еще лучше – один бит, различающий $R$ и $S$), а не все отношение.
- **Функция Reduce.** Если с ключом $t$ ассоциирован список $[R]$, то породить пару $(t, t)$. Иначе не порождать ничего.

## 2.3.7. Вычисление естественного соединения с помощью MapReduce

Понять идею реализации естественного соединения можно, рассмотрев частный случай соединения $R(A,B)$ с $S(B,C)$. Мы должны найти кортежи, согласованные по компонентам $B$, т. е. второй компоненте кортежей из $R$ и первой компоненте кортежей из $S$. В качестве ключа будем использовать значение B из кортежей обоих отношений. А значением будет другая компонента и имя отношения, чтобы функция Reduce знала, откуда поступил каждый кортеж.

- **Функция Map.** Для каждого кортежа $(a, b)$, принадлежащего $R$, породить пару $((b,(R,a))$. Для каждого кортежа $(b, c)$, принадлежащего $S$, породить пару $((b,(S,c))$.
- **Функция Reduce.** С каждым ключом $b$ будет ассоциирован список пар вида $(R, a)$ или $(S, c)$. Построить все пары, состоящие из одной пары с первой компонентой $R$ и другой с первой компонентой $S$, например $(R, a)$ и $(S, c)$. Результатом для данного ключа и списка значений будет последовательность пар ключ-значение. Сам ключ неважен. Каждое значение представляет собой тройку $(a, b, c)$ такую, что $(R, a)$ и $(S, c)$ встречаются во входном списке значений.

Тот же алгоритм работает, если в отношениях больше двух атрибутов. Можно считать, что $A$ представляет все атрибуты, присутствующие в схеме $R$, но отсутствующие в схеме $S$. $B$ представляет атрибуты, присутствующие в обеих схемах, а $C$ – атрибуты, присутствующие только в схеме $S$. Ключом для кортежа из $R$ или

$S$ будет список значений всех атрибутов, присутствующих как в схеме $R$, так и в схеме $S$. Значением для кортежа из $R$ будет имя $R$ вместе со значениями всех атрибутов, принадлежащих $R$, но не $S$, а значением для кортежа из $S$ – имя $S$ вместе со значениями всех атрибутов, принадлежащих $S$, но не $R$.

Функция Reduce просматривает все пары ключ-значение с данным ключом и комбинирует значения из $R$ со значениями из $S$ всеми возможными способами. Для каждой комбинации порождается кортеж, содержащий значения из $R$, значения ключа и значения из $S$.

## 2.3.8. Вычисление группировки и агрегирования с помощью MapReduce

Как и в случае соединения, обсудим минимальный пример группировки и агрегирования, когда есть только один группировочный атрибут и один агрегат. Пусть $R(A, B, C)$ – отношение, к которому мы применяем оператор $\gamma_{A,\theta(B)}(R)$. Распределитель производит группировку, а редуктор – агрегирование.

- **Функция Map.** Для каждого кортежа $(a, b, c)$ породить пару $(a, b)$.
- **Функция Reduce.** Каждый ключ представляет одну группу. Применить оператор агрегирования θ к списку $[b_1, b_2, ..., b_n]$ значений из $B$, ассоциированных с ключом $a$. На выходе породить пару $(a, x)$, где $x$ – результат применения θ к списку. Например, если θ – оператор SUM, то $x = b_1 + b_2 + ... + b_n$, а если θ – оператор MAX, то $x$ – наибольшее из значений $b_1, b_2, ..., b_n$.

Если группировочных атрибутов несколько, то ключом будет список выбранных из кортежа значений всех этих атрибутов. Если агрегатов несколько, то функция Reduce применяет каждый из них к списку значений, ассоциированных с данным ключом, и порождает кортеж, состоящий из ключа, содержащего компоненты для всех группировочных атрибутов, если таковых несколько, за которыми следуют результаты применения каждого агрегата.

## 2.3.9. Умножение матриц

Пусть $M$ – матрица, $m_{ij}$ – элемент $M$ на пересечении строки $i$ и столбца $j$, и пусть $N$ – другая матрица, а $n_{jk}$ – элемент $N$ на пересечении строки $j$ и столбца $k$. Тогда произведением $P = MN$ называется матрица $P$, в которой элемент $p_{ik}$ на пересечении строки $i$ и столбца $k$ вычисляется по формуле

$$p_{ik} = \sum_j m_{ij} n_{jk} .$$

Чтобы сумма по $j$ имела смысл, необходимо, чтобы количество столбцов $M$ было равно количеству строк $N$.

Матрицу можно рассматривать как отношение с тремя атрибутами: номер строки, номер столбца и значение на пересечении этой строки и столбца. Тогда матрицу $M$ можно записать в виде отношения $M(I, J, V)$ с кортежами вида $(i, j, m_{ij})$, а матрицу $N$ – в виде отношения $N(J, K, W)$ с кортежами вида $(j, k, n_{jk})$. Поскольку

большие матрицы часто разрежены (почти все элементы равны 0) и поскольку мы можем опускать кортежи для нулевых элементов матрицы, такое представление очень хорошо подходит для больших матриц. Однако может быть и так, что $i$, $j$ и $k$ неявно определяются позицией элемента матрицы в представляющем ее файле, а не хранятся явно в самом элементе. В таком случае функцию Map нужно будет спроектировать так, чтобы она строила компоненты $I$, $J$, $K$ кортежей по позиции данных.

Произведение $MN$ – это почти естественное соединение, за которым следует группировка и агрегирование. Точнее, естественное соединение отношений $M(I, J, V)$ и $N(J, K, W)$, имеющих единственный общий атрибут $J$, породило бы кортежи $(i, j, k, v, w)$ из каждого кортежа $(i, j, v)$ из $M$ и кортежа $(j, k, w)$ из $N$. Такой кортеж с пятью компонентами представляет пару элементов матрицы $(m_{ij}, n_{jk})$. А нам нужно произведение этих элементов, т. е. четырехкомпонентный кортеж $(i, j, k, v \times w)$, потому что он представляет произведение $m_{ij} n_{jk}$. Получив такое отношение с помощью одной операции MapReduce, мы сможем выполнить группировку и агрегирование, взяв в качестве группировочных атрибутов $I$ и $K$, а в качестве агрегата – сумму $V \times W$. То есть мы можем реализовать умножение матриц в виде каскада из двух операций MapReduce следующим образом. На первом шаге:

- **Функция Map.** Для каждого элемента матрицы $m_{ij}$ породить пару $(j, (M, i, m_{ij}))$. Аналогично для каждого элемента матрицы $n_{jk}$ породить пару $(j, (N, k, n_{jk}))$. Отметим, что $M$ и $N$ в значениях – не сами матрицы, а имена матриц (как мы уже отмечали, говоря об аналогичной функции Map для естественного соединения), а еще лучше – бит, показывающий, из какой матрицы поступил элемент.
- **Функция Reduce.** Для каждого ключа $j$ проанализировать список ассоциированных с ним значений. Для каждого значения, поступившего из $M$, например $(M, i, m_{ij})$, и каждого значения, поступившего из $N$, например $(N, k, n_{jk})$, породить пару с ключом $(i, k)$ и значением, равным произведению этих элементов $m_{ij} n_{jk}$.

Теперь с помощью еще одной операции MapReduce выполним группировку и агрегирование.

- **Функция Map.** Это тождественная функция, т. е. для каждого входного элемента с ключом $(i, k)$ и значением $v$ она порождает точно такую же пару ключ-значение.
- **Функция Reduce.** Для каждого ключа $(i, k)$ породить сумму списка значений, ассоциированного с этим ключом. Результатом является пара $((i, k), v)$, где $v$ – значение элемента на пересечении строки $i$ и столбца $k$ матрицы $P = MN$.

## 2.3.10. Умножение матриц за один шаг MapReduce

Часто существует несколько способов решить задачу с использованием MapReduce. Допустим, нам хотелось бы вычислить произведение матриц $P = MN$ в

один, а не в два прохода[5]. Это возможно, только нужно нагрузить обе функции дополнительной работой. Сначала мы воспользуемся функцией Map, чтобы создать два набора элементов матрицы, необходимых для вычисления каждого элемента $P$. Отметим, что один элемент $M$ или $N$ участвует в вычислении многих элементов результата, поэтому мы преобразуем каждый входной элемент в несколько пар ключ-значение. Ключами будут пары $(i, k)$, где $i$ – номер строки $M$, а $k$ – номер столбца $N$. Ниже приведена спецификация функций Map и Reduce.

- **Функция Map.** Для каждого элемента $m_{ij}$ матрицы $M$ породить все пары $((i, k), (M, j, m_{ij}))$ для $k = 1, 2, ...$, число столбцов $N$. Аналогично для каждого элемента $n_{jk}$ матрицы $N$ породить все пары $((i, k), (N, j, n_{jk}))$ для $i = 1, 2, ...$, число строк $M$. Как и раньше, $M$ и $N$ – на самом деле биты, показывающие, из какого отношения поступило значение.
- **Функция Reduce.** С каждым ключом $(i, k)$ будет ассоциирован список значений $(M, j, m_{ij})$ и $(N, j, n_{jk})$ для всех возможных значений $j$. Функция Reduce должна для всех $j$ соединить пары значений из этого списка с одинаковым значением $j$. Сделать это проще всего, построив два отсортированных по $j$ списка: в первый попадают значения, начинающиеся с $M$, а во второй – значения, начинающиеся с $N$. Затем из $j$-го элемента каждого списка нужно извлечь и перемножить значения $m_{ij}$ и $n_{jk}$. Произведения складываются, полученная сумма объединяется в пару с ключом $(i, k)$ и становится частью результата функции Reduce.

Отметим, что если строка матрицы $M$ или столбец матрицы $N$ не помещаются в оперативную память, то редукторы будут вынуждены воспользоваться внешней сортировкой для упорядочения значений с одним и тем же ключом $(i, k)$. Однако в данном случае сами матрицы настолько велики, порядка $10^{20}$ элементов, что вряд ли мы стали бы пытаться производить это вычисление, если бы они были плотными. А если матрицы разрежены, то можно ожидать, что с каждым ключом ассоциировано не так уж много значений, поэтому удастся просуммировать произведения, не покидая оперативную память.

## 2.3.11. Упражнения к разделу 2.3

**Упражнение 2.3.1.** Спроектируйте алгоритмы MapReduce, которые принимают на входе очень большой файл целых чисел и порождают:

(*а*) наибольшее число;

(*б*) среднее арифметическое всех чисел;

(*в*) набор тех же чисел, из которого исключены дубликаты;

(*г*) количество различных целых чисел во входном наборе.

**Упражнение 2.3.2.** В нашей постановке задачи об умножении матрицы на вектор предполагалось, что матрица $M$ квадратная. Обобщите алгоритм на случай,

[5] Однако в разделе 2.6.7 мы покажем, что обычно умножение матрицы лучше все же выполнять в два прохода MapReduce.

когда $M$ – матрица размерности $r \times c$, где $r$ – количество строк, а $c$ – количество столбцов.

**! Упражнение 2.3.3.** В варианте реляционной алгебры, реализованном в SQL, отношения являются не множествами, а *мультимножествами*, т. е. допускаются одинаковые кортежи. Ниже приведены определения объединения, пересечения и разности для мультимножеств. Напишите алгоритмы MapReduce для вычисления следующих операций над мультимножествами $R$ и $S$:

(*а*) *объединение мультимножеств*, определенное как мультимножество, для которого количество вхождений кортежа $t$ равно сумме двух чисел: количество его вхождений в $R$ и в $S$;

(*б*) *пересечение мультимножеств*, определенное как мультимножество, для которого количество вхождений кортежа $t$ равно минимуму из двух чисел: количество его вхождений в $R$ и в $S$;

(*в*) *разность мультимножеств*, определенная как мультимножество, для которого количество вхождений кортежа $t$ равно разности двух чисел: количества его вхождений в $R$ и в $S$. Если кортеж входит в $S$ больше раз, чем в $R$, то в разность он не включается.

**! Упражнение 2.3.4.** Для мультимножеств определена также операция выборки. Придумайте MapReduce-реализацию, которая порождает правильное количество копий каждого кортежа $t$, удовлетворяющего условию выборки. То есть предложите такой способ организации пар ключ-значение, при котором было бы легко получить из значений правильный результат выборки.

**Упражнение 2.3.5.** Операция реляционной алгебры $R(A, B) \bowtie_{B<C} S(C, D)$ порождает все такие кортежи $(a, b, c, d)$, что $(a, b)$ принадлежит отношению $R$, $(c, d)$ принадлежит отношению $S$ и $b < c$. Предложите MapReduce-реализацию этой операции в предположении, что $R$ и $S$ – множества.

# 2.4. Обобщения MapReduce

Технология MapReduce оказалась настолько привлекательной, что было предложено несколько ее обобщений и модификаций. Такие системы, как правило, имеют ряд общих черт с системами MapReduce.

1. Надстроены над распределенной файловой системой.
2. Управляют очень большим количеством задач, являющихся экземплярами небольшого числа написанных пользователями функций.
3. Включают способ обработки большинства отказов, случающихся во время выполнения большого задания, без необходимости перезапускать его с самого начала.

В этом разделе мы перечислим некоторые интересные направления исследования. Детали упоминаемых систем можно найти в списке литературы к этой главе.

### 2.4.1. Системы потоков работ

Две экспериментальные системы – Clustera, созданная в Висконсинском университете, и Hyracks, созданная в Калифорнийском университете в Ирвайне, – обобщают MapReduce с простого двухшагового потока работ на ациклический граф, представляющий поток работ, передаваемых от одной функции к другой. Иначе говоря, существует ациклический потоковый граф, в котором ребро $a \rightarrow b$ представляет тот факт, что выход функции $a$ является входом функции $b$. На рис. 2.6 показан пример такого графа. Мы видим, что пять функций, от $f$ до $j$, передают данные слева направо таким образом, что поток данных не имеет циклов и никакая задача не должна возвращать результат до того, как получит входные данные. Например, функция $h$ получает входные данные из существующего файла в распределенной файловой системе. Элементы, порождаемые $h$ на выходе, передаются по крайней мере одной из функций $i$ или $j$.

Как и функции Map и Reduce, любая функция потока работ может выполняться несколькими задачами, каждой из которых назначается порция входных данных. За распределение работы между задачами, реализующими одну и ту же функцию, отвечает главный контроллер. Обычно для этого вычисляется хэш-код входного элемента, который определяет, какой задаче этот элемент передать. Все задачи, реализующие функцию $f$, подобно распределителям, порождают выходные файлы, предназначенные функции (или нескольким функциям), следующей за $f$. Главный контроллер доставляет эти файлы в подходящий момент – после того как задача завершит свою работу.

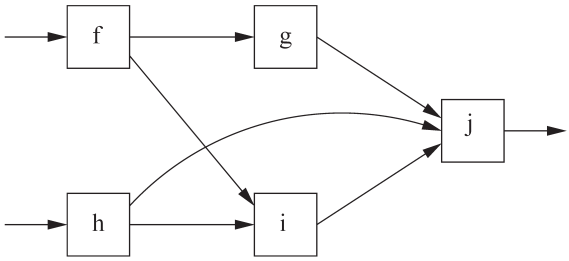



**Рис. 2.6.** Пример потока работ, более сложного, чем распределитель, подающий данные редуктору

Функции потока работ, а, следовательно, и задачи имеют важное общее свойство с задачами MapReduce: результат доставляется следующей задаче только после завершения текущей. Поэтому если задача завершается с ошибкой, ее результаты не будут доставлены следующей за ней в потоковом графе. Главный контроллер может в этом случае перезапустить сбойную задачу на другом узле, не беспокоясь о том, что результат перезапущенной задачи продублирует данные, которые ранее уже были переданы какой-то другой задаче.

Многие приложения для систем потоков работ, в частности Clustera и Hyracks, представляют собой каскады заданий MapReduce. Примером может служить соединение трех отношений, когда одно задание MapReduce соединяет первые два

отношения, а второе задание MapReduce соединяет третье отношение с результатом соединения первых двух. Оба задания могут использовать алгоритм, описанный в разделе 2.3.7.

У реализации таких каскадов в виде одного потока работ есть свои преимущества. Например, главный контроллер может управлять потоком данных между задачами и его репликацией, не храня временный файл, являющийся выходом одного задания MapReduce, в распределенной файловой системе. Размещая задачи на тех вычислительных узлах, где имеется копия их входных данных, мы сможем избежать большого объема коммуникаций, который понадобился бы, если бы мы сначала сохраняли результат одного задания MapReduce, а затем запускали второе (хотя Hadoop и другие системы MapReduce и так пытаются размещать задачи-распределители там, где уже находится копия их входных данных).

## 2.4.2. Рекурсивные обобщения MapReduce

Многие крупномасштабные вычисления имеют рекурсивную природу. Важный пример – вычисление PageRank, рассматриваемое в главе 5. Попросту говоря, оно сводится к нахождению неподвижной точки умножения матрицы на вектор. Вычисление производится в системе MapReduce путем повторного применения алгоритма умножения матрицы на вектор из раздела 2.3.1 или с помощью более сложной стратегии, которую мы опишем в разделе 5.2. Как правило, число итераций заранее неизвестно; каждая итерация является заданием MapReduce, и вычисление продолжается, пока результаты двух последовательных итераций не окажутся достаточно близки. В этом случае мы считаем, что процесс сошелся.

Рекурсия обычно реализуется с помощью повторения заданий MapReduce, потому что истинно рекурсивная задача не обладает свойством, необходимым для независимого перезапуска сбойных задач. Это было бы невозможно для набора взаимно рекурсивных задач, выход каждой из которых подается на вход хотя бы одной другой задаче, а окончательный результат порождается только в конце задачи. Если бы все они следовали этой политике, то ни одна задача не получила бы входные данные, и результат никогда не был бы достигнут. Поэтому в системе с рекурсивными потоками работ (в которых потоковые графы не являются ациклическими) должен существовать какой-то механизм, отличный от простого перезапуска сбойных задач. Начнем с изучения примера рекурсии, реализованной в виде потока работ, а затем обсудим подходы к обработке сбоев задач.

**Пример 2.6.** Пусть имеется ориентированный граф, ребра которого представлены отношением $E(X, Y)$, означающим, что существует ребро из вершины $X$ в вершину $Y$. Мы хотим вычислить отношение путей $P(X, Y)$, интерпретируемое как наличие пути длины 1 или более между вершиной $X$ и вершиной $Y$. Вот как выглядит простой рекурсивный алгоритм решения этой задачи.

1. В начале положить $P(X, Y) = E(X, Y)$.
2. Пока отношение $P$ продолжает изменяться, добавлять в $P$ все кортежи из

$$\pi_{X,Y}(P(X,Z) \bowtie P(Z, Y))$$

Иначе говоря, найти пары вершин $X$ и $Y$ такие, что для некоторой вершины $Z$ заведомо существует путь из $X$ в $Z$ и путь из $Z$ в $Y$.

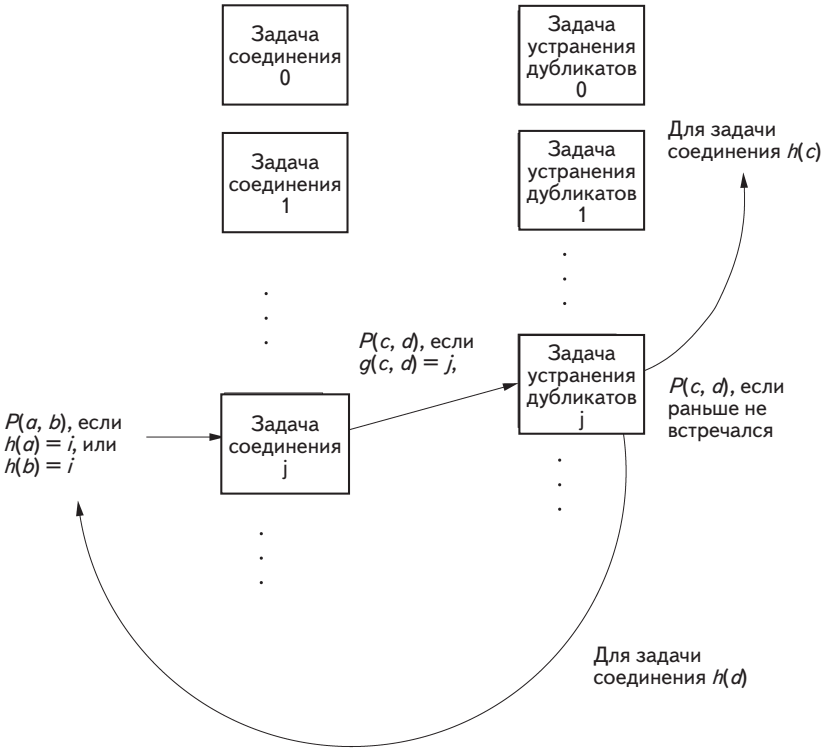



**Рис. 2.7.** Реализация транзитивного замыкания с помощью набора рекурсивных задач

На рис. 2.7 показано, как можно было бы организовать рекурсивные задачи для этого вычисления. Есть два вида задач: *задачи соединения* и *задачи устранения дубликатов*. Существует некоторое число $n$ задач соединения, и каждой из них соответствует ячейка хэш-функции $h$. Обнаруженный кортеж пути $P(a, b)$ подается на вход двум задачам соединения: с номерами $h(a)$ и $h(b)$. Работа $i$-ой задачи соединения, получившей на входе кортеж $P(a, b)$, заключается в том, чтобы найти другие кортежи, которые уже встречались раньше (и сохранены этой задачей на локальном диске).

1. Сохранить $P(a, b)$ в локальном файле.
2. Если $h(a) = i$, то искать кортежи $P(x, a)$ и порождать выходные кортежи $P(x, b)$.
3. Если $h(b) = i$, то искать кортежи $P(b, y)$ и порождать выходные кортежи $P(a, y)$.

Может случиться, что $h(a) = h(b)$, и тогда выполняются оба шага (2) (3). Но обычно для данного кортежа нужно выполнить только один из двух шагов.

Существуют также задачи устранения дубликатов, каждой из которых соответствует ячейка хэш-функции $g$, принимающей два аргумента. Если $P(c, d)$ – выход какой-то задачи соединения, то он передается задаче устранения дубликатов с номером $j = g(c, d)$. Получив этот кортеж, $j$-ая задача устранения дубликатов проверяет, видела ли она его раньше. Если да, то кортеж игнорируется. А новый кортеж сохраняется в локальном файле и отправляется двум задачам соединения с номерами $h(c)$ и $h(d)$.

Каждая задача соединения порождает $m$ выходных файлов – по одному для каждой задачи устранения дубликатов, а каждая задача устранения дубликатов порождает $n$ выходных файлов – по одному для каждой задачи соединения. Существует несколько стратегий распределения этих файлов. Первоначально кортежи $E(a, b)$, представляющие ребра графа, распределяются задачам устранения дубликатов: $E(a, b)$ передается как $P(a, b)$ задаче устранения дубликатов с номером $g(a, b)$. Мастер может подождать, пока все задачи соединения обработают все свои входные данные в этом раунде. Затем все выходные файлы распределяются задачам устранения дубликатов, которые создают собственные выходные файлы. Эти файлы распределяются между задачами соединения и становятся для них входными данными на следующем раунде. Можно поступить и по-другому: каждая задача может ждать, пока объем порожденных ей данных не станет достаточно большим, чтобы оправдать передачу данных потребителю, даже если она сама еще не обработала все свои входные данные.

В примере 2.6 необязательно заводить задачи двух видов. Задачи соединения могли бы устранять дубликаты по мере получения, поскольку они все равно должны сохранять ранее полученные входные данные. Но у такой организации есть преимущество, которое становится очевидным, когда приходится восстанавливаться после сбоя задачи. Если каждая задача хранит все когда-либо созданные ей выходные файлы и если мы поместим задачи соединения и задачи устранения дубликатов на разные стойки, то сможем обработать отказ одного вычислительного узла или одной стойки. Точнее, перезапускаемая задача соединения может получить все необходимые ей файлы, ранее сгенерированные задачами устранения дубликатов, и наоборот.

В частном случае вычисления транзитивного замыкания необязательно предотвращать порождение перезапущенной задачей выходных файлов, которые уже были сгенерированы ранее. Повторное обнаружение пути никак не влияет на окончательный результат. Однако существует много вычислений, для которых недопустима ситуация, когда исходная и перезапущенная задача передают другой задаче одни и те же выходные данные. Например, если последним шагом вычисления является агрегирование, скажем подсчет вершин, достижимых из каждой вершины графа, то, если мы будем учитывать один и тот же путь дважды, то, очевидно, получим неверный ответ. В таком случае главный контроллер может запоминать, какие файлы были сгенерированы каждой задачей и переданы другим задачам. Тогда, перезапустив сбойную задачу, он сможет игнорировать файлы, сгенерированные ей повторно.

**Pregel и Giraph**

Система Pregel, как и MapReduce, первоначально была разработана компанией Google. И, как и в случае MapReduce, фонд Apache распространяет эквивалентную систему Giraph с открытым исходным кодом.

## 2.4.3. Система Pregel

Другой подход к управлению отказами при реализации рекурсивных алгоритмов в вычислительном кластере, предложен в системе Pregel. В ней данные рассматриваются как граф. Каждая вершина графа примерно соответствует задаче (хотя на практике многие вершины большого графа можно объединить в одну задачу, как в случае задач соединения в примере 2.6). Каждая вершина графа генерирует выходные сообщения, предназначенные другим вершинам, и каждая вершина обрабатывает входные данные, получаемые от других вершин.

**Пример 2.7.** Предположим, что данными является набор взвешенных ребер графа, и мы хотим для каждой вершины найти длины кратчайших путей ко всем другим вершинам. Первоначально в каждой вершине $a$ хранится набор пар $(b,w)$ таких, что существует ребро из $a$ в $b$ с весом $w$. Эти факты в начале работы алгоритма рассылаются всем остальным вершинам в виде троек $(a,b,w)$[6]. Вершина $a$, получив тройку $(c,d,w)$, ищет текущее расстояние до $c$, т. е. хранящуюся в локальном файле пару $(c,v)$, если такая существует. Она также пытается найти пару $(d, u)$. Если $w + v < u$, то пара $(d,u)$ заменяется парой $(d,w + v)$. Если же пара $(d,u)$ не была найдена, то в вершине $a$ сохраняется пара $(d,w + v)$. Кроме того, в обоих случаях остальным вершинам посылается сообщение $(a,d,w + v)$.

Вычисления в Pregel организованы как супершаги. На каждом супершаге все сообщения, полученные любой вершиной на предыдущем супершаге (или в начальный момент, если этот супершаг первый) обрабатываются, а затем все сообщения, сгенерированные этими вершинами, отправляются получателю.

В случае отказа вычислительного узла не делается попытки перезапустить работавшие на нем задачи. Вместо этого Pregel после некоторых супершагов сохраняет контрольную точку всего состояния вычисления. Контрольная точка состоит из копий состояния каждой задачи, что позволяет при необходимости перезапустить ее с этой точки. Если какой-нибудь вычислительный узел откажет, то все задание можно перезапустить с последней контрольной точки.

Эта стратегия восстановления заставляет многие задачи, не испытавшие отказ, переделать всю работу, но в ряде ситуаций это приемлемая цена. Напомним, что MapReduce поддерживает перезапуск только сбойных задач лишь для того, чтобы

[6] В этом алгоритме объем коммуникаций слишком велик, но он дает простой пример коммуникационной модели в Pregel.

гарантировать, что ожидаемое время завершения всего задания при наличии отказов будет не намного больше времени выполнения задания в отсутствие отказов. Любая система управления отказами будет обладать этим свойством при условии, что время восстановления после отказа намного меньше среднего времени между отказами. Таким образом, нужно лишь, чтобы Pregel сохраняла контрольную точку всего вычисления после такого количества супершагов, что вероятность отказа на протяжении этих супершагов достаточно мала.

### 2.4.4. Упражнения к разделу 2.4

**! Упражнение 2.4.1.** Допустим, что задание состоит из $n$ задач, каждая из которых занимает $t$ секунд. Следовательно, в отсутствие отказов общее время, которое потребуется узлам для выполнения всех задач равно $nt$. Допустим также, что вероятность отказа задачи равна $p$, а если задача отказывает, то накладные расходы на управление перезапуском таковы, что полное время выполнения задания увеличивается на $10t$ секунд. Каково математическое ожидание полного времени выполнения задания?

**! Упражнение 2.4.2.** Предположим, что для задания в системе Pregel вероятность отказа на любом супершаге равна $p$. Предположим также, что время сохранения контрольной точки (просуммированное по всем вычислительным узлам) в $c$ раз больше времени выполнения супершага. После скольких супершагов следует сохранять контрольную точку, чтобы минимизировать ожидаемое время выполнения задания?

## 2.5. Модель коммуникационной стоимости

В этом разделе мы познакомимся с моделью измерения качества алгоритмов, работающих в вычислительном кластере. Предполагается, что вычисление можно описать ациклическим потоковым графом, как в разделе 2.4.1. Для многих приложений узким местом является перемещение данных между задачами, например передача выходных файлов от распределителей к соответствующим редукторам. В качестве примера мы исследуем вычисление многопутевых соединений в виде одношаговых заданий MapReduce и увидим, что в некоторых случаях этот подход оказывается более эффективным, чем прямолинейная организация каскада двухпутевых соединений.

### 2.5.1. Коммуникационная стоимость для сетей задач

Представим, что некий алгоритм реализован ациклической сетью задач. Это могут быть распределители, подающие данные редукторам, как в стандартном алгорит-

ме MapReduce, или каскад из нескольких заданий MapReduce, или какая-то более общая структура, например коллекция задач, каждая из которых реализует поток работ, как на рис. 2.6[7]. *Коммуникационной стоимостью* задачи будет размер ее входных данных. Этот размер можно измерить в байтах. Но поскольку в наших примерах будут использоваться операции в реляционных базах данных, то часто мы будем выражать размер в кортежах.

*Коммуникационная стоимость алгоритма* равна сумме коммуникационных стоимостей всех задач, реализующих этот алгоритм. Мы будем рассматривать коммуникационную стоимость как основную меру эффективности алгоритма. В частности, при оценке времени работы алгоритма не будем принимать во внимание время выполнения каждой задачи. Хотя бывают исключения, когда время выполнения задач является доминирующим фактором, на практике они встречаются редко. Объяснить и обосновать важность коммуникационной стоимости можно следующим образом.

- Алгоритм, выполняемый каждой задачей, обычно очень простой, его время работы часто линейно зависит от размера входных данных.
- Типичная скорость передачи данных между узлами вычислительного кластера составляет один гигабит в секунду. На первый взгляд, очень много, но это ничто по сравнению со скоростью выполнения команд процессором. Более того, во многих кластерных архитектурах возникает конкуренция за соединительный канал, когда несколько узлов хотят обмениваться данными одновременно. Таким образом, узел может выполнить огромный объем работы над полученным входным элементом за время, которое требуется для его доставки.
- Даже если задача выполняется в узле, где имеется копия порции данных, которую эта задача обрабатывает, эта порция обычно хранится на диске, и время, необходимое для ее перемещения в оперативную память, может оказаться больше, чем время обработки данных, находящихся в памяти.

Но даже в предположении, что коммуникационная стоимость доминирует, возникает вопрос, почему мы измеряем только размер входа, а не размер выхода. Ответ состоит из двух пунктов.

1. Если выход одной задачи $\tau$ является входом другой задачи, то размер выхода $\tau$ будет учтен при измерении размера входа ее задачи-получателя. Поэтому имеет смысл подсчитывать только размер выхода тех задач, которые формируют окончательный результат алгоритма.
2. Но на практике размер выхода алгоритма редко сопоставим с размером входа или промежуточных данных. Причина в том, что массивные выходные данные было бы невозможно использовать без какого-то вида обобщения или агрегирования. И хотя в примере 2.6 речь шла о вычислении всего

[7] Напомним, что на этом рисунке представлены функции, а не задачи. В сети задач могло бы быть много задач, реализующих, к примеру, функцию *f*, и каждая из них подавала бы данные каждой задаче, реализующей функцию *g*, и каждой задаче, реализующей функцию *i*.

транзитивного замыкания графа, на практике нам нужно нечто гораздо более простое, например количество достижимых узлов для каждого узла или множество узлов, достижимых из конкретного узла.

**Пример 2.8.** Рассчитаем коммуникационную стоимость алгоритма соединения из раздела 2.3.7. Пусть мы вычисляем $R(A,B) \bowtie S(B,C)$, и $r$ и $s$ – размеры $R$ и $S$ соответственно. Каждая порция файлов, в которых хранятся $R$ и $S$, подается на вход одной задаче-распределителю, поэтому сумма коммуникационных стоимостей для всех распределителей равна $r + s$. Отметим, что в типичном случае каждый распределитель будет выполняться на узле, содержащем копию обрабатываемой им порции. Поэтому для распределителей межузловая коммуникация не нужна, но читать данные с диска им все равно придется. Поскольку распределители выполняют очень простое преобразование каждого входного кортежа в пару ключ-значение, то мы ожидаем, что стоимость вычисления будет мала по сравнению с коммуникационной стоимостью вне зависимости от того, хранятся входные данные локально или должны быть доставлены вычислительному узлу, на котором работает задача.

Сумма размеров выходных данных распределителей приблизительно такая же, как суммарный размер входа. Каждая пара ключ-значение передается ровно одному редуктору и маловероятно, что этот редуктор будет выполняться на том же узле, что распределитель. Поэтому коммуникационная стоимость передачи данных от распределителей редукторам, скорее всего, обусловлена межузловыми соединениями в кластере, а не записью из памяти на диск. Объем таких коммуникаций составляет $O(r+s)$, так что коммуникационная стоимость алгоритма соединения равна $O(r+s)$.

Задачи-редукторы выполняют операцию редукции (применение функции Reduce к ключу и ассоциированному с ним списку значений) для одного или нескольких значений атрибута $B$. Каждая операция редукция берет полученные входные данные и разделяет их на кортежи, поступившие из $R$ и из $S$. Каждый кортеж из $R$ объединяется в пару с каждым кортежем из $S$, и эта пара составляет один выходной элемент. Размер выхода для операции соединения может быть больше или меньше $r + s$ в зависимости от того, насколько вероятно соединение данного $R$-кортежа с данным $S$-кортежем. Например, можно ожидать, что если различных значений $B$ много, то размер выхода будет мал, а если мало – то велик.

Если размер выхода велик, то вычислительная стоимость генерации всех выходных данных редуктором может оказаться гораздо больше $O(r+s)$. Однако мы будет опираться на предположение о том, что если размер выхода операции соединения велик, то, вероятно, понадобится его как-то агрегировать и таким образом уменьшить. Нам придется передать результат соединения другому набору задач, которые произведут агрегирование, и, следовательно, коммуникационная стоимость будет как минимум пропорциональна стоимости вычислений, необходимых для порождения выхода соединения.

## 2.5.2. Физическое время

Хотя коммуникационная стоимость часто определяет выбор алгоритма для использования в кластерном окружении, необходимо также помнить о важности физического времени, т. е. времени, необходимого для завершения параллельного алгоритма. При бездумном подходе можно было бы минимизировать суммарную коммуникационную стоимость, просто поручив всю работу одной задаче; тогда коммуникации уж точно были бы сведены к минимуму. Однако физическое время работы такого алгоритма оказалось бы очень велико. Алгоритмы, которые мы предлагаем или предлагали до сих пор, обладают тем свойством, что работа равномерно распределяется между задачами. Поэтому физическое время настолько мало, насколько это вообще возможно при заданном количестве доступных узлов.

## 2.5.3. Многопутевое соединение

Чтобы понять, как анализ коммуникационной стоимости может помочь при выборе алгоритма в кластерном окружении, мы внимательно рассмотрим случай многопутевого соединения. Существует общая теория, согласно которой мы:

1. Выбираем некоторые атрибуты отношений, участвующие в естественном соединении трех или более отношений, предполагая хэшировать их значения в некоторое количество ячеек.
2. Выбираем количество ячеек для каждого из этих атрибутов, соблюдая ограничение: произведение числа ячеек для каждого атрибута – $k$ – равно количеству используемых редукторов.
3. Определяем все $k$ редукторов с помощью вектора номеров ячеек. В таком векторе есть по одному элементу для каждого атрибута, выбранного на шаге (1).
4. Отправляем кортежи каждого отношения тем из этих редукторов, которые могут найти кортежи, подходящие для соединения. Точнее, в данном кортеже $t$ будут значения некоторых атрибутов, выбранных на шаге (1), поэтому мы можем применить к ним хэш-функции и определить часть элементов вектора, описывающего редукторы. Прочие элементы вектора неизвестны, поэтому $t$ следует послать всем редукторам, которые соответствуют векторам с любыми значениями неизвестных элементов.

Несколько примеров этой общей техники приведены в упражнениях.

А здесь мы рассмотрим только соединение $R(A,B) \bowtie S(B,C) \bowtie T(C,D)$ в качестве примера. Пусть размеры отношений $R$, $S$ и $T$ соответственно равны $r$, $s$ и $t$. Для простоты предположим, что $p$ – вероятность того, что

1. $R$-кортеж и $S$-кортеж согласованы по $B$, а также вероятность того, что
2. $S$-кортеж и $T$-кортеж согласованы по $C$.

Если сначала соединить $R$ и $S$ с применением MapReduce-алгоритма из раздела 2.3.7, то коммуникационная стоимость будет равна $O(r+s)$, а размер промежуточного соединения $R \bowtie S$ составит $prs$. Если затем соединить результат с $T$, то коммуникационная стоимость второго задания MapReduce будет равна $O(t+prs)$.

Следовательно, полная коммуникационная стоимость алгоритма, состоящего из двух двухпутевых соединений равна $O(r+s+t+prs)$. Если же сначала соединить $S$ и $T$, а затем результат с $R$, то коммуникационная стоимость этого алгоритма составит $O(r+s+t+pst)$.

Третий способ вычислить это соединение заключается в том, чтобы использовать единственное задание MapReduce, которое соединяет все три отношения сразу. Предположим, что мы планируем использовать для этой цели $k$ редукторов. Выберем числа $b$ и $c$, представляющие количество ячеек, в которые будем хэшировать значения $B$ и $C$ соответственно. Пусть $h$ – хэш-функция, которая распределяет значения $B$ по $b$ ячейкам, а $g$ – хэш-функция, которая распределяет значения $C$ по $c$ ячейкам. Мы требуем, чтобы $bc = k$, т. е. каждой паре чисел – для значения $B$ и для значения $C$ – соответствует редуктор. Редуктор, соответствующий паре $(i, j)$, отвечает за соединение кортежей $R(u, v)$, $S(v, w)$ и $T(w, x)$ при условии, что $h(v) = i$ и $g(w) = j$.

Таким образом, задачи-распределители, которые отправляют кортежи из $R$, $S$ и $T$ заинтересованным в них редукторам, должны отправлять $R$- и $T$-кортежи сразу нескольким редукторам. Для $S$-кортежа $S(v, w)$ мы знаем значения атрибутов $B$ и $C$, поэтому можем отправить его только редуктору, соответствующему паре $(h(v), g(w))$. Рассмотрим, однако, $R$-кортеж $R(u, v)$. Мы знаем, что его нужно отправить только редукторам, соответствующим паре $(h(v), y)$ для некоторого $y$. Но значение $y$ нам неизвестно – атрибут $C$ может иметь произвольное значение. Поэтому мы должны отправить кортеж $R(u ,v)$ $c$ редукторам, поскольку $y$ может находиться в любой из $c$ ячеек для значений $C$. Точно так же, $T$-кортеж $T(w, x)$ необходимо отправить всем редукторам $(z, g(w))$ для любого значения $z$. Всего существует $b$ таких редукторов.

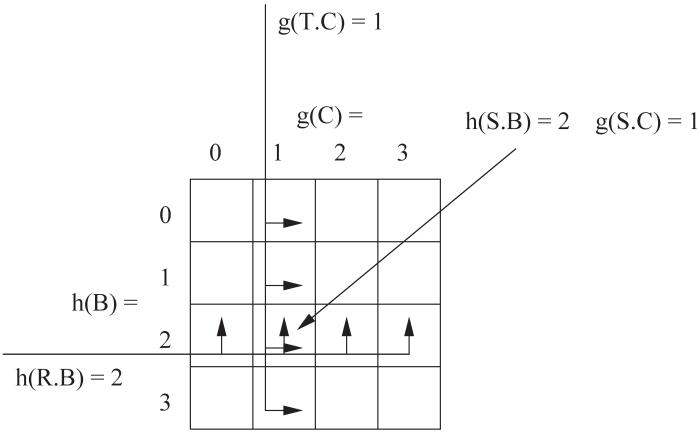



**Рис. 2.8.** Шестнадцать редукторов совместно выполняют трехпутевое соединение

**Пример 2.9.** Предположим, что $b = c = 4$, так что $k = 16$. Можно представить себе, что шестнадцать редукторов расположены в виде прямоугольника,

как на рис. 2.8. Мы видим гипотетический $S$-кортеж $S(v, w)$, для которого $h(v) = 2$ и $g(w) = 1$. Этот кортеж обрабатывающий его распределитель отправляет только редуктору, соответствующему ключу (2, 1). Мы также видим $R$-кортеж $R(u, v)$. Поскольку $h(v) = 2$, этот кортеж отправляется всем редукторам $(2, y)$ для $y = 1, 2, 3, 4$. Наконец, мы видим $T$-кортеж $T(w, x)$. Поскольку $g(w) = 1$, этот кортеж отправляется всем редукторам $(z, 1)$ для $z = 1, 2, 3, 4$. Отметим, что эти три кортежа соединяются и должны встретиться ровно в одном редукторе, а именно том, который соответствует ключу (2, 1).

### Вычислительная стоимость трехпутевого соединения

Каждый редуктор должен соединять части трех отношений, и разумно задаться вопросом, можно ли выполнить такое соединение за время, линейно зависящее от размера входных данных задачи-редуктора. Хотя более сложные операции соединения не всегда можно вычислить за линейное время, в нашем примере каждый редуктор может сделать это эффективно. Сначала построим индекс по $R.B$, чтобы организовать полученные $R$-кортежи. Точно так же построим индекс по $T.C$ для $T$-кортежей. Затем рассмотрим каждый полученный $S$-кортеж $S(v, w)$. Воспользуемся индексом по $R.B$ для нахождения всех $R$-кортежей, в которых $R.B = v$, и индексом по $T.C$ для нахождения всех $T$-кортежей, в которых $T.C = w$.

Предположим теперь, что размеры $R$, $S$ и $T$ различны; напомним, что мы обозначили эти размеры $r$, $s$ и $t$ соответственно. Если хэшировать значения $B$ в $b$ ячеек, а значения $C$ – в $c$ ячеек, где $bc = k$, то полная коммуникационная стоимость доставки кортежей соответствующим редукторам равна сумме трех слагаемых:

1. $s$, чтобы однократно переместить каждый кортеж $S(v, w)$ редуктору $(h(v), g(w))$.
2. $cr$, чтобы переместить каждый кортеж $R(u, v)$ $c$ редукторам $(h(v), y)$ для каждого из $c$ возможных значений $y$.
3. $bt$, чтобы переместить каждый кортеж $T(w, x)$ $b$ редукторам $(z, g(w))$ для каждого из $c$ возможных значений $z$.

Еще необходимо доставить каждый кортеж каждого отношения задачам-распределителям. Стоимость этой работы составляет $r + s + t$, но она фиксирована и не зависит от $b$, $c$ и $k$.

Мы должны выбрать $b$ и $c$, соблюдая ограничение $bc = k$, так чтобы минимизировать сумму $s + cr + bt$. Воспользуемся методом множителей Лагранжа, чтобы найти точку, в которой частные производные функции $s + cr + bt - \lambda(bc - k)$ по $b$ и $c$ обращаются в 0. То есть мы должны решить уравнения $r - \lambda b = 0$ и $t - \lambda c = 0$. Поскольку $r = \lambda b$ и $t = \lambda c$, то, почленно перемножая эти равенства, получаем $rt = \lambda^2 bc$. Так как $bc = k$, то имеем $rt = \lambda^2 k$ или $\lambda = \sqrt{rt/k}$. Таким образом, минимум коммуникационной стоимости достигается, когда $c = t/\lambda = \sqrt{kt/r}$ $kt/r$ и $b = r/\lambda = \sqrt{kr/t}$ $kr/t$.

Подставив эти значения в формулу $s + cr + bt$, получим $s + 2\sqrt{krt}$. Это и есть коммуникационная стоимость редукторов, к которой нужно еще прибавить ве-

личину $s + r + t$ коммуникационной стоимости распределителей. Следовательно, полная коммуникационная стоимость равна $r + 2s + t + 2\sqrt{krt}$. В большинстве случаев слагаемым $r + t$ можно пренебречь, потому что оно меньше $2\sqrt{krt}$ обычно в $O(\sqrt{k})$ раз.

**Пример 2.10.** Рассмотрим, при каких условиях коммуникационная стоимость трехпутевого соединения меньше, чем стоимость каскада из двух двухпутевых соединений. Для простоты предположим, что $R$, $S$ и $T$ – одно и то же отношение $R$, представляющее «друзей» в социальной сети типа Facebook. В Facebook примерно миллиард подписчиков, и у каждого в среднем 300 друзей, поэтому отношение $R$ содержит $r = 3 \times 10^{11}$ кортежей. Пусть требуется вычислить $R \bowtie R \bowtie R$, быть может, для того, чтобы затем найти число друзей друзей друзей каждого подписчика, или просто человека, у которого больше всего друзей друзей друзей[8]. Стоимость трехпутевого слияния $R$ с самим собой составляет $4r + 2r\sqrt{k}$; из этой величины $3r$ приходится на распределители, а $r + 2\sqrt{kr^2}$ – на редукторы. Т. к. по предположению $r = 3 \times 10^{11}$, то стоимость равна $1.2 \times 10^{12} + 6 \times 10^{11}\sqrt{k}$.

Теперь рассмотрим коммуникационную стоимость соединения $R$ с собой, а затем еще раз с $R$. Стоимость как распределителей, так и редукторов для первого соединения равна $2r$, поэтому стоимость одного лишь первого соединения равна $4r = 1.2 \times 10^{12}$. Но размер $R \bowtie R$ велик. Мы не можем точно сказать, насколько велик, потому что друзья обычно распадаются на узкие группы, поэтому у человека, имеющего 300 друзей, количество друзей друзей будет намного меньше потенциального максимума 90 000. Примем осторожную оценку и будем считать, что размер $R \bowtie R$ равен не $300r$, а только $30r$, или $9 \times 10^{12}$. Тогда коммуникационная стоимость второго соединения $(R \bowtie R) \bowtie R$ будет равна $1.8 \times 10^{13} + 6 \times 10^{11}$. А общая стоимость обоих соединений составит $1.2 \times 10^{12} + 1.8 \times 10^{13} + 6 \times 10^{11} = 1.98 \times 10^{13}$.

Спрашивается, когда стоимость трехпутевого соединения, равная

$$1.2 \times 10^{12} + 6 \times 10^{11}\sqrt{k}$$

будет меньше $1.98 \times 10^{13}$. Тогда, когда $6 \times 10^{11}\sqrt{k} < 1.86 \times 10^{13}$, или $\sqrt{k} < 31$. Таким образом, трехпутевое соединение предпочтительно, если используется не более $31^2 = 961$ редукторов.

## 2.5.4. Упражнения к разделу 2.5

**Упражнение 2.5.1.** Выразите коммуникационную стоимость каждого из следующих алгоритмов в виде функции от размера отношений, матриц или векторов, к которым они применяются.

(*а*) Алгоритм умножения матрицы на вектор из раздела 2.3.2.

(*б*) Алгоритм объединения из раздела 2.3.6.

[8] С этого человека, или более общо, людей с большим расширенным кругом друзей было бы очень неплохо начать маркетинговую кампанию, предложив им бесплатные образцы.

(*в*) Алгоритм агрегирования из раздела 2.3.8.

(*г*) Алгоритм умножения матриц из раздела 2.3.10.

### Звездообразные соединения

В коммерческих приложениях добычи данных широко распространена специальная структура данных – *звездообразное соединение*. Например, в базе данных сети магазинов, скажем Walmart, хранится *таблица фактов*, в которой каждый кортеж представляет одну продажу. Это отношение вида $F(A1, A2, \ldots)$, в котором каждый атрибут $A_i$ является ключом, описывающим одну из важных характеристик продажи, например: покупателя, купленный товар, конкретный магазин и дату. Каждому ключевому атрибуту соответствует *размерная таблица*, содержащая дополнительную информацию об этом атрибуте. Например, размерная таблица $D(A_1, B_{11}, B_{12}, \ldots)$ может представлять покупателей. Если $A_1$ – идентификатор покупателя, то $B_{1i}$ может быть его фамилией, адресом, телефоном и т. д. Как правило, таблица фактов гораздо больше размерных таблиц. Скажем, в таблице фактов может быть миллиард кортежей, а каждой из размерных таблиц – по миллиону.

Аналитики предъявляют *аналитические запросы*, в которых таблица фактов соединяется с несколькими размерными таблицами («звездообразное соединение»), а затем результат агрегируется для получения какой-то полезной информации. К примеру, аналитик может попросить «дай мне таблицу продаж брюк в разрезе регионов и цветов за каждый месяц 2012 года». С точки зрения модели коммуникационной стоимости, многопутевое соединение таблицы фактов с размерными таблицами почти наверняка эффективнее попарного соединения. На самом деле, имеет смысл хранить копии таблицы фактов во всех вычислительных узлах и реплицировать размерные таблицы на постоянной основе точно так же, как мы реплицировали бы их при соединении с таблицей фактов. В этом частном случае по ячейкам хэшируются только ключевые атрибуты (атрибуты $A$), а количество ячеек для каждого такого атрибута пропорционально размеру соответствующей ему размерной таблицы.

**! Упражнение 2.5.2.** Пусть $r$, $s$ и $t$ – размеры отношений $R$, $S$ и $T$ соответственно и пусть требуется произвести трехпутевое соединение $R(A,B) \bowtie S(B,C) \bowtie T(A,C)$ с использованием $k$ редукторов. Мы будем хэшировать значения атрибутов $A$, $B$, $C$ в $a$, $b$, $c$ ячеек соответственно, где $abc = k$. С каждым редуктором ассоциирован вектор ячеек, по одной для каждой из трех хэш-функций. Выразите в виде функции от $r$, $s$, $t$, $k$ значения $a$, $b$, $c$, которые минимизируют коммуникационную стоимость алгоритма.

**! Упражнение 2.5.3.** Пусть требуется вычислить звездообразное соединение таблицы фактов $F(A_1, A_2, ..., A_m)$ с размерными таблицами $D_i(A_i, B_i)$ для $i = 1, 2, ..., m$. Предположим, что имеется $k$ редукторов, с каждым из которых ассоциирован вектор ячеек, по одной для каждого ключевого атрибута $A_1, A_2, ..., A_m$.

Обозначим $a_i$ число ячеек, в которые хэшируется атрибут $A_i$. Естественно, $a_1 a_2 \ldots a_m = k$. Наконец, обозначим $d_i$ размер размерной таблицы $D_i$ и предположим, что размер таблицы фактов много больше размера любой размерной таблицы. Найдите значения $a_i$, которые минимизируют стоимость вычисления звездообразного соединения одной операцией MapReduce.

# 2.6. Теория сложности MapReduce

Изучим теперь структуру алгоритмов MapReduce более пристально. В разделе 2.5 мы выдвинули идею о том, что на коммуникацию между распределителями и редукторами приходится львиная доля затрачиваемого времени. А сейчас мы рассмотрим, как коммуникационная стоимость соотносится с другими аспектами алгоритмов MapReduce и, в частности, с желанием уменьшить физическое время выполнения и выполнять каждую операцию редукции, не выходя за пределы оперативной памяти. Напомним, что «редукцией» мы называем применение функции Reduce к одному ключу и ассоциированному с ним списку значений. Это исследование предпринято потому, что для многих задач существует целый спектр алгоритмов MapReduce, требующих различного объема коммуникаций. Но может оказаться, что алгоритм с меньшим объемом коммуникаций хуже в других отношениях, включая физическое время и объем потребляемой памяти.

## 2.6.1. Размер редукции и коэффициент репликации

Введем два параметра, характеризующие семейства алгоритмов MapReduce. Первый – *размер редукции*, который мы обозначим $q$. Это верхняя граница числа значений в списке, ассоциированном с одним ключом. При выборе размера редукции руководствуются, по меньшей мере, двумя соображениями.

1. Выбор небольшого размера может привести к увеличению числа операций редукции, т. е. количества различных ключей, по которым задача разбивается распределителями. Если мы также создадим много редукторов – вплоть до отдельного редуктора на каждую операцию редукции – то повысится степень параллелизма, и можно будет ожидать уменьшения физического времени.
2. Можно выбрать размер редукции настолько малым, что соответствующее вычисление заведомо будет производиться в оперативной памяти узла, на котором размещен редуктор. Вне зависимости от характера вычисления время выполнения редукции существенно уменьшится, если мы сможем избежать постоянного перемещения данных между оперативной памятью и диском.

Второй параметр – *коэффициент репликации*, который мы обозначим $r$. Он определяется как количество пар ключ-значение, порождаемых всеми распределителями для всех входных файлов, поделенное на количество входных файлов. Иначе

говоря, коэффициент репликации – это средний объем данных, передаваемых от распределителей редукторам (измеренный в количестве пар ключ-значение), в расчете на один входной файл.

**Пример 2.11.** Рассмотрим однопроходный алгоритм умножения матриц из раздела 2.3.10. Предположим, что все матрицы имеют размерность $n \times n$. Тогда коэффициент репликации $r$ равен $n$. В этом легко убедиться, потому что для каждого элемента $m_{ij}$ порождается $n$ пар ключ-значение; ключи в них имеют вид $(i, k)$, где $1 \leq k \leq n$. Аналогично для каждого элемента другой матрицы, например $n_{jk}$, мы порождаем $n$ пар ключ-значение, в которых ключи имеют вид $(i, k)$, $1 \leq i \leq n$. В этом случае $n$ – не просто среднее число пар ключ-значение, порождаемых для входного элемента, но для каждого входного элемента порождается в точности такое число пар.

Мы также видим, что $q$, требуемый размер редукции, равен $2n$. То есть для каждого ключа $(i, k)$ существует $n$ пар ключ-значение, представляющих элементы $m_{ij}$ первой матрицы, и еще $n$ пар ключ-значение, произведенных из элементов $n_{jk}$ второй матрицы. Эти два параметра описывают только один конкретный алгоритм однопроходного умножения матриц, но ниже мы увидим, что он входит в некое семейство алгоритмов и в действительности представляет собой точку экстремума, в которой $q$ достигает минимума, а $r$ – максимума. В общем случае существует компромисс между выбором $r$ и $q$, выражаемый неравенством $qr \geq 2n^2$.

## 2.6.2. Пример: соединение по сходству

Чтобы понять, как выглядит компромисс между $r$ и $q$ в реальной ситуации, рассмотрим задачу о так называемом *соединении по сходству* (similarity join). В этой задаче дан большой набор элементов $X$ и мера сходства $s(x, y)$, которая говорит, насколько похожи элементы $x$ и $y$, принадлежащие $X$. В главе 3 мы изучим наиболее важные понятия сходства и узнаем о некоторых приемах, позволяющих быстро находить похожие пары. Здесь же нас будет интересовать только задача в чистом виде, когда требуется взять каждую пару элементов $X$ и определить степень их сходства, применив функцию $s$. Предполагается, что функция $s$ симметрична, т. е. $s(x, y) = s(y, x)$, но больше никаких предположений о ней не делается. На выходе алгоритм должен вернуть пары, для которых степень сходства превышает заданный порог $t$.

Предположим, к примеру, что имеется набор из миллиона изображений, каждое размером 1 мегабайт. Следовательно, размер всего набора данных равен терабайту. Мы не будем пытаться описать функцию сходства $s$, но она могла бы, скажем, возвращать большее значение, если распределение цветов в изображениях примерно одинаково или если в изображениях есть соответственные области с одинаковым распределением цветов. Цель – найти пары изображений, на которых запечатлены объекты или сцены одного типа. Это чрезвычайно трудная задача, но классификация по распределению цветов обычно является одним из шагов на пути к цели.

Посмотрим, как можно было выполнить это вычисление с помощью MapReduce, чтобы воспользоваться присутствующим в задаче естественным параллелизмом. Входными данными являются пары $(i, P_i)$, где $i$ – идентификатор изображения, а $P_i$ – само изображение. Мы хотим сравнивать каждую пару изображений, поэтому будем использовать один ключ для каждого набора двух идентификаторов $\{i, j\}$. Существует приблизительно $5 \times 10^{11}$ пар идентификаторов. Мы хотим ассоциировать с каждым ключом $\{i, j\}$ два значения $P_i$ и $P_j$, так что на вход соответствующей операции редукции попадает пара $(\{i, j\}, [P_i, P_j])$. Тогда функция Reduce может просто применить функцию сходства $s$ к двум изображениям в списке значений, т. е. вычислить $s(P_i, P_j)$ и решить, превышает результат пороговое значение или нет. Если превышает, пара выводится.

Увы, этот алгоритм никуда не годится. Размер редукции мал, поскольку во всех списках не больше двух значений, общим объемом 2 МБ. Хотя мы не знаем точно, как работает функция сходства $s$, но вправе ожидать, что ей не потребуется вся имеющаяся оперативная память. Однако коэффициент репликации составляет 999 999, потому что для каждого изображения мы генерируем много пар ключ-значение – по одной для всех остальных изображений в наборе данных. Общее количество байтов, передаваемых от распределителей редукторам, равно произведению 1 000 000 (число изображений) на 999 999 (коэффициент репликации) и на 1 000 000 (размер каждого изображения). Получается $10^{18}$ байтов, или один экзабайт. Для передачи такого объема данных по гигабитной сети Ethernet понадобится $10^{10}$ секунд, или примерно 300 лет[9].

К счастью, этот алгоритм – лишь крайняя точка в целом спектре возможных алгоритмов. Чтобы охарактеризовать их, объединим изображения в $g$ групп по $10^6/g$ изображений в каждой.

- **Функция Map.** Взять входной элемент $(i, P_i)$ и сгенерировать $g - 1$ пар ключ-значение. В каждой паре ключом будет одно из множеств $\{u, v\}$, где $u$ – группа, которой принадлежит изображение $i$, а $v$ – одна из других групп. Ассоциированное с этим ключом значение – пара $(i, P_i)$.
- **Функция Reduce.** Рассмотрим ключ $\{u, v\}$. В ассоциированном с ним списке значений $2 \times 10^6/g$ элементов $(j, P_j)$, где $j$ принадлежит любой из групп $u$ или $v$. Функция Reduce перебирает все комбинации $(i, P_i)$ и $(j, P_j)$ в этом списке, где $i$ и $j$ принадлежат разным группам, и вычисляет функцию сходства $s(P_i, P_j)$. Кроме того, нам нужно сравнить изображения, принадлежащие одной группе, но мы не хотим производить одно и то же сравнение в каждой из $g - 1$ операций редукции, в которых ключ содержит данный номер группы. Решить эту проблему можно разными способами, один из них описан ниже. Сравниваем члены группы $u$ в операции редукции $\{u, u + 1\}$, где «+1» – сложение с оборачиванием при выходе за границу. То есть, если

[9] В типичном кластере много коммутаторов, соединяющих подмножества вычислительных узлов, так что не все данные проходят через единственный гигабитный коммутатор. Но все равно пропускная способность сети недостаточна для реализации этого алгоритма с предложенными размерными параметрами.

$u = g$ ($u$ – последняя группа), то $u + 1$ – группа 1, в противном случае $u + 1$ – группа с номером на 1 большим $u$.

Мы можем вычислить коэффициент репликации и размер редукции в виде функции от числа групп $g$. Каждый входной элемент превращается в $g - 1$ пар ключ-значение. Следовательно, коэффициент репликации равен $g - 1$, но можно считать, что $r = g$, т. к. предполагается, что количество групп велико. Размер редукции равен $2 \times 10^6/g$, поскольку таково количество значений в списке для каждой операции редукции. Размер каждого значения примерно один мегабайт, так что для хранения входных данных понадобится $2 \times 10^{12}/g$ байтов.

**Пример 2.12.** Если $g$ равно 1000, то размер входных данных составляет примерно 2 ГБ, т. е. данные целиком помещаются в типичную оперативную память. А общее число передаваемых байтов теперь равно $10^6 \times 999 \times 10^6$, или примерно $10^{15}$ байтов. Этот объем еще очень велик, но все же в 1000 раз меньше, чем в наивном алгоритме. Кроме того, требуется примерно полмиллиона редукций. Вряд ли у нас есть столько вычислительных узлов, поэтому мы можем объединить операции редукции в несколько задач-редукторов и обеспечить при этом занятость всех узлов, т. е. получить такую степень параллелизма, какую поддерживает кластер.

Вычислительная стоимость алгоритмов из этого семейства не зависит от числа групп $g$, если только входные данные для каждой операции редукции помещаются в оперативной памяти. Причина в том, что вычисление в основном сводится к применению функции $s$ к паре изображений. При любом значении $g$ функция $s$ применяется к каждой паре один и только один раз. Поэтому, хотя работу алгоритмов из этого семейства можно распределить между операциями редукции по-разному, все члены семейства выполняют одно и то же вычисление.

## 2.6.3. Графовая модель для проблем MapReduce

В этом разделе мы начнем изучать технику, которая для ряда проблем[10] позволит получить более низкий коэффициент репликации в виде функции от размера редукции. Первым делом мы познакомимся с графовой моделью проблем. Для каждой проблемы, решаемой алгоритмом MapReduce, существует:

1. Множество входов.
2. Множество выходов.
3. Связь многие-ко-многим между входами и выходами, которая описывает, какие входы необходимы для порождения каких выходов.

**Пример 2.13.** На рис. 2.9 показан граф для проблемы соединения по сходству, рассмотренной в разделе 2.6.2, в случае, когда изображений не миллион, а всего четыре. Входами являются изображения, выходами – шесть

[10] К сожалению, слово «задача» в русском языке чрезмерно перегружено. Поэтому здесь слово «problem» переводится как «проблема», чтобы не путать со словом «task» (задача), употребляемым в том же контексте. – *Прим. перев.*

возможных пар изображений. Каждый выход соединен с двумя входами, которые являются членами его пары.

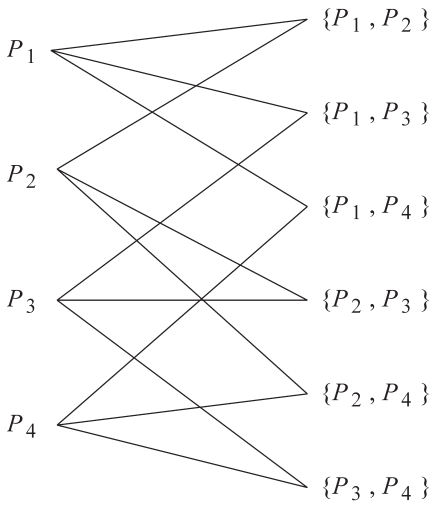



**Рис. 2.9.** Связи между входами и выходами для соединения по сходству

**Пример 2.14.** Граф проблемы умножения матриц сложнее. Если мы перемножаем матрицы $M$ и $N$ размерности $n \times n$ и получаем матрицу $P$, то существует $2n^2$ входов, $m_{ij}$ и $n_{jk}$, и $n^2$ выходов $p_{ik}$. Каждый выход $p_{ik}$ соединен с $2n$ входами: $m_{i1}$, $mp_{i2}$, ..., $m_{in}$ и $n_{1k}$, $n_{2k}$, ..., $n_{nk}$. Кроме того, каждый вход связан с $n$ выходами. Например, $m_{ij}$ связан с $p_{i1}$, $p_{i2}$, ..., $p_{in}$. На рис. 2.10 показан граф связей между входами и выходами для умножения матриц в простом случае матриц размерности $2 \times 2$, а точнее:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} i & j \\ k & l \end{bmatrix}.$$

В примерах 2.13 и 2.14 присутствуют все входы и выходы. Однако бывает и так, что все возможные входы и (или) выходы не представлены ни в каком конкретном экземпляре проблемы. В качестве примера рассмотрим естественное соединение $R(A,B)$ и $S(B,C)$, обсуждавшееся в разделе 2.3.7. Мы предполагаем, что область определения каждого атрибута $A$, $B$, $C$ конечна, т. е. существует конечное число возможных входов и выходов. Входами являются все возможные $R$-кортежи, т. е. пары, состоящие из одного значения из области определения $A$ и одного значения из области определения $B$, и все возможные $S$-кортежи – пары значений из областей определения $B$ и $C$. Выходами являются все возможные тройки, компоненты которых берутся из областей определения $A$, $B$, $C$ в указанном порядке. Выход $(a, b, c)$ соединен с двумя входами: $R(a, b)$ и $S(b, c)$.

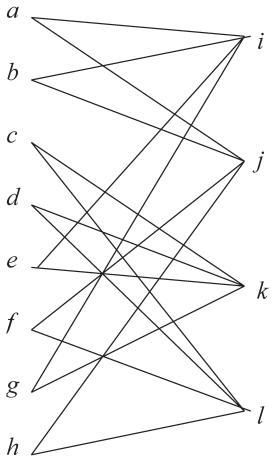



**Рис. 2.10.** Связи между входами и выходами для умножения матриц

Но при вычислении соединения будут присутствовать только некоторые из возможных входов и, следовательно, будут порождаться только некоторые из возможных выходов. Этот факт не влияет на граф проблемы. Нам все равно нужно знать, как каждый возможный выход связан с входами, и неважно, порождается этот выход в конкретной проблеме или нет.

## 2.6.4. Схема сопоставления

Поняв, как проблемы, решаемые с помощью MapReduce, представляются в виде графов, мы можем определить требования к MapReduce-алгоритму для решения данной проблемы. В любом таком алгоритме должна быть *схема сопоставления*, показывающая, как выходы порождаются различными используемыми в алгоритме операциями редукции. Иными словами, схема сопоставления для данной проблемы с данным размером редукции $q$ определяет назначение входов одной или нескольким операциям редукции с соблюдением следующих условий:

1. Ни одной операции редукции не назначено более $q$ входов.
2. Для любого выхода существует хотя бы одна операция редукции, которой назначены все входы, связанные с данным выходом. Мы говорим, что эта редукция *покрывает* выход.

Можно сказать, что существование схемы сопоставления для любого размера редукции – это то, что отличает проблемы, разрешимые с помощью одного задания MapReduce, от неразрешимых.

**Пример 2.15.** Давайте пересмотрим стратегию «группировки», описанную в разделе 2.6.2 в связи с обсуждением соединения по сходству. Обобщим проблему, предположив, что вход представляет собой $p$ изображений, которые разбиваются на $g$ групп равного размера $p/g$. Количество выходов

равно $\binom{p}{2}$, или приблизительно $p^2/2$. Операция редукции будет получать входы из двух групп, т. е. $2p/g$ входов, поэтому размер редукции равен $q = 2p/g$. Каждое изображение отправляется операциям редукции, которые соответствуют парам, состоящим из его группы и любой из $g - 1$ других групп. Таким образом, коэффициент репликации равен $g - 1$, т. е. приблизительно $g$. Если заменить $g$ коэффициентом репликации $r$ в формуле $q = 2p/g$, то найдем, что $r = 2p/q$. Следовательно, коэффициент репликации обратно пропорционален размеру редукции. Это соотношение типично: чем меньше размер редукции, тем больше коэффициент репликации и, значит, тем выше объем передаваемых данных.

Это семейство алгоритмов описывается семейством схем сопоставления, по одной для каждого возможного $q$. В схеме сопоставления для $q = 2p/g$ существует $\binom{g}{2}$, или приблизительно $g^2/2$ операций редукции. Каждая редукция соответствует паре групп, а вход $P$ назначается всем редукциям, чья пара включает группу $P$. Таким образом, ни одной редукции не назначается более $2p/g$ входов; на самом деле, каждой редукции ровно столько входов и назначается. Далее, каждый выход покрыт хотя бы одной редукцией. Точнее, если выходом является пара из двух разных групп $u$ и $v$, то он покрыт редукцией для пары $\{u, v\}$. Если же выход соответствует входам только из одной группы $u$, то он покрыт несколькими редукциями – теми, что соответствуют множеству пар $\{u, v\}$ для любого $v \neq u$. Отметим, что в описанном алгоритме только одна из этих операций редукции вычисляет выход, но в принципе *могла бы* вычислить любая.

Из того, что выход зависит от конкретного входа, следует, что когда этот вход обработан распределителем, будет сгенерирована по меньшей мере одна пара ключ-значение, используемая при вычислении данного выхода. Значение может и не совпадать с входом (как в примере 2.15), но оно произведено из этого входа. Важно, что для каждого связанного входа и выхода существует единственная пара ключ-значение, которую необходимо передать. Отметим, что технически никогда не возникает не возникает необходимости передавать более одной пары ключ-значение для заданных входа и выхода, потому что вход можно было бы передать операции редукции в неизменном виде, и все те преобразования входа, которые выполняет функция Map, можно было бы выполнить в функции Reduce той операции редукции, которая порождает данный выход.

## 2.6.5. Когда присутствуют не все входы

В проблеме из примера 2.15 мы знаем, что присутствуют все возможные входы, потому что можем определить входной набор, как состоящий из тех и только тех изображений, которые хранятся в базе данных. Но, как было сказано в конце раздела 2.6.3, бывают проблемы, например вычисление соединения, где граф входов и выходов описывает входы, которые могли бы существовать, и выходы, которые порождаются только в случае, когда в наборе данных существует хотя бы один из этих входов. Конкретно в случае соединения для порождения некоторого выхода должны существовать оба входа, связанные с этим выходом.

Алгоритм решения проблемы, в которой выходы могут отсутствовать, все равно нуждается в схеме сопоставления. Объясняется это тем, что *могли бы* присутствовать все входы или некоторое их подмножество, поэтому алгоритм без схемы сопоставления не смог бы породить некий выход, если все связанные с ним входы присутствуют, но ни одна операция редукция этот выход не покрывает.

Единственное, чем отличается случай отсутствия некоторых входов, – это необходимость переосмыслить выбор желаемого значения размера редукции $q$ при выборе алгоритма из семейства. В особенности, если в качестве $q$ мы выбираем число с расчетом, что входные данные займут всю оперативную память, то имеет смысл несколько увеличить $q$, учтя возможность отсутствия некоторых входов.

> **Пример 2.16.** Допустим, мы знаем, что можем выполнить функцию Reduce в оперативной памяти для некоторого ключа и ассоциированного с ним списка, содержащего $q$ значений. Но мы также знаем, что в наборе данных реально присутствует только 5 % всех возможных входов. Тогда схема сопоставления для размера редукции $q$ в действительности отправит лишь примерно $q/20$ входов каждой операции редукции. Иначе говоря, мы могли бы использовать алгоритм для размера редукции $20q$ и ожидать, что в списке для каждой редукции будет в среднем $q$ входов. Следовательно, можно взять размер редукции равным $20q$ или несколько меньше, скажем $18q$ – из-за случайного разброса количества входов, реально подаваемых каждой операции редукции.

## 2.6.6. Нижняя граница коэффициента репликации

Семейство алгоритмов соединения по сходству, описанное в примере 2.15, позволяет выбирать компромисс между объемом передачи данных и размером редукции, а, варьируя размер редукции, обменивать объем данных на степень параллелизма или на возможность выполнить функцию Reduce в оперативной памяти. Как узнать, что выбран оптимальный компромисс? Если мы хотим быть уверены, что достигнут минимально возможный объем коммуникации, то должны доказать существование соответствующей нижней границы. Воспользовавшись в качестве отправной точки существованием схемы сопоставления, мы часто можем вычислить нижнюю границу. Ниже приведен план применения этой методики.

1. Установить верхнюю границу количества выходов, которые может покрыть операция редукции с $q$ входами. Назовем эту границу $g(q)$. Этот шаг может быть трудным, но в примерах типа соединения по сходству, все очень просто.
2. Определить общее количество выходов, порождаемых проблемой.
3. Предположим, что существует $k$ операций редукции и что у $i$-ой операции имеется $q_i < q$ входов. Отметим, что сумма $\sum_{i=1}^{k} g(q_i)$ должна быть не меньше количества выходов, вычисленного на шаге (2).
4. Преобразовать неравенство из шага (3), так чтобы получить нижнюю границу $\sum_{i=1}^{k} q_i$. Часто на этом шаге помогает такой прием: заменить некоторые

множители $q_i$ их верхней границей $q$, оставив один множитель $q_i$, зависящий от $i$.

5. Поскольку $\sum_{i=1}^{k} q_i$ – это общий объем данных, передаваемых от распределителей редукторам, разделить нижнюю границу этой величины, полученную на шаге (4), на количество входов. Полученный результат и есть нижняя граница коэффициента репликации.

**Пример 2.17.** Эта последовательность шагов может показаться загадочной, но рассмотрим в качестве примера соединение по сходству в надежде прояснить ситуацию. Напомним, что в примере 2.15 мы нашли верхнюю границу коэффициента репликации $r$ в виде $r \leq 2p/q$, где $p$ – количество входов, а $q$ – размер редукции. Мы покажем, что нижняя граница $r$ вдвое меньше, откуда следует, что хотя улучшения алгоритма и возможны, уменьшить объем передаваемых данных при заданном размере редукции удастся не более чем вдвое.

При выполнении шага (1) заметим, что если операция редукции получает $q$ входов, то она не может покрыть более $\binom{g}{2}$, или приближенно $q^2/2$ выходов. Что касается шага (2), то мы знаем, что всего существует $\binom{p}{2}$, или приближенно $p^2/2$ выходов, и все они должны быть покрыты. Таким образом, на шаге (3) мы можем записать такое неравенство:

$$\sum_{i=1}^{k} q_i^2 / 2 \geq p^2/2$$

или, умножая обе части на 2:

$$\sum_{i=1}^{k} q_i^2 \geq p^2 . \qquad (2.1)$$

Теперь мы должны преобразовать это неравенство, как предписывает шаг (4). Следуя совету, замечаем, что в каждом члене в левой части (2.1) есть два множителя $q_i$, поэтому один из них заменим на $q$, а второй оставим равным $q_i$. Так как $q \geq q_i$, левая часть при этом может только увеличиться, следовательно, неравенство останется в силе:

$$q\sum_{i=1}^{k} q_i \geq p^2$$

или, поделив обе части на $q$:

$$\sum_{i=1}^{k} q_i \geq p^2 / q . \qquad (2.2)$$

На последнем шаге (5) мы должны поделить обе части неравенства (2.2) на $p$, количество входов. В результате левая часть, $(\sum_{i=1}^{k} q_i)/p$, окажется равна коэффициенту репликации, а в правой части останется $p/q$. Таким образом, мы нашли нижнюю границу $r$:

$$r \geq p/q$$

Как и утверждалось выше, это доказывает, что в любом алгоритме из семейства, описанного в примере 2.15, коэффициент репликации не более

чем вдвое превосходит минимально возможный.

## 2.6.7. Пример: умножение матриц

В этом разделе мы применим технику нахождения нижней границы к однопроходным алгоритмам умножения матриц. Один такой алгоритм мы видели в разделе 2.3.10, но это лишь крайний случай целого семейства алгоритмов. В частности, для этого алгоритма редукция соответствует одному элементу выходной матрицы. Но как в проблеме соединения по сходству мы группировали входы, чтобы сократить объем передаваемых данных за счет увеличения размера редукции, так и здесь мы можем сгруппировать строки и столбцы в полосы. Каждая пара, состоящая из полосы строк первой матрицы и полосы столбцов второй матрицы, используется в одной операции редукции для порождения квадратного блока элементов выходной матрицы. Пример показан на рис. 2.11.

Теперь подробнее. Предположим, что требуется вычислить произведение $MN = P$, и все три матрицы имеют размерность $n \times n$. Сгруппируем строки $M$ в $g$ полос по $n/g$ строк в каждой, а столбцы $N$ в $g$ полос по $n/g$ столбцов в каждой. Эта группировка иллюстрируется на рис. 2.11. Ключи соответствуют двум группам (полосам): одна из $M$, другая из $N$.

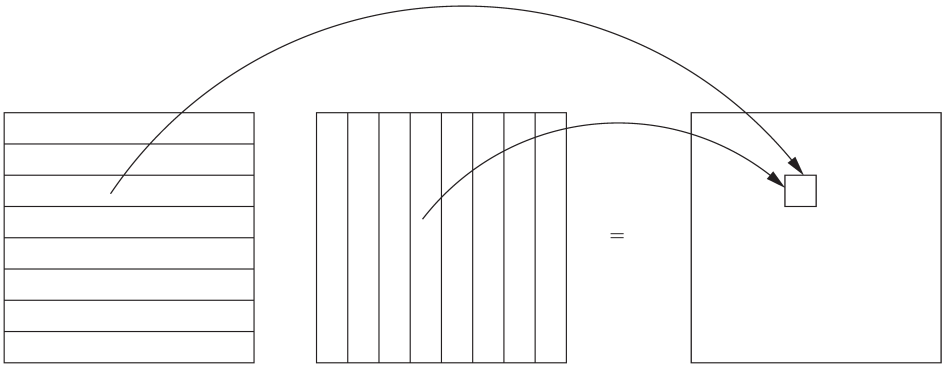



**Рис. 2.11.** Разбиение матрицы на полосы для уменьшения объема передаваемых данных

- **Функция Map.** Для каждого элемента $M$ функция Map генерирует $g$ пар ключ-значение. В каждом случае значением является сам элемент вместе с номерами своих строки и столбца, чтобы функция Reduce могла его идентифицировать. Ключом является пара, состоящая из группы, которой принадлежит элемент, и любой группы матрицы $N$. Аналогично для каждого элемента $N$ функция Map генерирует $g$ пар ключ-значение. Ключом является пара, состоящая из группы, которой принадлежит элемент, и любой группы матрицы $M$, а значением – сам элемент вместе с номерами строки и столбца.

- **Функция Reduce.** Операция редукции, соответствующая ключу $(i, j)$, где $i$ – группа из $M$, а $j$ – группа из $N$, получает список значений, состоящий из всех элементов в $i$-й полосе $M$ и $j$-й полосе $N$. Следовательно, у нее есть все значения, необходимые для вычисления элементов $P$, находящихся в одной из строк, составляющих $i$-ю полосу $M$, и в одном из столбцов, составляющих $j$-ю полосу $N$. Так, на рис. 2.11 показано, что редукция (3,4), получающая комбинацию третьей группы $M$ и четвертой группы $N$, вычисляет квадратный блок $P$.

Каждая операция редукция получает $n(n/g)$ элементов из каждой матрицы, поэтому $q = 2n^2/g$. Коэффициент репликации равен $g$, поскольку каждый элемент каждой матрицы отправляет $g$ операциям редукции. Следовательно, $r = g$. Объединяя $r = g$ и $q = 2n^2/g$, заключаем, что $r = 2n^2/q$. Таким образом, для соединения по сходству коэффициент репликации обратно пропорционален размеру редукции.

Оказывается, что эта верхняя граница коэффициента репликации одновременно является и нижней. То есть мы не можем добиться лучшего результата, чем позволяет семейство алгоритмов, описанное выше для одного прохода MapReduce. Но интересно, и мы увидим это ниже, что общий объем передаваемых данных можно уменьшить при том же размере редукции, если использовать два прохода MapReduce, как в разделе 2.3.9. Мы не будем приводить полный вывод нижней границы, но упомянем некоторые важные моменты доказательства.

На шаге (1) мы хотим получить верхнюю границу количества выходов, покрываемых редукцией размера $q$. Прежде всего, отметим, что если операция редукции получает некоторые, но не все элементы одной строки $M$, то элементы этой строки бесполезны: редукция не может породить ни одного элемента такой строки $P$. Аналогично, если редукция получает некоторые, но не все элементы одного столбца $N$, то эти входы бесполезны. Следовательно, можно предположить, что оптимальная схема сопоставления отправляет каждой операции редукции какое-то количество полных строк $M$ и полных столбцов $N$. Тогда эта операция редукции сможет породить выходной элемент $p_{ik}$ тогда и только тогда, когда получит всю $i$-ю строку $M$ и весь $k$-й столбец $N$. Далее на шаге (1) мы должны доказать, что наибольшее количество выходов покрывается, когда операция редукции получает одинаковое количество строк и столбцов. Оставляем это читателю в качестве упражнения.

Однако если редукция получает $k$ строк $M$ и $k$ столбцов $N$, то $q = 2nk$ и покрывается $k^2$ выходов. Следовательно, $g(q)$ – максимальное число выходов, покрываемых редукцией, получившей $q$ входов, – равно $q^2/4n^2$.

На шаге (2) мы знаем, что число выходов равно $n^2$. На шаге (3) мы замечаем, что если существует $k$ операций редукции, причем $i$-я операция получает $q_i \leq q$ входов, то:

$$\sum_{i=1}^{k} q_i^2 / 4n^2 \geq n^2$$

или

$$\sum_{i=1}^{k} q_i^2 \geq 4n^4.$$

Из этого неравенства вытекает, что

$$r \geq 2n^2/q.$$

Оставляем алгебраические преобразования, похожие на рассмотренные в примере 2.17, в качестве упражнения.

Теперь рассмотрим обобщение двухпроходного алгоритма умножения матриц, который был описан в разделе 2.3.9. Этот алгоритм тоже можно обобщить, так что в первом задании MapReduce будут использоваться операции редукции размера больше 2. Идея показана на рис. 2.12. Мы можем разбить строки и столбцы обеих входных матриц $M$ и $N$ на $g$ групп по $n/g$ строк или столбцов в каждой. Пересечения этих групп разбивают каждую матрицы на $g^2$ квадратов по $n^2/g^2$ элементов в каждом.

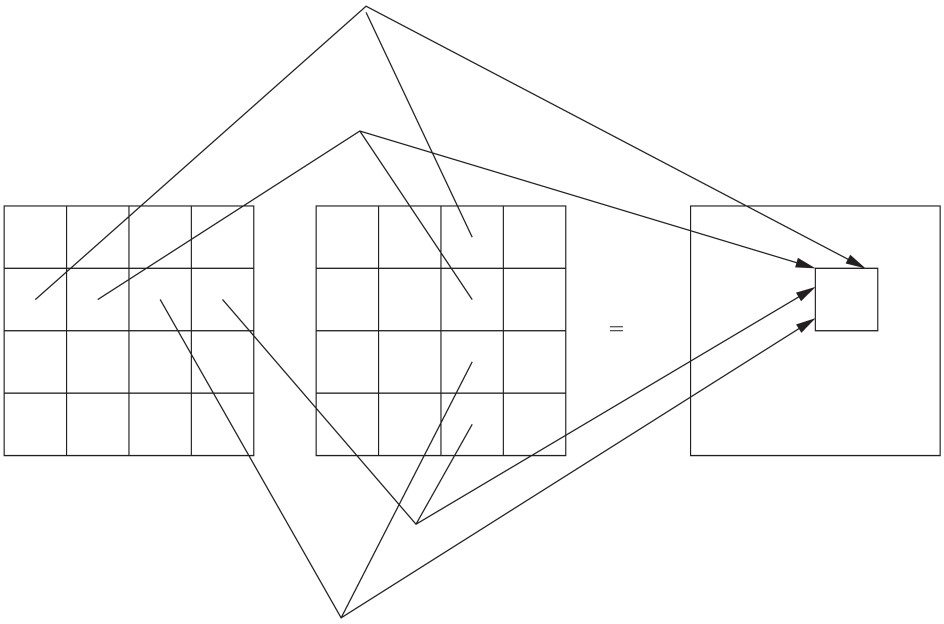



**Рис. 2.12.** Разбиение матриц на квадраты в двухпроходном алгоритме MapReduce

Квадрат $M$, соответствующий множеству строк $I$ и множеству столбцов $J$, комбинируется с квадратом $N$, соответствующий множеству строк $J$ и множеству столбцов $K$. Эти два квадрата вычисляют некоторые члены, необходимые для порождения квадрата выходной матрицы $P$, образованного множеством строк $I$ и множеством столбцов $K$. Однако эти два квадрата не вычисляют элементы $P$ до конца, они лишь порождают часть суммы. Вклад в тот же самый квадрат $P$ вносят и другие пары квадратов из $M$ и $N$. Эти вклады показаны на рис. 2.12. Мы видим,

что все квадраты $M$ с одним и тем же множеством строк $I$ комбинируются со всеми квадратами $N$ с одним и тем же множеством столбцов $K$, а множество $J$ может при этом варьироваться.

Таким образом, на первом проходе мы вычисляем произведения квадрата $(I, J)$ матрицы $M$ и квадрата $(J, K)$ матрицы $N$ для всех $I, J, K$. Затем, на втором проходе, мы для всех $I$ и $K$ суммируем эти произведения по всем возможным множествам $J$. Точнее, первое задание MapReduce делает следующее.

- **Функция Map.** Ключами являются тройки, состоящие из номеров строк и (или) столбцов $(I, J, K)$. Пусть элемент $m_{ij}$ принадлежит группе строк $I$ и группе столбцов $J$. Тогда из $m_{ij}$ мы генерируем $g$ пар ключ-значение, в которых значение содержит сам элемент $m_{ij}$, а также номера его строки и столбца $i$ и $j$, позволяющие идентифицировать элемент матрицы. Существует ровно одна пара ключ-значение для каждого ключа $(I, J, K)$, где $K$ может быть любой из $g$ групп столбцов $N$. Аналогично из элемента $n_{jk}$ матрицы $N$, где $j$ принадлежит группе $J$, а $k$ – группе $K$, функция Map генерирует $g$ пар ключ-значение, в которых значение содержит $n_{jk}$, $j$ и $k$, а ключи равны $(I, J, K)$ для любой группы $I$.
- **Функция Reduce.** Операция редукции, соответствующая ключу $(I, J, K)$, получает на входе все элементы $m_{ij}$, где $i$ принадлежит $I$, $j$ принадлежит $J$, а также все элементы $n_{jk}$, где $j$ принадлежит $J$, $k$ принадлежит $K$. Она вычисляет сумму

$$x_{iJk} = \sum_{j \in J} m_{ij} n_{jk}$$

по всем $i$ из $I$, $k$ из $K$.

Отметим, что коэффициент репликации для первого задания MapReduce равен $g$, поэтому общий объем передаваемых данных равен $2gn^2$. Еще отметим, что каждая операция редукции получает $2n^2/g^2$ входов, так что $q = 2n^2/g^2$, откуда $g = n\sqrt{2/q}$. Следовательно, общий объем передаваемых данных $2gn^2$ можно выразить через $q$ в виде $2\sqrt{2}n^3/\sqrt{q}$.

Второе задание MapReduce проще; оно суммирует элементы $x_{ijk}$ по всем множествам $J$.

- **Функция Map.** Мы предполагаем, что задачи-распределители выполняются на тех же узлах, на которых выполнялись задачи-редукторы из предыдущего задания. Поэтому передавать данные между ними не нужно. Функция Map принимает на входе один элемент $x_{ijk}$, и мы считаем, что предшествующие операции редукции пометили его числами $i$ и $k$, так что мы знаем, в какой элемент матрицы $P$ этот член вносит вклад. Генерируется одна пара ключ-значение с ключом $(i, k)$ и значением $x_{ijk}$.
- **Функция Reduce.** Эта функция просто суммирует значения, ассоциированные с ключом $(i, k)$, вычисляя тем самым выходной элемент $P_{ik}$.

Объем данных, передаваемых между распределителями и редукторами второго задания, равен $gn^2$, поскольку существует $n$ возможных значений $i$, $n$ возможных

значений $k$ и $g$ возможных значений множества $J$, и каждый элемент $x_{ijk}$ передается ровно один раз. Вспомним, что в ходе анализа первого задания MapReduce мы установили, что $g = n\sqrt{2/q}$, и, значит, объем передаваемых данных для второго задания можно записать в виде $n^2g = \sqrt{2}n^3/\sqrt{q}$. Эта величина ровно вдвое меньше объема данных, передаваемых в первом задании, поэтому общий объем передаваемых данных в двухпроходном алгоритме равен $3\sqrt{2}n^3/\sqrt{q}$. Оказывается (хотя мы не станем здесь останавливаться на этом подробнее), что можно еще немного уменьшить эту величину, если разбивать матрицы $M$ и $N$ не на квадратные, а на прямоугольные блоки, у которых одна сторона вдвое больше другой. В таком случае вместо константы $3\sqrt{2} = 4{,}24$ получается 4, так что мы имеем двухпроходный алгоритм с объемом передаваемых данных $4n^3/\sqrt{q}$.

Теперь вспомним, что коэффициент репликации для однопроходного алгоритма равен $4n^4/q$. Можно предполагать, что $q$ меньше $n^2$, иначе мы просто воспользовались бы последовательным алгоритмом в одном вычислительном узле и не применяли бы MapReduce вовсе. Следовательно, $n^3/\sqrt{q}$ меньше $n^4/q$, и если $q$ близко к своему минимально возможному значению $2n$,[11] то двухпроходный алгоритм оказывается лучше однопроходного в $O(\sqrt{n})$ раз с точки зрения объема передаваемых данных. И можно ожидать, что такое различие в объеме заметно скажется на стоимости. Оба алгоритма выполняют $O(n^3)$ арифметических операций. Естественно, что для двухпроходного алгоритма накладные расходы на управление больше. С другой стороны, на втором проходе двухпроходного алгоритма применяется ассоциативная и коммутативная функция Reduce, следовательно, можно будет уменьшить коммуникационную стоимость, воспользовавшись комбинатором.

## 2.6.8. Упражнения к разделу 2.6

**Упражнение 2.6.1.** Опишите графы, решающие следующие проблемы:

(*а*) Умножение матрицы размерности $n \times n$ на вектор длины $n$.

(*б*) Естественное соединение $R(A,B)$ и $S(B,C)$, где размеры областей определения $A$, $B$, $C$ равны $a$, $b$ и $c$ соответственно.

(*в*) Группировка и агрегирование для отношения $R(A,B)$, где $A$ – группировочный атрибут, а $B$ агрегируется операцией MAX. Предполагается, что размеры областей определения $A$ и $B$ равны $a$ и $b$ соответственно.

**! Упражнение 2.6.2.** Восполните детали доказательства того, что для однопроходного алгоритма умножения матриц коэффициент репликации $r$ должен быть не меньше $2n^2/q$, придерживаясь следующего плана:

(*а*) Докажите, что при фиксированном размере редукции максимальное количество выходов покрывается редукцией, когда она получает одинаковое количество строк $M$ и столбцов $N$.

(*б*) Алгебраическому преобразованию подвергните неравенство $\sum_{i=1}^{k} q_i^2 \geq 4n^4$.

[11] Если $q$ меньше $2n$, то операция редукции не может получить даже одну строку и один столбец, а потому не сможет вычислить выход.

!! **Упражнение 2.6.3.** Предположим, что входами являются битовые строки длины $b$, а выходы соответствуют парам строк с расстоянием Хэмминга 1.[12]

(*а*) Докажите, что редукция размера $q$ может покрыть не более $(q/2)\log_2 q$ выходов.

(*б*) Используя часть (*а*), покажите, что имеет место следующая оценка нижней границы коэффициента репликации: $r \geq b/\log_2 q$.

(*в*) Покажите, что существуют алгоритмы с таким коэффициентом репликации для случаев $q = 2$, $q = 2^b$ и $q = 2^{b/2}$.

# 2.7. Резюме

- *Кластерные вычисления.* Распространенной архитектурой для очень крупных приложений является кластер вычислительных узлов (процессор, оперативная память и диск). Вычислительные узлы монтируются в стойках, узлы в одной стойке обычно связаны между собой гигабитной сетью Ethernet. Сами стойки также связаны высокоскоростной сетью или коммутатором.
- *Распределенные файловые системы.* Архитектура для очень больших файловых систем была разработана недавно. Файлы состоят из порций размером порядка 64 МБ, и каждая порция несколько раз реплицируется в различных узлах или стойках.
- *MapReduce.* Эта система программирования позволяет задействовать параллелизм, свойственный кластерным вычислениям и справляться с аппаратными отказами, которые могут происходить в течение длительного вычисления на многих узлах. Главный процесс (мастер) управляет большим числом задач-распределителей и редукторов. Задачи, которые работали на отказавшем узле, перезапускаются мастером.
- *Функция Map.* Эту функцию пишет пользователь. Она принимает коллекцию входных объектов и преобразует каждый в нуль или более пар ключ-значение. Ключи могут повторяться.
- *Функция Reduce.* Система программирования MapReduce сортирует пары ключ-значение, порожденные всеми задачами-распределителями, собирает все значения, ассоциированные с данным ключом, в список и раздает пары ключ-список задачам-редукторам. Редуктор каким-то образом обрабатывает элементы каждого списка, применяя функцию Reduce, написанную пользователем. Результаты, порожденные всеми редукторами, образуют выход процесса MapReduce.
- *Операции редукции.* Часто удобно называть применение функции Reduce к одному ключу и ассоциированному с ним списку значений «редукцией».

[12] Говорят, что *расстояние Хэмминга* между битовыми строками равно 1, если они различаются ровно в одной позиции. Более общее определение приведено в разделе 3.5.6.

- *Hadoop*. Это реализация с открытым исходным кодом распределенной файловой системы (HDFS, или Hadoop Distributed File System) и технологии MapReduce (собственно Hadoop). Ее распространяет фонд Apache Foundation.
- *Управление отказами вычислительных узлов*. Системы MapReduce поддерживают перезапуск задач, завершившихся аварийно из-за отказа соответствующего узла или стойки, содержащей этот узел. Поскольку выход распределителей и редукторов становится доступен только после успешного завершения, открывается возможность перезапустить сбойную задачу, не заботясь о последствиях двойной обработки. Перезапускать все задание необходимо, только в случае отказа узла, на котором работает мастер.
- *Приложения MapReduce*. Не все параллельные алгоритмы пригодны для реализации в системе MapReduce, но существуют простые реализации умножения матрицы на вектор и умножения матриц. Кроме того, основные операторы реляционной алгебры также легко реализуются с помощью MapReduce.
- *Системы потоков работ*. Идея MapReduce обобщается на системы, поддерживающие ациклический граф функций, каждая из которых может выполняться любым числом задач, отвечающих за применение функции к части данных.
- *Рекурсивные потоки работ*. При реализации рекурсивной коллекции функций не всегда возможно сохранить возможность перезапуска любой сбойной задачи, поскольку рекурсивные задачи могли уже породить выход, потребленный какой-то другой задачей до сбоя. Были предложены различные схемы создания контрольной точки всего вычисления с целью перезапуска отдельных задач или всех задач с последней контрольной точки.
- *Коммуникационная стоимость*. Во многих приложениях MapReduce и подобных систем каждая задача делает что-то очень простое. Поэтому полная стоимость работы определяется стоимостью транспортировки данных из места создания в место использования. В таких случаях эффективность алгоритма MapReduce можно оценить, подсчитав суммарный размер входов всех задач.
- *Многопутевое соединение*. Иногда более эффективно реплицировать кортежи отношений, участвующих в соединении, и вычислять соединение трех и более отношений в одном задании MapReduce. Для оптимизации степени репликации каждого соединяемого отношения можно применить метод множителей Лагранжа.
- *Звездообразное соединение*. В аналитических запросах часто участвует очень большая таблица фактов, соединяемая с гораздо меньшими размерными таблицами. Такое соединение очень эффективно выполняется с применением многопутевой техники. Альтернатива – распределить таблицу фактов и на постоянной основе реплицировать размерные таблицы, при-

меняя ту же стратегию, что при многопутевом соединении таблицы фактов со всеми размерными таблицами.

- *Коэффициент репликации и размер редукции*. Часто удобно измерять объем передаваемых данных с помощью коэффициента репликации, определенного как объем коммуникаций в расчете на один вход. Размером редукции называется максимальное число входов, ассоциированных с любой операцией редукции. Для многих проблем можно получить нижнюю границу коэффициента репликации в виде функции от размера редукции.
- *Представление проблем в виде графов*. Многие проблемы, решаемые с помощью MapReduce, можно представить в виде графа, вершины которого соответствуют входам и выходам. Выход соединен со всеми входами, которые необходимы для его вычисления.
- *Схемы сопоставления*. Если дан граф проблемы и размер редукции, то схемой сопоставления называется назначение входов одной или нескольким операциям редукции, так что ни одной операции не назначено больше входов, чем допускает размер редукции, но при этом для каждого выхода существует редукция, которая получает все входы, необходимые для вычисления этого выхода. Требование о наличии схемы сопоставления для любого алгоритма MapReduce – хороший способ выразить отличие алгоритмов MapReduce от параллельных вычислений общего вида.
- *Умножение матриц с помощью MapReduce*. Существует семейство однопроходных алгоритмов MapReduce, которые выполняют умножение матриц размерности $n \times n$ с минимально возможным коэффициентом репликации $r = 2n^2/q$, где $q$ – размер редукции. С другой стороны, двухпроходный алгоритм MapReduce, решающий ту же проблему с таким же размером редукции, может обойтись в $n$ раз меньшим объемом передаваемых данных.

# 2.8. Список литературы

GFS, файловая система Google, описана в работе [10]. Реализации MapReduce компании Google посвящена статья [8]. Сведения о Hadoop и HDFS можно найти в [11]. Дополнительную информацию об отношениях и реляционной алгебре смотрите в [16].

Система Clustera рассмотрена в [9], а Hyracks (прежнее название – Hyrax) – в [4]. Система Dryad [13] имеет похожие возможности, но требует, чтобы параллельные задачи создавал пользователь. Эта обязанность была автоматизирована с появлением DryadLINQ [17]. Обсуждение кластерной реализации рекурсии см. в [1]. Система Pregel описана статье [14].

Другой подход к рекурсии принят в системе Haloop [5]. Там рекурсия рассматривается как итерация, причем выход одного раунда подается на вход следующего. Эффективность достигается путем управления местом размещения промежуточных данных и задач, реализующих каждый раунд.

Существует ряд других систем, построенных поверх распределенной файловой системы и (или) MapReduce, которые мы здесь не рассматривали, но знать о которых полезно. В работе [6] описана система BigTable, написанная в Google реализация объектного хранилища очень большого размера. В несколько ином направлении развивалась система Pnuts [7] в компании Yahoo!. Например, она поддерживает ограниченную форму транзакционной обработки.

PIG [15] – реализация реляционной алгебры поверх Hadoop. Наконец, Hive [12] реализует ограниченный вариант SQL поверх Hadoop.

Модель коммуникационной стоимости для алгоритмов MapReduce и оптимальные реализации многопутевого соединения взяты из работы [3]. Материал о коэффициенте репликации, размере редукции и их взаимосвязях см. в [2]. Там же можно найти решения упражнений 2.6.2 и 2.6.3.

1. F. N. Afrati, V. Borkar, M. Carey, A. Polyzotis, J.D. Ullman «Cluster computing, recursion, and Datalog», Proc. Datalog 2.0 Workshop, Elsevier, 2011.
2. F. N. Afrati, A. Das Sarma, S. Salihoglu, J.D. Ullman «Upper and lower bounds on the cost of a MapReduce computation», Proc. Intl. Conf. on Very Large Databases, 2013. Доступно также в архиве CoRR, abs/1206.4377.
3. F. N. Afrati, J.D. Ullman «Optimizing joins in a MapReduce environment», Proc. Thirteenth Intl. Conf. on Extending Database Technology, 2010.
4. V. Borkar, M. Carey «Hyrax: demonstrating a new foundation for data-parallel computation», `http://asterix.ics.uci.edu/pub/hyraxdemo.pdf`, Univ. of California, Irvine, 2010.
5. Y. Bu, B. Howe, M. Balazinska, M. Ernst «HaLoop: efficient iterative data processing on large clusters», Proc. Intl. Conf. on Very Large Databases, 2010.
6. F. Chang, J. Dean, S. Ghemawat,W.C. Hsieh, D.A.Wallach, M. Burrows, T. Chandra, A. Fikes, R. E. Gruber «Bigtable: a distributed storage system for structured data», ACM Transactions on Computer Systems 26:2, pp. 1–26, 2008.
7. B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, R. Yerneni «Pnuts: Yahoo!'s hosted data serving platform», PVLDB 1:2, pp. 1277–1288, 2008.
8. J. Dean, S. Ghemawat «Mapreduce: simplified data processing on large clusters», Comm. ACM 51:1, pp. 107–113, 2008.
9. D. J. DeWitt, E. Paulson, E. Robinson, J.F. Naughton, J. Royalty, S. Shankar, A. Krioukov «Clustera: an integrated computation and data management system», PVLDB 1:1, pp. 28–41, 2008.
10. S. Ghemawat, H. Gobioff, S.-T. Leung «The Google file system», 19th ACM Symposium on Operating Systems Principles, 2003.
11. `hadoop.apache.org`, Apache Foundation.
12. `hadoop.apache.org/hive`, Apache Foundation.
13. M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly «Dryad: distributed data-parallel programs from sequential building blocks», Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, pp. 59–72, ACM, 2007.

14. G. Malewicz, M.N. Austern, A.J.C. Sik, J. C. Denhert, H. Horn, N. Leiser, G. Czajkowski «Pregel: a system for large-scale graph processing», Proc. ACM SIGMOD Conference, 2010.
15. C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins «Pig latin: a not-so-foreign language for data processing», Proc. ACM SIGMOD Conference, pp. 1099–1110, 2008.
16. J. D. Ullman, J. Widom «A First Course in Database Systems», Third Edition, Prentice-Hall, Upper Saddle River, NJ, 2008.
17. Y. Yu, M. Isard, D. Fetterly, M. Budiu, I. Erlingsson, P.K. Gunda, J. Currey «DryadLINQ: a system for general-purpose distributed dataparallel computing using a high-level language», OSDI, pp. 1–14, USENIX Association, 2008.

# ГЛАВА 3.
# Поиск похожих объектов

Фундаментальная проблема добычи данных – поиск «похожих» объектов. В разделе 3.1 мы рассмотрим некоторые приложения, но в качестве примера прямо сейчас назовем отыскание почти повторяющихся страниц в коллекции веб-страниц. Это может быть, к примеру, плагиат или зеркала с почти одинаковым содержимым, отличающиеся только информацией о сервере и других зеркалах.

Начнем с постановки задачи о сходстве – найти множества с достаточно большим пересечением. Мы покажем, как задачу поиска текстуально похожих документов можно свести к задаче о множествах с помощью метода разбиения на «шинглы». Затем мы познакомимся с алгоритмом MinHash, который сжимает большие множества так, что остается возможность установить сходство исходных множеств по их сжатым вариантам. В разделе 3.9 рассматриваются другие методы, работающие, когда требуется очень высокая степень сходства.

При поиске похожих объектов любого рода возникает еще одна серьезная проблема: пар объектов может оказаться слишком много, чтобы проверить их все на сходство, даже если вычислить сходство любой пары очень просто. Для решения этой проблемы разработана техника «хэширования с учетом близости», позволяющая сконцентрировать внимание на парах, которые с наибольшей вероятностью окажутся похожими.

Наконец, мы поговорим о таких видах «сходства», которые невозможно выразить в терминах пересечения множеств. Это приведет нас к теории метрик в произвольных пространствах. И станет побудительным мотивом для разработки общей теории хэширования с учетом близости, применимой и к другим определениям «сходства».

## 3.1. Приложения поиска близкого соседям

Сначала мы сосредоточимся на конкретном понятии «сходства»: сходство множеств, оцениваемое по относительному размеру их пересечения. Это так называемое «сходство по Жаккару», которое мы изучим в разделе 3.1.1. Затем мы исследуем некоторые применения поиска похожих множеств: нахождение текстуально

похожих документов и коллаборативная фильтрация, заключающаяся в поиске похожих клиентов и похожих товаров. Чтобы преобразовать задачу о текстуальном сходстве документов в задачу о пересечении множеств, нам понадобится техника разбиения на «шинглы», описанная в разделе 3.2.

## 3.1.1. Сходство множеств по Жаккару

*Коэффициентом Жаккара* двух множеств $S$ и $T$ называется величина $|S \cap T| / |S \cup T|$, т. е. отношение размера их пересечения к размеру объединения. Будем обозначать коэффициент Жаккара множеств $S$ и $T$ $\text{SIM}(S, T)$.

**Пример 3.1.** На рис. 3.1 изображены два множества $S$ и $T$. Их пересечение состоит из трех элементов, а объединение – из восьми. Следовательно $\text{SIM}(S, T) = 3/8$.

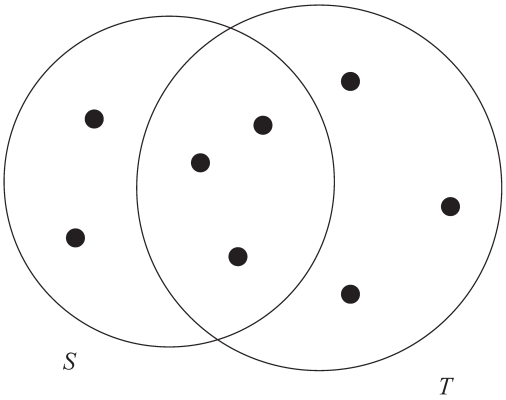



**Рис. 3.1.** Два множества с коэффициентом Жаккара 3/8

## 3.1.2. Сходство документов

Важный класс задач, для которых хорошо подходит сходство по Жаккару, – поиск текстуально похожих документов в большом корпусе, например, вебе или наборе новостей. Следует понимать, что мы здесь говорим о сходстве на уровне символов, а не «смысловой схожести», для установления которой необходимо анализировать встречающиеся в документах слова и их использование. Эта проблема тоже интересна, но решается другими методами, о которых мы кратко упомянули в разделе 1.3.1. Однако и у текстуального сходства есть важные применения и, прежде всего, поиск дубликатов или почти дубликатов. Сразу отметим, что проверить, являются ли два документа точными копиями, просто; нужно лишь сравнить их посимвольно и, если хотя бы в одном символе они различаются, значит, это не дубликаты. Но во многих приложениях встречаются документы, которые, хотя и не идентичны, но содержат большие одинаковые куски. Приведем несколько примеров.

### Плагиат

Поиск незаконно заимствованных документов – это проверка нашей способности обнаруживать текстуальное сходство. Плагиатор может включить части чужого текста в свой собственный, изменив при этом несколько слов или порядок предложений. И, тем не менее, получившийся документ будет содержать 50 % оригинала или даже больше. Простой процесс посимвольного сравнения документов не выявит завуалированный плагиат.

### Зеркальные страницы

Часто важные или особо популярные веб-сайты дублируются на нескольких серверах с целью разделения нагрузки. Страницы таких сайтов-зеркал очень похожи, но редко бывают идентичными. Например, страница может содержать информацию, относящуюся к серверу, на котором размещена, или ссылки на все остальные зеркала, кроме данного. Сходная проблема – индивидуализация страниц для различных учебных курсов. На таких страницах могут быть заметки к курсу, домашние задания и слайды. У похожих страниц могут различаться название курса и год. Кроме того, от года к году в страницы могут вноситься мелкие изменения. Умение обнаруживать такого рода похожие страницы важно, поскольку поисковая система считается более качественной, если опускает ссылки, которые почти не отличаются от присутствующих на первой странице результатов.

### Статьи из одного источника

Часто бывает, что репортер пишет статью, которая направляется, скажем через агентство Associated Press, в редакции разных газет, а те публикуют его на своих сайтах. Каждая газета немного изменяет исходную статью: вырезает часть абзацев или даже добавляет свой материал. Скорее всего, статья будет окружена логотипом издания, рекламными объявлениями и ссылками на другие статьи на том же сайте. Но ядром страницы останется исходная статья. Новостные агрегаторы, например Google News, пытаются найти все варианты такой статьи и показать только один, а для решения этой задачи нужно уметь определять, что две веб-страницы текстуально похожи, хотя и не идентичны.[1]

## *3.1.3. Коллаборативная фильтрация как задача о сходстве множеств*

Еще один класс приложений, в которых сходство множеств играет важную роль, называется *коллаборативной фильтрацией*. Именно так мы рекомендуем пользователям предметы, которые нравятся другим пользователям со схожими вкусами. Мы подробно изучим коллаборативную фильтрацию в разделе 9.3, а пока рассмотрим несколько расхожих примеров.

---

[1] Агрегирование новостей подразумевает также поиск статей на одну и ту же тему, пусть даже текстуально не похожих. Эту задачу тоже можно свести к поиску по сходству, но коэффициента Жаккара для ее решения недостаточно.

## Покупки в интернет-магазинах

У сайта Amazon.com миллионы клиентов и продает он миллионы товаров. В его базе данных хранится информация о том, что покупал каждый клиент. Мы можем назвать двух клиентов похожими, если коэффициент Жаккара для множеств их покупок высок. Аналогично два товара, множества покупателей которых имеют высокий коэффициент Жаккара, можно считать похожими. Отметим, однако, что для сайтов-зеркал коэффициент Жаккара может достигать 90 %, но маловероятно, что столь же высокое значение будет у любых двух покупателей (если только они не купили всего один товар). Даже значение 20 % достаточно необычно, чтобы заподозрить покупателей с одинаковыми вкусами. То же относится и к товарам: коэффициент Жаккара будет значимым, даже если он не очень высок.

Для коллаборативной фильтрации необходимо несколько инструментов, помимо поиска похожих покупателей или товаров, мы обсудим это в главе 9. Например, два клиента Amazon, любящих научную фантастику, возможно, купили много книг на эту тему, но одинаковых среди них раз-два и обчелся. Однако, сочетая поиск похожих объектов с кластеризацией (глава 7), мы сможем обнаружить, что книги фантастического жанра похожи друг на друга, и поместить их в одну группу. Таким образом, мы можем получить более полное представление о сходстве клиентов, задавая вопрос, покупали ли они товары из одинаковых групп.

## Оценка фильмов

Компания NetFlix хранит информацию о том, какие фильмы заказывал каждый ее клиент, а также оценки, которые клиенты выставляли фильмам. Мы можем считать два фильма похожими, если их заказывали или высоко оценивали одни и те же клиенты, а двух клиентов считать похожими, если они заказывали или высоко оценивали одни и те же фильмы. К этой ситуации применимы те же наблюдения, что для Amazon: коэффициент сходства является значимым, даже если он не особенно велик, а кластеризация фильмов по жанрам упростит дело.

Когда данные содержат оценки, а не просто бинарные решения (купил – не купил или понравилось – не понравилось), уже нельзя представлять клиентов или товары просто множествами. Вот некоторые альтернативы.

1. Игнорировать пары клиент-фильм с низкими оценками, т. е. считать, что клиент вообще не смотрел этот фильм.
2. При сравнении клиентов считать, что с клиентом ассоциировано множество, в котором каждый фильм представлен элементом с двумя значениями: «хорошо» и «плохо». Если клиент высоко оценил фильм, записать в множество этого клиента для этого фильма значение «хорошо», а если низко – то «плохо». Затем эти множества можно сравнить с помощью коэффициента Жаккара. Аналогичный прием годится для сравнения фильмов.
3. Если оценка – это число звездочек от 1 до 5, то помещать фильм в множество клиента $n$ раз, если клиент поставил ему $n$ звездочек. Затем при измерении сходства клиентов использовать коэффициент Жаккара для мультимножеств. Для мультимножеств $B$ и $C$ коэффициент Жаккара опре-

деляется путем включения элемента в пересечение $n$ раз, где $n$ – минимум из двух чисел: количество вхождений в $B$ и количество вхождений в $C$. При подсчете числа вхождений элемента в объединение мы складываем количества его вхождений в $B$ и $C$.[2]

**Пример 3.2.** Коэффициент Жаккара для мультимножеств {*a*, *a*, *a*, *b*} и {*a*, *a*, *b*, *b*, *c*} равен 1/3. В пересечении *a* считается дважды, а *b* один раз, поэтому его размер равен 3. Размер объединения двух мультимножеств всегда равен сумме их размеров, т. е. в данном случае 9. Т. к. максимально возможный коэффициент Жаккара для мультимножеств равен 1/2, то величина 1/3 означает, что мультимножества очень похожи, что и понятно из визуального сравнения их содержимого.

### 3.1.4. Упражнения к разделу 3.1

**Упражнение 3.1.1.** Вычислить коэффициент Жаккара для каждой пары следующих трех множеств: {1, 2, 3, 4}, {2, 3, 5, 7}, {2, 4, 6}.

**Упражнение 3.1.2**. Вычислить коэффициент Жаккара для каждой пары следующих трех мультимножеств: {1, 1, 1, 2}, {1, 1, 2, 2, 3}, {1, 2, 3, 4}.

**!! Упражнение 3.1.3**. Пусть имеется универсальное множество $U$ с $n$ элементами, и мы случайным образом выбираем из него два подмножества $S$ и $T$ по $m$ элементов в каждом. Каково математическое ожидание коэффициента Жаккара $S$ и $T$?

## 3.2. Разбиение документов на шинглы

Для идентификации лексически похожих документов самый эффективный способ представить документ в виде множества заключается в том, чтобы построить множество коротких строк, встречающихся в документе. В таком случае для документов, в которых встречаются общие короткие фрагменты, например предложения или даже фразы, в соответствующих множествах будет много общих элементов, даже если порядок предложений в исходных документах различается. В этом разделе мы познакомимся с простейшим и наиболее употребительным подходом – разбиением на шинглы, – а также с одной его интересной модификацией.

---

[2] Хотя обычно (например, в стандарте SQL) при объединении мультимножеств учитывается сумма количеств вхождений в каждое мультимножество, это определение вступает в некоторое противоречие со смыслом коэффициента Жаккара. При таком определении максимальный коэффициент Жаккара равен 1/2, а не 1, поскольку в объединении мультимножеств с самим собой оказывается в два раза больше элементов, чем в его пересечении с самим собой. Если мы предпочитаем, чтобы коэффициент Жаккара для мультимножества и его копии был равен 1, то можем переопределить объединение мультимножеств, так чтобы количество вхождений каждого элемента было равно максимуму из количеств его вхождений в каждое мультимножество. При таком изменении коэффициент Жаккара не всегда оказывается ровно вдвое больше, но оценка сходства получается разумной.

### 3.2.1. k-шинглы

Документ – это строка символов. Определим $k$-шингл документа как любую встречающуюся в нем подстроку длины $k$. Тогда с каждым документом можно ассоциировать множество $k$-шинглов, встречающихся в нем хотя бы один раз.

> **Пример 3.3.** Пусть документом $D$ является строка *abcdabd* и $k = 2$. Тогда множеством 2-шинглов для $D$ будет множество {*ab*, *bc*, *cd*, *da*, *bd*}.
>
> Отметим, что подстрока *ab* дважды встречается в $D$, но только один раз входит в множество шинглов. Вариантом разбиения на шинглы является формирование не множества, а мультимножества, в которое каждый шингл входит столько раз, сколько встречается в документе. Мы, однако, использовать мультимножества шинглов не будем.

Существует несколько подходов к обращению с пробельными символами (пробел, знаки табуляции, перехода на новую строку и т. д.). Иногда имеет смысл заменять последовательность одного и более пробельных символов одним пробелом. Тогда мы сможем отличить шинглы, покрывающие два и более слов, от всех прочих.

> **Пример 3.4.** Если мы возьмем $k = 9$, но полностью выбросим пробелы, то увидим некоторое лексическое сходство между предложениями «`The plane was ready for touch down`» и «`The quarterback scored a touchdown`». Если пробелы сохранены, то в первом предложении будут шинглы `touch dow` и `ouch down`, а во втором – `touchdown`. А после выбрасывания в обоих будет один и тот же шингл `touchdown`.

### 3.2.2. Выбор размера шингла

В качестве $k$ можно взять любую константу. Но если $k$ слишком мало, то в большинстве документов мы обнаружим значительную часть возможных последовательностей $k$ символов. В таком случае множества шинглов для двух документов могут иметь высокий коэффициент Жаккара, хотя в самих документах нет ни одного общего предложения или хотя бы связной фразы. В предельном случае, когда $k = 1$, у большинства веб-страниц почти все символы будут общими, а отличаться они будут всего несколькими символами, так что почти все страницы окажутся очень похожи.

Насколько большим должно быть значение $k$, зависит от длины типичных документов и размера множества типичных символов. Важно запомнить следующее:

- $k$ следует выбирать настолько большим, чтобы вероятность появления любого наперед заданного шингла в любом документе была мала.

Таким образом, если корпусом документов являются сообщения электронной почты, то $k = 5$ – неплохой выбор. Чтобы понять, почему это так, предположим, что сообщения состоят только из букв и пробельных символов (хотя на практике встречаются почти все символы ASCII, имеющие графическое начертание). Но тогда существует $27^5 = 14\,348\,907$ возможных шинглов. Поскольку типичное по-

чтовое сообщение содержит гораздо меньше 14 миллионов символов, то следует ожидать, что $k = 5$ даст удовлетворительный результат. И так оно и есть.

Однако в реальности вычисление не столь тривиально. Конечно, в сообщениях встречается больше 27 символов. Но вероятности вхождения символов различны. Доминируют часто употребляемые буквы и пробелы, тогда как «z» и другие буквы, за которые в игре «Скрэббл» дают много очков, встречаются гораздо реже. Так что даже в коротких сообщениях будет много 5-шинглов, состоящих из часто встречающихся букв, поэтому вероятность того, что в совсем разных сообщениях будет много общих шинглов такого вида, больше, чем показывает приведенное выше вычисление. Есть хорошее эвристическое правило: представьте, что существует только 20 символов и оценивайте количество $k$-шинглов, как $20^k$. Для больших документов, например научных статей, считается безопасным значение $k = 9$.

## 3.2.3. Хэширование шинглов

Вместо того чтобы использовать в качестве шинглов сами подстроки, мы можем взять хэш-функцию, которая распределяет строки длины $k$ по какому-то количеству ячеек, и считать шинглом номер ячейки. Тогда документ представляется множеством целых чисел – хэш-кодов одного или нескольких встречающихся в нем $k$-шинглов. Например, можно построить множество 9-шинглов для документа, а затем сопоставить каждому шинглу номер ячейки в диапазоне от 0 до $2^{32} - 1$. Тогда каждый шингл будет представлен четырьми байтами вместо девяти. Мало того что при этом уменьшается объем данных, так еще к этим (хэшированным) шинглам можно применять машинные команды, работающие со словами.

Отметим, что качество различения будет лучше, если использовать 9-шинглы и сворачивать их в четыре байта, чем если изначально использовать 4-шинглы, хотя объем памяти, занимаемый шинглом, в обоих случаях один и тот же. О причине мы говорили в разделе 3.2.2. Если использовать 4-шинглы, то большинство последовательностей из четырех байтов встречаются в типичных документах редко или вообще не встречаются. Следовательно, количество различных шинглов в действительности гораздо меньше $2^{32} - 1$. Если, как в разделе 3.2.2, предположить, что в английском тексте часто встречаются только 20 символов, то число различных вероятных 4-шинглов равно только $20^4 = 160\,000$. Тогда как вероятных 9-шинглов будет гораздо больше $2^{32}$. Если свернуть каждый из них в четыре байта, то можно ожидать, что встретятся почти все возможные последовательности из четырех байтов. Мы говорили об этом в разделе 1.3.2.

## 3.2.4. Шинглы, построенные из слов

Существует альтернативная форма шингла, доказавшая эффективность при решении задачи идентификации похожих новостей, упоминавшейся в разделе 3.1.2. Важной отличительной особенностью этой задачи является тот факт, что текст новости по стилю отличается от окружающих ее материалов на странице. В новостях, да и в большинстве написанных человеком текстов встречается много стоп-слов (см. разделе 1.3.1) – таких, как «and», «you», «to» и т. п. Во многих прило-

жениях желательно игнорировать стоп-слова, потому что они не несут полезной информации о статье, в частности, о ее тематике.

Однако обнаружилось, что в задаче о поиске похожих новостей, шинглы, образованные стоп-словом и двумя следующими за ним словами – все равно, являются они стоп-словами или нет, – образуют полезное множество. Достоинство этого подхода в том, что текст новости внесет больше шинглов в множество, представляющее веб-страницу, чем окружающие новость элементы. Напомним, что наша задача – найти страницы, содержащие одну и ту же новость, какие бы элементы ее ни окружали. Поскольку мы повышаем вес шинглов из статьи, можно ожидать, что коэффициент Жаккара для страниц, содержащих ту же новость, но другие окружающие материалы, будет выше, чем для страниц с такими же окружающими материалами, но другой новостью.

**Пример 3.5.** Рекламное объявление может содержать простой текст, например «Buy Sudzo» (Купи Sudzo). Но новость, содержащая ту же самую мысль, скорее, будет выглядеть так: «`A spokesperson for the Sudzo Corporation revealed today that studies have shown it is good for people to buy Sudzo products`» (Представитель корпорации Sudzo сегодня рассказал, что согласно проведенным исследованиям людям полезно покупать продукцию Sudzo). Курсивом мы выделили все вероятные стоп-слова, хотя не существует общепринятого мнения о том, что считать стоп-словами. Вот первые три шингла, образованные из стоп-слова и двух следующих за ним слов:

```
A spokesperson for
for the Sudzo
the Sudzo Corporation
```

Из этого предложения можно составить девять шинглов, а из рекламного объявления – ни одного.

## 3.2.5. Упражнения к разделу 3.2

**Упражнение 3.2.1.** Выпишите первые десять 3-шинглов из первого предложения в разделе 3.2. (По-английски это предложение выглядит так: «The most effective way to represent documents as sets, for the purpose of identifying lexically similar documents is to construct from the document the set of short strings that appear within it».)

**Упражнение 3.2.2.** Если мы решим использовать шинглы на основе стоп-слов, как описано в разделе 3.2.4, и примем, что стоп-словами являются все слова, содержащие не больше трех букв, то каковы тогда будет шинглы из первого предложения в разделе 3.2?

**Упражнение 3.2.3.** Каково максимальное возможное количество $k$-шинглов в документе, содержащем $n$ байтов? Можете предполагать, что размер алфавита достаточно велик, так что число возможных строк длины $k$ не меньше $n$.

# 3.3. Сигнатуры множеств с сохранением сходства

Множества шинглов велики по размеру. Даже если хэшировать каждый шингл в четыре байта, все равно память, занимаемая этим множеством, примерно в четыре раза больше самого документа. Если документов миллионы, то все шинглы могут и не поместиться в оперативной памяти[3].

Наша цель в этом разделе – заменить большие множества значительно меньшими, называемыми «сигнатурами». Сигнатуры должны обладать важным свойством: их можно сравнивать и, зная только сигнатуры, давать оценку коэффициента Жаккара исходных множеств. Конечно, нельзя требовать, чтобы сигнатуры точно отражали сходство множеств, из которых получены, но они дают близкие оценки и чем длиннее сигнатура, тем точнее оценка. Например, если заменить множества хэшированных шинглов размером 200 000 байтов, полученные из документов размером 50 000 байтов, сигнатурами длиной 1000 байтов, то обычно расхождение в оценках составит всего несколько процентов.

## 3.3.1. Матричное представление множеств

Прежде чем объяснять, как построить небольшую сигнатуру большого множества, полезно наглядно представить коллекцию множеств в виде *характеристической матрицы*. Столбцы матрицы соответствуют множествам, а строки – элементам универсального множества, из которого берутся элементы всех множеств. На пересечении строки $r$ и столбца $c$ находится 1, если элемент, соответствующий строке $r$, входит в множество, соответствующее столбцу $c$. В противном случае в позиции $(r, c)$ находится 0.

**Пример 3.6.** На рис. 3.2 показан пример матрицы, представляющей множества с элементами из универсального множества $\{a, b, c, d, e\}$. Здесь $S_1 = \{a, d\}$, $S_2 = \{c\}$, $S_3 = \{b, d, e\}$ и $S_4 = \{a, c, d\}$. Верхняя строка и левый столбец не являются частью матрицы, а просто напоминают, что именно в ней представлено.

| *Элемент* | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| *a* | 1 | 0 | 0 | 1 |
| *b* | 0 | 0 | 1 | 0 |
| *c* | 0 | 1 | 0 | 1 |
| *d* | 1 | 0 | 1 | 1 |
| *e* | 0 | 0 | 1 | 0 |

**Рис. 3.2.** Матрица, представляющая четыре множества

[3] Существует еще одна серьезная опасность: даже если все множества шинглов помещаются в памяти, количество пар может оказаться настолько велико, что мы не сможем вычислить сходство для каждой из них. Решение этой проблемы мы отложим до раздела 3.4.

Важно помнить, что характеристическая матрица применяется не для хранения данных, а для их визуализации. Одна из причин не хранить данные в таком виде состоит в том, что на практике эти матрицы почти всегда разрежены (нулей в них гораздо больше, чем единиц). Можно сэкономить память, если представлять разреженную матрицу из 0 и 1 позициями, в которых находятся единицы. Другая причина – тот факт, что данные обычно хранятся в каком-то другом формате, подходящем для определенных целей.

Например, если строки – товары, а столбцы – покупатели, представленные множеством купленных ими товаров, то эти данные хранятся в таблице базы данных. Кортеж этой таблицы, вероятно, содержит товар, покупателя и какие-то дополнительные сведения о покупке, например дату и номер банковской карты.

## 3.3.2. Минхэш

Мы собираемся построить для множества сигнатуру, получающуюся в результате большого количества вычислений, скажем сотни, каждое из которых вычисляет «минхэш» характеристической матрицы. В этом разделе мы узнаем, как вычисляется минхэш в принципе, а в последующих покажем, как можно получить аппроксимацию минхэша, приемлемую для практических целей.

Чтобы вычислить *минхэш* множества, представленного столбцом характеристической матрицы, выберем некую перестановку строк. Значение минхэша любого столбца – это номер первой строки, в порядке перестановки, в которой в этом столбце встречается единица.

**Пример 3.7.** Пусть для матрицы на рис. 3.2 выбран порядок строк *beadc*. Эта перестановка определяет минхэш-функцию $h$, которая отображает множества на строки. Вычислим значение минхэша множества $S_1$ в соответствии с $h$. Первый столбец, соответствующий множеству $S_1$, содержит 0 в строке $b$, поэтому мы переходим к строке $e$, второй в порядке перестановки. И в этой строке столбец, соответствующий множеству $S_1$, содержит 0, поэтому переходим к строке $a$, где наконец находим 1. Следовательно, $h(S_1) = a$.

| *Элемент* | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| *b* | 0 | 0 | 1 | 0 |
| *e* | 0 | 0 | 1 | 0 |
| *a* | 1 | 0 | 0 | 1 |
| *d* | 1 | 0 | 1 | 1 |
| *c* | 0 | 1 | 0 | 1 |

**Рис. 3.3.** Перестановка строк таблицы, показанной на рис. 3.2

Хотя невозможно физически переставить строки очень большой характеристической матрицы, минхэш-функция $h$ неявно переупорядочивает строки матрицы на рис. 3.2, преобразуя ее в матрицу на рис. 3.3. В этой матрице мы можем прочитать значения $h$, просматривая столбцы сверху вниз, пока не наткнемся на 1. Таким образом, мы видим, что $h(S_2) = c$, $h(S_3) = b$, $h(S_4) = a$.

### 3.3.3. Минхэш и коэффициент Жаккара

Существует примечательная связь между минхэшем и коэффициентом Жаккара тех множеств, для которых вычислен минхэш.

Вероятность того, что минхэш-функция, соответствующая случайной перестановке строк, порождает одно и то же значение для двух множеств, равна коэффициенту Жаккара этих множеств.

Чтобы понять, почему это так, нужно нарисовать столбцы для этих двух множеств. Если ограничиться столбцами для $S_1$ и $S_2$, то строки можно отнести к одному из трех классов:

1. В строках типа $X$ в обоих столбцах находится 1.
2. В строках типа $Y$ в одном столбце находится 1, а в другом 0.
3. В строках типа $Z$ в обоих столбцах находится 0.

Поскольку матрица разреженная, большинство строк имеют тип $Z$. Однако именно отношением числа строк типа $X$ и типа $Y$ определяется как $\text{SIM}(S_1, S_2)$, так и вероятность того, что $h(S_1) = h(S_2)$. Пусть $x$ – число строк типа $X$, а $y$ – число строк типа $Y$. Тогда $\text{SIM}(S_1, S_2) = x/(x + y)$. Объясняется это тем, что $x$ размер пересечения $S_1 \cap S_2$, а $x + y$ – размер объединения $S_1 \cup S_2$.

Теперь рассмотрим вероятность того, что $h(S_1) = h(S_2)$. Если представить себе, что строки переставлены в случайном порядке и просматривать матрицу сверху вниз, то вероятность встретить строку типа $X$ раньше строки типа $Y$ равна $x/(x + y)$. Но если первая сверху строка типа, отличного от $Z$, имеет тип $X$, то, конечно, $h(S_1) = h(S_2)$. С другой стороны, если первая встретившаяся строка типа, отличного от $Z$, имеет тип $Y$, то множество, для которого в этой строке находится 1, получает эту строку в качестве значения минхэша. Однако минхэш того множества, для которого в этой строке находится 0, окажется в заведомо более низкой строке переставленного списка. Таким образом, мы знаем, что, если первой встретилась строка типа $Y$, то $h(S_1) \neq h(S_2)$. Следовательно, вероятность того, что $h(S_1) = h(S_2)$, равна $x/(x + y)$, а это также значение коэффициента Жаккара множеств $S_1$ и $S_2$.

### 3.3.4. Минхэш-сигнатуры

Снова рассмотрим коллекцию множеств, представленную своей характеристической матрицей $M$. Чтобы представить множества, мы случайным образом выбираем какое-то число $n$ перестановок строк $M$. Быть может, будет достаточно одной или нескольких сотен перестановок. Назовем минхэш-функции, определяемые этими перестановками $h_1, h_2, ..., h_n$. Из столбца, представляющего множество $S$, построим *минхэш-сигнатуру* $S$ – вектор $[h_1(S), h_2(S), ..., h_n(S)]$. Обычно этот список хэш-кодов представляется в виде столбца. Следовательно, мы можем по матрице $M$ сформировать *матрицу сигнатур*, в которой $i$-й столбец $M$ заменен минхэш-сигнатурой множества в $i$-ом столбце.

Отметим, что в матрице сигнатур столько же столбцов, сколько в матрице $M$, но строк только $n$. Даже если $M$ представлена не явно, а в сжатом виде, подходящем

для разреженной матрицы (например, позициями единиц), все равно матрица сигнатур будет гораздо меньше $M$.

## 3.3.5. Вычисление минхэш-сигнатур

Переставить строки большой характеристической матрицы на практике невозможно. Даже на выбор случайной перестановки миллионов или миллиардов строк уходит много времени, а последующая сортировка строк займет еще больше. Поэтому матрицы с переставленными строками, как на рис. 3.3, хоть концептуально и выглядят весьма привлекательно, на деле нереализуемы.

По счастью, можно смоделировать эффект случайной перестановки с помощью случайно выбранной хэш-функции, которая отображает номера строк на столько ячеек, сколько строк в матрице. Хэш-функция, отображающая числа 0, 1, ..., $k - 1$ на номера ячеек от 0 до $k - 1$, обычно помещает несколько чисел в одну и ту же ячейку, оставляя некоторые ячейки пустыми. Однако это несущественно, если $k$ велико, а коллизий не слишком много. Мы можем принять, что наша хэш-функция $h$ «переставляет» строку $r$ в позицию $h(r)$.

Итак, вместо того чтобы выбирать $n$ случайных перестановок строк, мы случайно выберем $n$ хэш-функций $h_1, h_2, ..., h_n$, применяемых к строкам. А матрицу сигнатур построим, рассматривая каждую строку в заданном порядке. Пусть $\text{SIG}(i, c)$ – элемент матрицы сигнатур для $i$-й хэш-функции и столбца $c$. Первоначально установим $\text{SIG}(i, c)$ в $\infty$ для всех $i$ и $c$. Строка $r$ обрабатывается следующим образом:

1. Вычислить $h_1(r), h_2(r), ..., h_n(r)$.
2. Для каждого столбца $c$ выполнить:
   (*а*) Если на пересечении $c$ и $r$ находится 0, ничего не делать.
   (*б*) Если же на пересечении $c$ и $r$ находится 1, то для каждого $i = 1, 2, ..., n$ присвоить $\text{SIG}(i, c)$ минимум из текущего значения $\text{SIG}(i, c)$ и $h_i(r)$.

| *Строка* | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $x$+1 mod 5 | 3$x$+1 mod 5 |
|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 2 | 4 |
| 2 | 0 | 1 | 0 | 1 | 3 | 2 |
| 3 | 1 | 0 | 1 | 1 | 4 | 0 |
| 4 | 0 | 0 | 1 | 0 | 0 | 3 |

**Рис. 3.4.** Вычисление хэш-функций для матрицы на рис. 3.2

**Пример 3.8.** Снова рассмотрим характеристическую матрицу на рис. 3.2, которую мы воспроизводим вместе с дополнительными данными на рис. 3.4. Мы заменили буквенные обозначения строк целыми числами от 0 до 4. Мы также выбрали две хэш-функции: $h_1(x) = x+1 \bmod 5$ и $h_2(x) = 3x+1 \bmod 5$. Результаты вычисления этих функций для номеров строк, приведены в последних двух столбцах на рис. 3.4. Отметим, что эти простые хэш-функции дают настоящие перестановки строк, но это возможно только потому, что

количество строк, 5 – простое число. В общем случае будут коллизии, когда двум строкам соответствует одинаковый хэш-код.

Теперь смоделируем алгоритм вычисления матрицы сигнатур. Первоначально все элементы матрицы равны ∞:

| | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| $h_1$ | ∞ | ∞ | ∞ | ∞ |
| $h_2$ | ∞ | ∞ | ∞ | ∞ |

Сначала рассмотрим строку 0 на рис. 3.4. Мы видим, что оба значения $h_1(0)$ и $h_2(0)$ равны 1. В строке 0 единицы находятся в столбцах, соответствующих множествам $S_1$ и $S_4$, поэтому только эти столбцы матрицы сигнатур могут измениться. Поскольку 1 меньше ∞, то мы заменяем оба значения в столбцах, соответствующих $S_1$ и $S_4$. Текущая оценка матрицы сигнатур выглядит так:

| | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| $h_1$ | 1 | ∞ | ∞ | 1 |
| $h_2$ | 1 | ∞ | ∞ | 1 |

Затем переходим к строке с номером 1 на рис. 3.4. В ней единица есть только в столбце $S_3$, а хэш-коды для нее равны $h_1(1) = 2$ и $h_2(1) = 4$. Таким образом, устанавливаем SIG(1, 3) равным 2, а SIG(2, 3) равным 4. Все остальные сигнатуры остаются без изменения, потому что в соответствующих им столбцах строки 1 находятся нули. Вот как выглядит новая матрица сигнатур:

| | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| $h_1$ | 1 | ∞ | 2 | 1 |
| $h_2$ | 1 | ∞ | 4 | 1 |

В строке с номером 2 на рис. 3.4 единицы находятся в столбцах $S_2$ и $S_2$, а ее хэш-коды равны $h_1(2) = 3$ и $h_2(2) = 2$. Мы могли бы изменить значения сигнатур для $S_4$, но в этом столбце обе сигнатуры, [1, 1], меньше соответствующих хэш-кодов, [3, 2]. Однако значения в столбце $S_2$ все еще равны ∞, поэтому заменим их на [3, 2]. В результате получаем:

| | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| $h_1$ | 1 | 3 | 2 | 1 |
| $h_2$ | 1 | 2 | 4 | 1 |

Следующей на очереди строка с номером 3. Здесь во всех столбцах, кроме $S_2$, находятся единицы, а хэш-коды равны $h_1(3) = 4$ и $h_2(3) = 0$. Значение $h_1$ больше тех, что уже записаны во все столбцы матрицы сигнатур, поэтому не меняем ни одно значение в первой строке. Но значение $h_2 = 0$ меньше уже записанных, поэтому переустанавливаем в 0 элементы SIG(2, 1), SIG(2, 3) и SIG(2, 4). Отметим, что уменьшать SIG(2, 2) нельзя, т. к. в столбце $S_2$ матрицы

на рис. 3.4 на пересечении с рассматриваемой строкой стоит 0. Вот новая матрица сигнатур:

| | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| $h_1$ | 1 | 3 | 2 | 1 |
| $h_2$ | 0 | 2 | 0 | 0 |

Наконец, рассмотрим строку с номером 4. Имеем $h_1(4) = 0$ и $h_2(4) = 3$. Поскольку в строке 4 единица стоит только на пересечении со столбцом $S_3$, то сравниваем только столбец текущей матрицы сигнатур для этого множества [2, 0] с хэш-кодами [0, 3]. Т. к. $0 < 2$, то меняем SIG(1, 3) на 0, но, поскольку $3 > 0$, то SIG(2, 3) не меняем. Окончательная матрица сигнатур имеет вид:

| | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| $h_1$ | 1 | 3 | 0 | 1 |
| $h_2$ | 0 | 2 | 0 | 0 |

На основе этой матрицы мы можем оценить коэффициенты Жаккара исходных множеств. Заметим, что столбцы 1 и 4 одинаковы, поэтому можно предположить, что SIM($S_1$, $S_4$) = 1.0. Взглянув на рис. 3.4, убеждаемся, что истинный коэффициент Жаккара для $S_1$ и $S_4$ равен 2/3. Напомним, что доля строк, в которых значения сигнатур совпадают, дает лишь оценку истинного коэффициента Жаккара, а этот пример слишком мал, чтобы начал действовать закон больших чисел, поэтому ожидать близости оценок не приходится. Продолжая рассмотрение, мы видим, что столбцы сигнатур $S_1$ и $S_3$ совпадают в половине строк (истинный коэффициент равен 1/4), тогда как для $S_1$ и $S_2$ сигнатуры дают оценку 0 (и это правильное значение).

## 3.3.6. Упражнения к разделу 3.3

**Упражнение 3.3.1.** Проверьте теорему из раздела 3.3.3, которая связывает коэффициент Жаккара с вероятностью равенства минхэшей, в частном случае матрицы на рис. 3.2.

(*а*) Вычислите коэффициент Жаккара для каждой пары столбцов матрицы на рис. 3.2.

**!** (*б*) Для каждой пары столбцов на том же рисунке вычислите, в скольких из 120 возможных перестановок строк два столбца хэшируются в одно и то же значение.

**Упражнение 3.3.2.** Используя данные на рис. 3.4, добавьте в состав сигнатур столбцов значения следующих хэш-функций:

(*а*) $h_3(x) = 2x + 4 \bmod 5$.
(*b*) $h_4(x) = 3x - 1 \bmod 5$.

**Упражнение 3.3.3.** В матрице на рис. 3.5 шесть строк.

(*а*) Вычислите минхэш-сигнатуры для каждого столбца при следующих хэш-функциях $h_1(x) = 2x + 1 \bmod 6$; $h_2(x) = 3x + 2 \bmod 6$; $h_3(x) = 5x + 2 \bmod 6$.

(*б*) Какие из этих хэш-функций дают истинные перестановки?

(*в*) Насколько близки оценки коэффициента Жаккара для шести пар столбцов к истинным значениям?

| *Строка* | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 2 | 1 | 0 | 0 | 1 |
| 3 | 0 | 0 | 1 | 0 |
| 4 | 0 | 0 | 1 | 1 |
| 5 | 1 | 0 | 0 | 0 |

**Рис. 3.5.** Матрица для упражнения 3.3.3

**! Упражнение 3.3.4.** Теперь, когда мы знаем о связи коэффициента Жаккара с вероятностью совпадения минхэшей двух множеств, вернемся к упражнению 3.1.3. Можете ли вы, воспользовавшись этой связью, упростить задачу вычисления математического ожидания коэффициента Жаккара двух случайно выбранных множеств?

**! Упражнение 3.3.5.** Докажите, что если коэффициент Жаккара двух столбцов равен 0, то вычисление минхэшей всегда дает правильную его оценку.

**!! Упражнение 3.3.6.** Можно ожидать, что для вычисления оценки коэффициента Жаккара необязательно использовать все возможные перестановки строк. Например, можно было бы использовать только циклические перестановки, т. е. начать с произвольно выбранной строки $r$, которая станет первой по порядку, за ней расположить строки $r + 1$, $r + 2$ и т. д. до последней строки, затем продолжить с первой строки вплоть до строки $r - 1$. Существует ровно $n$ таких перестановок $n$ строк. Однако их недостаточно для корректной оценки коэффициента Жаккара. Приведите пример матрицы с двумя столбцами, для которой усреднение по всем циклическим перестановкам не дает коэффициент Жаккара.

**! Упражнение 3.3.7.** Допустим, мы хотим использовать технологию MapReduce для вычисления минхэш-сигнатур. Если матрица хранится в виде порций, соответствующих некоторым столбцам, то распараллелить вычисление очень просто. Каждый распределитель получает некоторые столбцы и все хэш-функции и вычисляет минхэш-сигнатуры полученных столбцов. Предположим, однако, что матрица разбита на порции по строкам, так что распределитель получает хэш-функции и множество строк. Придумайте функции Map и Reduce, которые позволяют задействовать MapReduce при таком представлении данных.

# 3.4. Хэширование документов с учетом близости

Хотя мы можем использовать минхэши для сжатия больших документов в меньшие по размеру сигнатуры с сохранением ожидаемого сходства любой пары документов, все равно может оказаться невозможно эффективно искать пары документов с наибольшим сходством. Причина в том, что количество пар может быть слишком большим, хотя документов не так много.

> **Пример 3.9.** Допустим, что имеется миллион документов, и мы используем сигнатуры длины 250. Тогда сигнатура каждого документа будет занимать 1000 байтов, и для хранения всех данных понадобится меньше одного гигабайта – меньше размера оперативной памяти типичного ноутбука. Однако число пар документов равно $\binom{1000000}{2}$, т. е. полтриллиона. Если на вычисление сходства двух сигнатур уходит 1 микросекунда, то для вычисления сходства всех пар на ноутбуке потребуется почти шесть дней.

Если наша цель – вычислить сходство каждой пары, то для сокращения объема работы сделать ничего нельзя, хотя распараллеливание и может уменьшить время. Но часто мы всего лишь хотим найти самые похожие пары или все пары, для которых сходство превышает заданный порог. Тогда нужно сосредоточить внимание только на парах, которых могут быть похожими, и не проверять каждую пару. Существует общая теория, как это сделать, она называется *хэширование с учетом близости* (locality-sensitive hashing, LSH), или *поиск близкого соседа* (near-neighbor search). В этом разделе мы рассмотрим одну из форму LSH, предназначенную для решения конкретной задачи: поиск среди документов, представленных множествами шинглов, для которых затем вычислены минхэш-сигнатуры. В разделе 3.6 мы опишем общую теорию хэширования с учетом близости, а также ряд ее приложений и родственных методов.

## 3.4.1. LSH для минхэш-сигнатур

Один из общих подходов к LSH – «хэшировать» объекты несколько раз, так чтобы вероятность попадания похожих элементов в одну ячейку оказалась больше, чем для непохожих. Все пары, попавшие в одну ячейку, мы считаем *кандидатами*. Только пары-кандидаты проверяются на сходство. Мы надеемся, что большинство непохожих пар никогда не попадут в одну ячейку и потому не будут проверяться. Те непохожие пары, которые все же оказались в одной ячейке, называются *ложноположительными результатами*; мы надеемся, что их будет относительно немного. Мы также надеемся, что большинство похожих пар будут помещены в одну и ту же ячейку хотя бы одной хэш-функцией. Те, для которых это не так, называются *ложноотрицательными результатами*; мы надеемся, что среди действительно похожих пар их будет мало.

Если мы уже имеем минхэш-сигнатуры объектов, то выбрать способ хэширования можно следующим образом. Разобьем матрицу сигнатур на $b$ полос по $r$ строк в каждой. Для каждой полосы существует хэш-функция, которая принимает $r$ целых чисел (часть одного столбца, находящуюся в этой полосе) и хэширует их в большое количество ячеек. Мы будем использовать одну и ту же хэш-функцию для всех полос, но возьмем отдельный массив ячеек для каждой полосы, так чтобы столбцы, которым соответствуют одинаковые векторы в разных полосах, не оказались в одной ячейке.

**Пример 3.10.** На рис. 3.6 показана часть матрицы сигнатур, содержащей 12 строк, разбитых на четыре полосы по три строки в каждой. Во втором и четвертом из показанных столбцов в первой полосе находится вектор [0, 2, 1], поэтому они обязательно попадут в одну ячейку при хэшировании первой полосы. Следовательно, эти столбцы составляют пару-кандидат независимо от того, как они выглядят в других полосах. Вполне возможно, что и другие столбцы, например первые два из показанных, попадут в ту же ячейку при хэшировании первой полосы. Но поскольку их частичные векторы, [1, 3, 0] и [0, 2, 1], различны, а ячеек при каждом хэшировании много, мы ожидаем, что вероятность случайной коллизии очень мала. При нормальных обстоятельствах можно предполагать, что два вектора попадают в одну ячейку тогда и только тогда, когда они одинаковы.

У двух столбцов, не совпадающих в первой полосе, есть еще три возможности стать парой-кандидатом: они могут совпасть в любой из трех других полос.

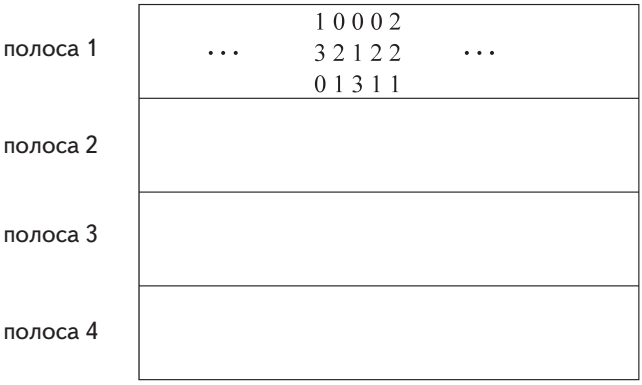



**Рис. 3.6.** Разбиение матрицы сигнатур на четыре полосы по три строки в каждой

Заметим, однако, что чем более похожи два столбца, тем вероятнее, что они совпадут в некоторой полосе. Поэтому интуитивно представляется, что при такой стратегии разбиения на полосы пары-кандидаты с гораздо большей вероятностью будут образованы из похожих столбцов, чем из непохожих.

## 3.4.2. Анализ метода разбиения на полосы

Пусть используется $b$ полос по $r$ строк в каждой и предположим, что для некоторой пары документов коэффициент Жаккара равен $s$. Как было показано в разделе 3.3.3, вероятность того, что минхэш-сигнатуры этих документов совпадают в любой наперед заданной строке матрицы сигнатур, равна $s$. Вычислить вероятность того, что эти документы (точнее, их сигнатуры) образуют пару-кандидат, можно следующим образом.

1. Вероятность того, что сигнатуры совпадают во всех строках одной наперед заданной полосы, равна $s^r$.
2. Вероятность того, что сигнатуры не совпадают по крайней мере в одной строке одной наперед заданной полосы, равна $1 - s^r$.
3. Вероятность того, что сигнатуры не совпадают по крайней мере в одной строке любой полосы, равна $(1 - s^r)^b$.
4. Вероятность того, что сигнатуры совпадают во всех строках по крайней мере одной полосы и потому образуют пару-кандидат, равна $1 - (1 - s^r)^b$.

Как ни странно, при любом выборе констант $b$ и $r$ график этой функции имеет *S-образную* форму, показанную на рис. 3.7. *Порог*, т. е. значение коэффициента Жаккара $s$, при котором вероятность стать кандидатом, равна 1/2, является функцией от $b$ и $r$. Грубо говоря, порог – это такое значение, при котором крутизна кривой максимальна, и для больших $b$ и $r$ мы видим, что пары, для которых сходство превышает пороговое значение, с очень большой вероятностью оказываются кандидатами, тогда как у пары, для которой сходство меньше порога, шансы стать кандидатом, невелики. А это как раз то, чего мы добивались.

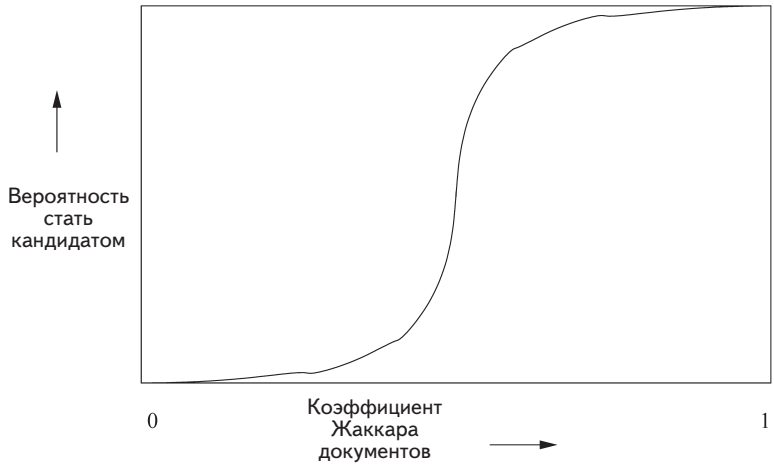



**Рис. 3.7.** *S*-кривая

Приближенное значение порога равно $(1/b)^{1/r}$. Например, если $b = 16$ и $r = 4$, то порог достигается приблизительно при $s = 1/2$, т. к. корень четвертой степени из 1/16 равен 1/2.

**Пример 3.11.** Рассмотрим случай $b = 20$, $r = 5$. То есть предположим, что имеется матрица сигнатур длины 100, разбитая на 20 полос по 5 строк в каждой. В таблице на рис. 3.8 приведено несколько значений функции $1 - (1 - s^5)^{20}$. Отметим, что пороговое значение $s$, при котором кривая достигает половины своей высоты, чуть меньше 0.5. Отметим также, что эта кривая не является идеальной ступенчатой функцией, которая меняет значение с 0 на 1 при пороговом значении аргумента, однако угловой коэффициент кривой в средней точке велик. Так, на отрезке от $s = 0.4$ до $s = 0.6$ она растет больше, чем на 0.6, поэтому угловой коэффициент в средней точке больше 3.

| $s$ | $1 - (1 - s^r)^b$ |
|---|---|
| 0.2 | 0.006 |
| 0.3 | 0.047 |
| 0.4 | 0.186 |
| 0.5 | 0.470 |
| 0.6 | 0.802 |
| 0.7 | 0.975 |
| 0.8 | 0.9996 |

**Рис. 3.8.** Значения $S$-кривой для $b = 20$, $r = 5$

Например, при $s = 0.8$ величина $1 - (0.8)^5$ составляет приблизительно 0.672. Если возвести это число в двадцатую степень, получим примерно 0.00035. Вычитание из единицы дает 0.99965. Таким образом, если взять два документа с коэффициентом сходства 80 %, то для любой наперед заданной полосы есть лишь примерно 33 % шансов на то, что части их столбцов совпадут во всех пяти строках, и, следовательно, они станут парой-кандидатом. Однако полос 20 и, стало быть, 20 возможностей стать кандидатом. Из 3000 пар, похожих на 80 %, в среднем лишь одна не окажется кандидатом, т. е. даст ложноотрицательный результат.

## 3.4.3. Сочетание разных методов

Теперь мы можем описать, как искать пары-кандидаты похожих документов, а затем выявлять среди них действительно похожие. Следует подчеркнуть, что при таком подходе возможны ложноотрицательные результаты – пары похожих документов, которые не сочтены таковыми, потому что не стали парами-кандидатами. Возможны также ложноположительные результаты – некоторая пара сочтена кандидатом, но после детального анализа оказалось, что документы в ней недостаточно похожи.

1. Выберем значение $k$ и для каждого документа построим множество $k$-шинглов. Можно, хотя это и необязательно, хэшировать $k$-шинглы, заменив их более короткими номерами ячеек.
2. Отсортируем пары документ-шингл в порядке возрастания шингла.
3. Выберем длину $n$ минхэш-сигнатур. Подадим отсортированный список на вход алгоритма из раздела 3.3.5, вычисляющего минхэш-сигнатуры всех документов.

4. Выберем порог $t$, определяющий, насколько похожи должны быть документы, чтобы рассматривать их как потенциальную «похожую пару». Выберем количество полос $b$ и количество строк $r$, удовлетворяющие соотношению $br = n$, так чтобы порог $t$ был приблизительно равен $(1/b)^{1/r}$. Если важно избежать ложноотрицательных результатов, то следует выбирать $b$ и $r$, так чтобы это выражение было меньше $t$; если же важна скорость и желательно избежать ложноположительных результатов, выбирайте $b$ и $r$, так чтобы $(1/b)^{1/r}$ было больше порога.
5. Построим пары-кандидаты, применяя метод LSH из раздела 3.4.1.
6. Для каждой пары-кандидата проверим, что доля совпадающих элементов сигнатур не меньше $t$.
7. Факультативный шаг: если сигнатуры достаточно похожи, рассмотреть сами документы и проверить, действительно они похожи или случайно оказались похожи только их сигнатуры.

### 3.4.4. Упражнения к разделу 3.4

**Упражнение 3.4.1.** Вычислите $S$-кривую $1 - (1 - s^r)^b$ для $s = 0.1, 0.2, ..., 0.9$ при следующих значениях $r$ и $b$:

- $r = 3, b = 10$;
- $r = 6, b = 20$;
- $r = 5, b = 50$.

**! Упражнение 3.4.2.** Для каждой пары $(r, b)$ из упражнения 3.4.1 вычислите порог, т. е. значение $s$, при котором $1 - (1 - s^r)^b$ в точности равно $1/2$. Как это значение соотносится с аппроксимацией $(1/b)^{1/r}$, предложенной в разделе 3.4.2?

**! Упражнение 3.4.3.** Примените технику, описанную в разделе 1.3.5, для аппроксимации $S$-кривой $1 - (1 - s^r)^b$ при очень малых значениях $s^r$.

**! Упражнение 3.4.4.** Пусть мы хотим реализовать алгоритм LSH с помощью MapReduce. Точнее, предположим, что порции матрицы сигнатур – это столбцы, а элементами являются пары ключ-значение, в которых ключом служит номер столбца, а значением – сама сигнатура (т. е. вектор).

(*а*) Покажите, как породить ячейки для всех полос в качестве выхода одного процесса MapReduce. *Подсказка*: вспомните, что функция Map может порождать несколько пар ключ-значение из одного элемента.

(*б*) Покажите, как другой процесс MapReduce может преобразовать выход (*а*) в список пар, подлежащих сравнению. Точнее, для каждого столбца $i$ должен быть выведен список тех столбцов $j > i$, с которыми нужно сравнить $i$.

## 3.5. Метрики

Сделаем небольшое отступление и познакомимся с общим понятием меры расстояния, или метрики. Коэффициент Жаккара – является мерой близости множеств, хотя и не настоящей метрикой. То есть, чем ближе множества, тем больше

их коэффициент Жаккара. А вот 1 минус коэффициент Жаккара – действительно метрика, как мы вскоре увидим; она называется *расстоянием Жаккара*.

Однако расстояние Жаккара – не единственная разумная мера близости. В следующем разделе мы изучим другие метрики, нашедшие применение на практике. А затем, в разделе 3.6, увидим, что для некоторых из них также имеются методики LSH, позволяющие сосредоточиться на близких точках, не сравнивая между собой все. Другие приложения метрик будут рассмотрены в главе 7, посвященной кластеризации.

## 3.5.1. Определение метрики

Пусть имеется множество точек, называемое *пространством*. *Метрикой* в этом пространстве называется функция $d(x, y)$, которая принимает в качестве аргументов две точки, возвращает вещественное число и удовлетворяет следующим аксиомам.

1. $d(x, y) \geq 0$ (неотрицательность расстояния).
2. $d(x, y) = 0$ тогда и только тогда, когда $x = y$ (все расстояния положительны за исключением расстояния от точки до нее самой).
3. $d(x, y) = d(y, x)$ (симметричность).
4. $d(x, y) \leq d(x, z) + d(z, y)$ (*неравенство треугольника*).

Самым сложным условием является неравенство треугольника. Интуитивно оно означает, что невозможно получить выигрыш, добираясь из точки $x$ в точку $y$ через промежуточную точку $z$. Именно благодаря аксиоме неравенства треугольника все метрики ведут себя так, будто метрика описывает длину кратчайшего расстояния между двумя точками.

## 3.5.2. Евклидовы метрики

Самой знакомой метрикой является та, которую мы привычно называем «расстоянием». Точками *n-мерного евклидова пространства* являются векторы из $n$ вещественных чисел. Традиционно в этом пространстве используется метрика, называемая $L_2$*-нормой* и определенная следующим образом:

$$d([x_1, x_2, \ldots, x_n], [y_1, y_2, \ldots, y_n]) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Иначе говоря, мы вычисляем сумму квадратов расстояний по каждому измерению и извлекаем из нее квадратный корень.

Легко проверить, что первые три требования к метрике удовлетворяются. Евклидово расстояние между точками не может быть отрицательным, потому что имеется в виду положительное значение квадратного корня. Поскольку квадрат вещественного числа всегда неотрицателен, то если существует такое $i$, что $x_i \neq y_i$, то расстояние будет строго положительно. С другой стороны, если $x_i = y_i$ для всех $i$, то расстояние, очевидно, равно 0. Симметричность следует из того, что

$(x_i - y_i)^2 = (y_i - x_i)^2$. Для доказательства неравенства треугольника требуются нетривиальные алгебраические преобразования. Однако его можно интерпретировать как свойство евклидова пространства: сумма длин двух любых сторон треугольника не меньше длины третьей стороны.

Для евклидовых пространств существуют и другие метрики. Для любой константы $r$ можно определить *$L_r$-норму $d$* по следующей формуле:

$$d([x_1, x_2, \ldots, x_n], [y_1, y_2, \ldots, y_n]) = \left( \sum_{i=1}^{n} |x_i - y_i|^r \right)^{1/r}.$$

В случае $r = 2$ получаем обычную $L_2$-норму. Часто употребляется также $L_1$-норма, или *манхэттенское расстояние*, когда расстояние между двумя точками вычисляется как сумма абсолютных величин разностей координат по каждому измерению. Это расстояние называется «манхэттенским», потому что именно столько прошел бы человек, вынужденный добираться из одной точки в другой по линиям прямоугольной сетки, как в кварталах Манхэттена.

Еще одно интересное расстояние – $L_\infty$-норма, т. е. предел, к которому стремится значение $L_r$-нормы при $r$ стремящемся к бесконечности. С ростом $r$ значение приобретает только измерение, по которому расстояние максимально, поэтому формально $L_\infty$-норма определяется как максимум $|x_i - y_i|$ по всем $i$.

**Пример 3.12.** Рассмотрим двумерное евклидово пространство (обычную плоскость) и точки (2, 7) и (6, 4). $L_2$-норма дает расстояние $\sqrt{(2-6)^2 + (7-4)^2} = \sqrt{4^2 + 3^2} = 5$. $L_1$-норма дает расстояние $|2 - 6| + |7 - 4| = 4 + 3 = 7$. $L_\infty$-норма дает расстояние $\max(|2 - 6|, |7 - 4|) = \max(4, 3) = 4$.

## 3.5.3. Расстояние Жаккара

Как отмечалось в начале этого раздела, расстояние Жаккара между множествами $x$ и $y$ определяется как $d(x, y) = 1 - \text{SIM}(x, y)$, т. е. 1 минус отношение размера пересечения к размеру объединения. Мы должны проверить, что эта функция действительно определяет метрику.

1. $d(x, y)$ неотрицательно, поскольку размер пересечения заведомо не больше размера объединения.
2. $d(x, y) = 0$, если $x = y$, поскольку $x \cup x = x \cap x = x$. Однако если $x \neq y$, то размер $x \cap y$ строго меньше размера $x \cup y$, поэтому $d(x, y)$ строго положительно.
3. $d(x, y) = d(y, x)$, потому что операции пересечения и объединения симметричны, т. е. $x \cup y = y \cup x$ и $x \cap y = y \cap x$.
4. Что касается неравенства треугольника, вспомним (см. раздел 3.3.3), что $\text{SIM}(x, y)$ равно вероятности того, что случайно выбранная минхэш-функция отображает $x$ и $y$ в одно и то же значение. Следовательно, расстояние Жаккара $d(x, y)$ – вероятность того, что случайно выбранная минхэш-функ-

ция не отображает $x$ и $y$ в одно и то же значение. Значит, мы можем переформулировать условие $d(x, y) \le d(x, z) + d(z, y)$ следующим образом: если $h$ – случайно выбранная минхэш-функция, то вероятность того, что $h(x) \ne h(y)$, не больше суммы вероятностей того, что $h(x) \ne h(z)$, и того, что $h(z) \ne h(y)$. Но это утверждение справедливо, потому что коль скоро $h(x) \ne h(y)$, хотя бы одно из значений $h(x)$ и $h(y)$ должно быть отлично от $h(z)$. Оба не могут совпадать с $h(z)$, потому что тогда $h(x)$ и $h(y)$ совпадали бы.

## 3.5.4. Косинусное расстояние

*Косинусное расстояние* имеет смысл в пространствах, где определены измерения, в частности, в евклидовых пространствах или их дискретных вариантах, например, пространствах векторов с целочисленными или булевыми (0 или 1) элементами. Мы не различаем вектор и его произведение на скаляр. В таком случае косинусное расстояние между двумя точками – это угол между соответствующими им векторами. Этот угол изменяется от 0 до 180 градусов вне зависимости от числа измерений.

Для измерения косинусного расстояния мы можем сначала вычислить косинус угла, а затем взять арккосинус. Косинус угла между векторами $x$ и $y$ равен скалярному произведению $x \cdot y$, поделенному на произведение $L_2$-норм $x$ и $y$ (т. е. их евклидовых расстояний от начала координат). Напомним, что скалярное произведение векторов $[x_1, x_2, ..., x_n] \cdot [y_1, y_2, ..., y_n]$ равно $\sum_{i=1}^{k} x_i y_i$.

**Пример 3.13.** Возьмем два вектора $x = [1, 2, -1]$ и $y = [2, 1, 1]$. Скалярное произведение $x \cdot y$ равно $1 \times 2 + 2 \times 1 + (-1) \times 1 = 3$. $L_2$-норма обоих векторов равна $\sqrt{6}$. Например, $L_2$-норма $x$ равна $\sqrt{1^2 + 2^2 + (-1)^2} = \sqrt{6}$. Следовательно, косинус угла между $x$ и $y$ равен $3/(\sqrt{6}\sqrt{6}) = 1/2$. Если косинус угла равен 1/2, то сам угол – 60 градусов, это и есть косинусное расстояние между $x$ и $y$.

Мы должны доказать, что косинусное расстояние – действительно метрика. По определению, оно принимает значения от 0 до 180, так что отрицательным быть не может. Угол между двумя векторами равен 0 тогда и только тогда, когда их направление совпадает[4]. Симметричность очевидна: угол между $x$ и $y$ такой же, как между $y$ и $x$. Неравенство треугольника лучше всего проиллюстрировать физическим рассуждением. Чтобы совместить $x$ с $y$, можно сначала повернуть $x$ в $z$, а потом в $y$. Сумма углов этих поворотов не может быть меньше угла поворота $x$ сразу в $y$.

## 3.5.5. Редакционное расстояние

Это расстояние имеет смысл, когда в роли точек выступают строки. Расстоянием между двумя строками $x = x_1 x_2 ... x_n$ и $y = y_1 y_2 ... y_m$ называется наименьшее коли-

[4] Отметим, что для выполнения второй аксиомы мы должны считать, что векторы, отличающиеся только скалярным коэффициентом, определяют одно и то же направление, как оно и есть в действительности. В противном случае оказалось бы, что расстояние между разными векторами равно 0, что противоречит второй аксиоме, согласно которой нулю может быть равно только $d(x, x)$.

чество операций вставки и удаления одного символа, в результате которых $x$ превращается в $y$.

**Пример 3.14.** Редакционное расстояние между строками $x$ = `abcde` и $y$ = `acfdeg` равно 3. Для преобразования $x$ в $y$ нужно:

1. Удалить `b`.
2. Вставить `f` после `c`.
3. Вставить `g` после `e`.

Никакая последовательность из меньшего числа операций вставки и удаления не сможет преобразовать $x$ в $y$. Поэтому $d(x, y) = 3$.

Другой способ определить редакционное расстояние $d(x, y)$ – вычислить *наибольшую общую подпоследовательность* (LCS) $x$ и $y$. LCS строк $x$ и $y$ – это строка, которая строится путем удаления элементов из $x$ и $y$, и такая, что ее длина не меньше длины любой строки, которую можно построить таким способом. Редакционное расстояние $d(x, y)$ равно сумме длин $x$ и $y$ минус удвоенная длина их LCS.

**Пример 3.15.** Для строк $x$ = `abcde` и $y$ = `acfdeg` из примера 3.14 есть всего одна LCS – `acde`. Она самая длинная из возможных, потому что содержит все символы, общие для $x$ и $y$. По счастью, эти общие символы встречаются в обеих строках в одном и том же порядке, поэтому мы можем включить их все в LCS. Отметим, что длина $x$ равна 5, длина $y$ равна 6, а длина их LCS равна 4. Поэтому редакционное расстояние равно $5 + 6 - 2 \times 4 = 3$, что согласуется с прямым вычислением.

В качестве другого примера возьмем $x$ = `aba` и $y$ = `bab`. Редакционное расстояние между ними равно 2. Например, для преобразования $x$ в $y$ можно сначала удалить первое `a`, а затем вставить в конец `b`. Для этих строк существуют две LCS: `ab` и `ba`. Каждую можно получить удалением из строк по одному символу. Как и должно быть, когда для пары строк существует несколько LCS, длины обеих LCS равны. Поэтому редакционное расстояние вычисляется как $3 + 3 - 2 \times 2 = 2$.

Редакционное расстояние является метрикой. Естественно, оно не может быть отрицательно и лишь для одинаковых строк равно 0. Чтобы понять, почему редакционное расстояние симметрично, заметим, что последовательность операций вставки и удаления можно инвертировать: заменить каждую вставку удалением и наоборот. Неравенство треугольника также легко проверяется. Один из способов преобразовать строку $s$ в строку $t$ – сначала преобразовать $s$ в какую-то строку $u$, а затем преобразовать $u$ в $t$. Следовательно, количество операций редактирования, необходимых для перехода от $s$ к $u$, плюс количество операций, необходимых для перехода от $u$ к $t$, не может быть меньше наименьшего количества операций редактирования, необходимых для перехода от $s$ к $t$.

## 3.5.6. Расстояние Хэмминга

В пространстве векторов *расстояние Хэмминга* между двумя векторами можно определить, как количество позиций, в которых они различаются. Нетрудно

убедиться, что расстояние Хэмминга является метрикой. Очевидно, что оно не может быть отрицательным, а если равно нулю, то оба вектора совпадают. Расстояние Хэмминга не зависит от того, какой вектор считать первым. Неравенство треугольника также выполняется. Если $x$ и $z$ различаются в $m$ позициях, а $z$ и $y$ – в $n$ позициях, то $x$ и $y$ не могут различаться более чем в $m+n$ позициях. Как правило, расстояние Хэмминга используется для булевых векторов, все элементы которых равны 0 или 1. Но в принципе элементы можно выбирать из любого множества.

### Неевклидовы пространства

Отметим, что некоторые рассмотренные метрики относятся к неевклидовым пространствам. Важным для кластеризации (см. главу 7) свойством евклидова пространства является тот факт, что в нем всегда существует среднее значение любого множества точек, и это значение является точкой в том же пространстве. Рассмотрим, однако, пространство множеств, в котором определено расстояние Жаккара. Понятие «среднего» двух множеств не имеет смысла. Точно так же нельзя определить «среднее» двух строк.

Векторные пространства, для которых мы определили косинусное расстояние, могут быть или не быть евклидовыми. Если элементами векторов являются произвольные вещественные числа, то пространство евклидово. Если же мы ограничиваемся только целыми числами, то оно неевклидово. Так, например, не существует среднего двух векторов [1, 2] и [3, 1] в пространстве векторов с двумя целочисленными элементами, хотя если рассматривать их как точки в двумерном евклидовом пространстве, то мы могли бы сказать, что их среднее равно [2.0, 1.5].

**Пример 3.16.** Расстояние Хэмминга между векторами 10101 и 11110 равно 3, потому что они различаются во второй, четвертой и пятой позиции, но совпадают в первой и третьей.

## 3.5.7. Упражнения к разделу 3.5

**! Упражнение 3.5.1.** Какие из перечисленных ниже функций являются метриками в пространстве неотрицательных целых чисел? Докажите свой ответ либо приведите пример нарушения одной из аксиом.

(*а*) $\max(x, y)$ = наибольшее из чисел $x$ и $y$.
(*б*) $\text{diff}(x, y) = |x - y|$ (абсолютная величина разности между $x$ и $y$).
(*в*) $\text{sum}(x, y) = x + y$.

**Упражнение 3.5.2.** Найдите $L_1$- и $L_2$-расстояния между точками (5, 6, 7) и (8, 2, 4).

**!! Упражнение 3.5.3.** Докажите, что если $i$ и $j$ – любые положительные целые числа и $i < j$, то $L_i$-расстояние между любыми двумя точками меньше $L_j$-расстояния между ними.

**Упражнение 3.5.4.** Найдите расстояния Жаккара между следующими парами множеств:

(*а*) {1, 2, 3, 4} и {2, 3, 4, 5}.
(*б*) {1, 2, 3} и {4, 5, 6}.

**Упражнение 3.5.5.** Вычислите косинусы углов между следующими парами векторов[5]:

(*а*) (3, −1, 2) и (−2, 3, 1).
(*б*) (1, 2, 3) и (2, 4, 6).
(*в*) (5, 0, −4) и (−1, −6, 2).
(*г*) (0, 1, 1, 0, 1, 1) и (0, 0, 1, 0, 0, 0).

**! Упражнение 3.5.6.** Докажите, что косинусное расстояние между любыми двумя векторами одной и той же длины, состоящими из 0 и 1, не превосходит 90 градусов.

**Упражнение 3.5.7.** Найдите редакционные расстояния (с использованием только вставок и удалений) между следующими парами строк:

(*а*) `abcdef` и `bdaefc`.
(*б*) `abccdabc` и `acbdcab`.
(*в*) `abcdef` и `baedfc`.

**! Упражнение 3.5.8.** Существуют и другие определения редакционного расстояния. Например, помимо вставки и удаления, можно разрешить следующие операции:

*i.* *Замена*, когда один символ заменяется другим. Отметим, что замену всегда можно реализовать путем вставки с последующим удалением, но если замены разрешены, то при вычислении расстояния каждая такая операция считается как одна, а не две.

*ii.* *Транспозиция*, когда два соседних символа переставляются. Как и замену, транспозицию можно реализовать с помощью одной вставки и одного удаления, но считаться эта операция будет за один, а не за два шага.

Повторите упражнение 3.5.7 в случае, когда редакционное расстояние определено как количество вставок, удалений, замен и транспозиций, необходимых для преобразования одной строки в другую.

**! Упражнение 3.5.9.** Докажите, что редакционное расстояние, определенное в упражнении 3.5.8, действительно является метрикой.

**Упражнение 3.5.10.** Найдите расстояния Хэмминга между каждой парой следующих векторов: 000000, 110011, 010101, 011100.

---

[5] Отметим, что мы просим найти не совсем косинусное расстояние, но, зная косинус угла, легко найти сам угол, например, с помощью таблицы или библиотечной функции.

# 3.6. Теория функций, учитывающих близость

Метод LSH, разработанный в разделе 3.4, – пример одного семейства функций (минхэш-функций), которые можно комбинировать (с помощью разбиения на полосы), чтобы различить похожие и непохожие пары. Крутизна *S*-кривой на рис. 3.7 показывает, насколько эффективно мы можем избежать ложноположительных и ложноотрицательных результатов.

Теперь мы рассмотрим другие семейства функций, также позволяющие эффективно находить пары-кандидаты. Эти функции можно применять как к пространству множеств с расстоянием Жаккара, так и к другим пространствам с другими метриками. Семейство функций должно отвечать трем условиям.

1. Вероятность включить в число кандидатов близкую пару выше, чем далекую. Точную формулировку этого условия мы дадим в разделе 3.6.1.
2. Функции должны быть статистически независимы, т. е. вероятность того, что две или более функций совместно дадут некоторый результат, можно оценить как произведение вероятностей отдельных событий.
3. Они должны быть эффективны в двух отношениях:
   (*а*) Находить пары-кандидаты за время, гораздо меньшее того, что требуется для просмотра всех пар. Например, минхэш-функции обладают таким свойством, потому что вычислить минхэш-значения множеств можно за время, пропорциональное размеру набора данных, а не квадрату количества множеств в наборе. Поскольку множества с одинаковыми минхэшами попадают в одну ячейку, то мы неявно порождаем пары-кандидаты для одной минхэш-функции за время, гораздо меньшее требуемого для просмотра всех пар множеств.
   (*б*) Должны допускать комбинирование для построения функций, которые более эффективно избегают ложноположительных и ложноотрицательных результатов. При этом комбинированные функции также должны потреблять гораздо меньше времени, чем требуется для просмотра всех пар. Например, в методе разбиения на полосы из раздела 3.4.1 мы берем несколько минхэш-функций, каждая из которых по отдельности удовлетворяет условию 3(*а*), но не демонстрирует желаемого поведения *S*-кривой, и порождаем из них комбинированную функцию с *S*-кривой нужной формы.

Прежде всего, определим общее понятие «функции, учитывающей близость». Затем мы увидим, как эту идею можно применить к нескольким приложениям. И наконец, обсудим применение теории к произвольным данным с косинусной или евклидовой метрикой.

## 3.6.1. Функции, учитывающие близость

В этом разделе мы будем рассматривать функции, которые принимают два объекта и решают, следует ли считать их парой-кандидатом. Во многих случаях функция $f$ «хэширует» объекты, а решение основывается на том, получились ли результаты одинаковыми. Для краткости будем считать, что нотация $f(x) = f(y)$ означает, что $f(x, y)$ выработала решение «да, сделать $x$ и $y$ парой-кандидатом». А нотация $f(x) \neq f(y)$ будет означать «не делать $x$ и $y$ парой-кандидатом, если только какая-то другая функция не решит, что это нужно сделать».

Набор функций такого вида будем называть *семейством* функций. Например, семейством является набор минхэш-функций, основанных на перестановках строк характеристической матрицы.

Пусть $d_1 < d_2$ – два расстояния относительно некоторой метрики $d$. Говорят, что семейство **F** функций $(d_1, d_2, p_1, p_2)$-*чувствительно*, если для любой функции $f$, принадлежащей **F**:

1. Если $d(x, y) \leq d_1$, то вероятность того, что $f(x) = f(y)$, не меньше $p_1$.
2. Если $d(x, y) \geq d_2$, то вероятность того, что $f(x) = f(y)$, не больше $p_2$.

На рис. 3.9 показано, как должна вести себя вероятность того, что данная функция из $(d_1, d_2, p_1, p_2)$-чувствительного семейства объявит два объекта парой-кандидатом. Отметим, что мы ничего не говорим о том, что должно происходить, когда расстояние между объектами находится строго между $d_1$ и $d_2$, но можем делать $d_1$ и $d_2$ сколь угодно близкими. Правда, за это обычно приходится расплачиваться близостью также $p_1$ и $p_2$. Однако мы увидим, что можно раздвинуть границы $p_1$ и $p_2$, оставляя $d_1$ и $d_2$ фиксированными.

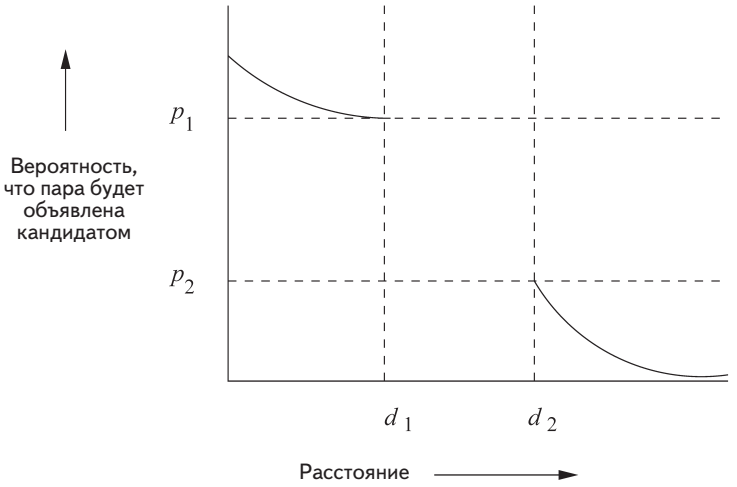



**Рис. 3.9.** Поведение $(d_1, d_2, p_1, p_2)$-чувствительной функции

## 3.6.2. LSH-семейства для расстояния Жаккара

На данный момент нам известно только одно семейство учитывающих близость функций (LSH-семейство): семейство минхэш-функций в предположении расстояния Жаккара. Как и раньше, считаем, что минхэш-функция $h$ объявляет $x$ и $y$ парой-кандидатом тогда и только тогда, когда $h(x) = h(y)$.

- Семейство минхэш-функций является $(d_1, d_2, 1 - d_1, 1 - d_2)$-чувствительным для любых $d_1$ и $d_2$ таких, что $0 \le d_1 < d_2 \le 1$.

Причина заключается в том, что если $d(x, y) \le d_1$, где $d$ – расстояние Жаккара, то $\text{SIM}(x, y) = 1 - d(x, y) \ge 1 - d_1$. Но, как мы знаем, коэффициент Жаккара $x$ и $y$ равен вероятности того, что минхэш-функция хэширует $x$ и $y$ в одно и то же значение. Аналогичное рассуждение применимо к $d_2$ и вообще любому расстоянию.

> **Пример 3.17.** Положим $d_1 = 0.3$, $d_2 = 0.6$. Тогда можно утверждать, что семейство минхэш-функций (0.3, 0.6, 0.7, 0.4)-чувствительно. Это означает, что если расстояние Жаккара между $x$ и $y$ не больше 0.3 (т. е. $\text{SIM}(x, y) \ge 0.7$), то с вероятностью не менее 0.7 минхэш-функция хэширует $x$ и $y$ в одно и то же значение, а если расстояние Жаккара не больше 0.6 (т. е. $\text{SIM}(x, y) \le 0.4$), то вероятность хэширования $x$ и $y$ в одно и то же значение не превышает 0.4. Отметим, что такое же рассуждение можно было бы провести при любом другом выборе $d_1$ и $d_2$, лишь бы выполнялось условие $d_1 < d_2$.

## 3.6.3. Расширение LSH-семейства

Пусть дано $(d_1, d_2, p_1, p_2)$-чувствительное семейство **F**. Можно построить новое семейство **F′**, применив к F следующую процедуру *AND-построения*. Каждый член **F′** состоит из $r$ членов **F** при некотором фиксированном $r$. Если $f$ принадлежит **F′** и $f$ построена из множества $\{f_1, f_2, \ldots, f_r\}$ членов **F**, то мы говорим, что $f(x) = f(y)$ тогда и только тогда, когда $f_i(x) = f_i(y)$ для всех $i = 1, 2, \ldots, r$. Отметим, что такое построение моделирует действие полосы из $r$ строк: полоса объявляет $x$ и $y$ парой-кандидатом, если каждая ее строка считает, что $x$ и $y$ равны (и потому образуют пару-кандидат с точки зрения одной лишь этой строки).

Поскольку при построении **F′** члены **F** выбираются независимо, мы можем утверждать, что **F′** – $(d_1, d_2, (p_1)^r, (p_2)^r)$-чувствительное семейство. То есть, если $p$ – вероятность того, что **F** объявит пару $(x, y)$ кандидатом, то **F′** сделает это с вероятностью $p^r$.

Существует также процедура *OR-построения*, которая преобразует $(d_1, d_2, p_1, p_2)$-чувствительное семейство **F** в $(d_1, d_2, 1 - (1 - p_1)^b, 1 - (1 - p_2)^b)$-чувствительное семейство **F′**. Каждый член $f$ семейства **F′** строится из $b$ членов **F** $f_1, f_2, \ldots, f_b$. По определению, $f(x) = f(y)$ только и только тогда, когда $f_i(x) = f_i(y)$ хотя бы для одного значения $i$. OR-построение моделирует комбинирование нескольких полос: $x$ и $y$ становятся парой-кандидатом, если хотя бы одна полоса объявляет их таковой.

Если $p$ – вероятность того, что член **F** объявит $(x, y)$ парой-кандидатом, что $1 - p$ – вероятность того, что не объявит, $(1 - p)^b$ – вероятность того, что ни одна из функций $f_1, f_2, \ldots, f_b$ не объявит $(x, y)$ парой-кандидатом, а $1 - (1 - p)^b$ – вероят-

ность того, что хотя бы одна функция $f_i$ объявит $(x, y)$ парой-кандидатом, т. е. что кандидатом эту пару объявит $f$.

Отметим, что AND-построение уменьшает все вероятности, но если выбрать **F** и $r$ осмотрительно, то мы сможем сделать маленькую вероятность $p_2$ очень близкой к 0, при том что более высокая вероятность $p_1$ будет далеко отстоять от 0. Аналогично OR-построение увеличивает все вероятности, но благодаря разумному выбору **F** и $b$ мы сможем приблизить большую вероятность к 1, но так, что меньшая будет отстоять от 1 достаточно далеко. AND- и OR-построения можно сочетать в любом порядке, чтобы максимально приблизить более низкую вероятность к 0, а более высокую – к 1. Конечно, чем больше построений входит в каскад и чем выше значения $r$ и $b$, тем большее число функций из исходного семейства придется использовать. Поэтому чем лучше конечное семейство функций, тем больше времени будет уходить на применение функций из него.

**Пример 3.18.** Пусть мы начинаем с семейства **F**. Воспользуемся AND-построением с $r = 4$ для получения семейства $\mathbf{F}_1$. Затем применим к $\mathbf{F}_1$ OR-построение с $b = 4$ для получения третьего семейства $\mathbf{F}_2$. Отметим, что члены $\mathbf{F}_2$ строятся из 16 членов **F**, и ситуация выглядит так же, как если бы мы начали с 16 минхэш-функций и рассматривали их как четыре полосы по четыре функции в каждой.

4-путевое AND-построение преобразует любую вероятность $p$ в $p^4$. Если вслед за ней выполнить 4-путевое OR-построение, то результат будет преобразован в $1 - (1 - p^4)^4$. На рис. 3.10 показано, как при этом преобразуются несколько значений $p$. График этой функции представляет собой $S$-кривую, которая долго остается малой, затем резко возрастает (впрочем, не так уж резко – угловой коэффициент нигде не становится много больше 2), после чего остается на высоком плато. Как у любой $S$-кривой, у нее есть неподвижная точка, в которой $f(p) = p$. В данном случае, неподвижной точкой будет такое значение $p$, при котором $p = 1 - (1 - p^4)^4$. Как видим, оно находится где-то между 0.7 и 0.8. До этого значения применение функции уменьшает вероятность, а после нее – увеличивает. Поэтому если взять высокую вероятность после неподвижной точки, а низкую – до нее, то мы получим желаемый результат: уменьшение низкой вероятности и увеличение высокой.

| $p$ | $1 - (1 - p^4)^4$ |
|---|---|
| 0.2 | 0.0064 |
| 0.3 | 0.0320 |
| 0.4 | 0.0985 |
| 0.5 | 0.2275 |
| 0.6 | 0.4260 |
| 0.7 | 0.6666 |
| 0.8 | 0.8785 |
| 0.9 | 0.9860 |

**Рис. 3.10.** Результат композиции 4-путевого AND-построения и 4-путевого OR-построения

Пусть **F** – (0.2, 0.6, 0.8, 0.4)-чувствительное семейство минхэш-функций. Тогда $\mathbf{F}_2$, семейство, построенное применением 4-путевого AND-построения и последующего 4-путевого OR-построения, будет (0.2, 0.6, 0.8785, 0.0985)-чувствительным, как следует из строк таблицы на рис. 3.10 для $p$ равного 0.8 и 0.4. Заменив **F** на $\mathbf{F}_2$, мы одновременно уменьшили частоты ложноотрицательных и ложноположительных результатов, заплатив за это 16-кратным увеличением времени вычисления функций.

**Пример 3.19.** Заплатив ту же цену, мы можем применить 4-путевое OR-построение с последующим 4-путевым AND-построением. На рис. 3.11 показано, как при этом преобразуются вероятности. Предположим, к примеру, что **F** – (0.2, 0.6, 0.8, 0.4)-чувствительное семейство. Тогда построенное семейство будет (0.2, 0.6, 0.9936, 0.5740)-чувствительным. Пожалуй, этот выбор не оптимален. Да, высокая вероятность стала гораздо ближе к 1, но и низкая возросла, в результате чего увеличится количество ложноположительных результатов.

| $p$ | $1 - (1 - p)^4)^4$ |
|---|---|
| 0.1 | 0.0140 |
| 0.2 | 0.1215 |
| 0.3 | 0.3334 |
| 0.4 | 0.5740 |
| 0.5 | 0.7725 |
| 0.6 | 0.9015 |
| 0.7 | 0.9680 |
| 0.8 | 0.9936 |

**Рис. 3.11.** Результат композиции 4-путевого OR-построения и 4-путевого AND-построения

**Пример 3.20.** Мы можем включить в каскад сколько угодно построений. Например, можно было бы применить построение из примера 3.18 к семейству минхэш-функций, а затем к результату применить построение из примера 3.19. Каждая функция в построенном таким образом семействе будет являться композицией 256 исходных минхэш-функций. А (0.2, 0.8, 0.8, 0.2)-чувствительное семейство при этом преобразуется в (0.2, 0.8, 0.9991285, 0.0000004)-чувствительное.

## 3.6.4. Упражнения к разделу 3.6

**Упражнение 3.6.1.** Как изменятся вероятности, если начать с семейства минхэш-функций и применить:

(*а*) 2-путевое AND-построение, а затем 3-путевое OR-построение.

(*б*) 2-путевое OR-построение, а затем 3-путевое AND-построение.

(*в*) 2-путевое AND-построение, а затем 2-путевое OR-построение и 2-путевое AND-построение.

(*г*) 2-путевое OR-построение, а затем 2-путевое AND-построение, 2-путевое OR-построение и снова 2-путевое AND-построение.

**Упражнение 3.6.2.** Найдите неподвижные точки всех функций, построенных в упражнении 3.6.1.

**! Упражнение 3.6.3.** Для любой функции от вероятности $p$, например показанной на рис. 3.10, угловой коэффициент вычисляется как производная функции. Максимальным угловой коэффициент будет в точке, где производная достигает максимума. Найдите значение $p$, в котором достигается максимум углового коэффициента, для $S$-кривых, соответствующих рисункам 3.10 и 3.11. Каковы эти значения углового коэффициента?

**!! Упражнение 3.6.4**. Обобщите результат упражнения 3.6.3 и выразите точку, в которой достигается максимум углового коэффициента, и его значение в виде функции от $r$ и $b$ для семейств функций, построенных из минхэш-функций путем применения следующих преобразований:

(а) $r$-путевого AND-построения с последующим $b$-путевым OR-построением.
(б) $b$-путевого OR-построения с последующим $r$-путевым AND-построением.

# 3.7. LSH-семейства для других метрик

Нет никакой гарантии, что для заданной метрики существует LSH-семейство хэш-функций. До сих пор мы видели такие семейства только для расстояния Жаккара. В этом разделе мы покажем, как построить семейства учитывающих близость функций для расстояния Хэмминга, косинусного расстояния и обычного евклидова расстояния.

## 3.7.1. LSH-семейства для расстояния Хэмминга

Построить LSH-семейство функций для расстояния Хэмминга совсем просто. Пусть имеется пространство $d$-мерных векторов, обозначим $h(x, y)$ расстояние Хэмминга между векторами $x$ и $y$. Взяв только одну позицию, скажем с номером $i$, мы сможем определить функцию $f_i(x)$ как $i$-ый элемент вектора $x$. Тогда $f_i(x) = f_i(y)$ тогда и только тогда, когда векторы $x$ и $y$ совпадают в $i$-ой позиции. Вероятность, что $f_i(x) = f_i(y)$ для случайно выбранного $i$, равна $1 - h(x, y)/d$, т. е. доле позиций, в которых $x$ и $y$ совпадают.

Эта ситуация почти идентична той, что имела место для минхэш-функций. Таким образом, семейство **F**, состоящее из функций $\{f_1, f_2, ..., f_d\}$, является $(d_1, d_2, 1 - d_1/d, 1 - d_2/d)$-чувствительным для любого $d_1 < d_2$. Между ним и семейством минхэш-функций есть всего два различия.

1. Расстояние Жаккара изменяется от 0 до 1, а расстояние Хэмминга в пространстве $d$-мерных векторов – от 0 до $d$. Поэтому расстояния необходимо масштабировать, поделив на $d$, чтобы получить из них вероятности.
2. Запас минхэш-функций практически неограничен, а размер семейства **F** для расстояния Хэмминга равен в точности $d$.

Первое отличие несущественно, необходимо лишь в нужной точке вычисления выполнить деление на $d$. А вот второе серьезнее. Если $d$ относительно мало, то количество функций, которые можно составить, применяя AND- и OR-построения, ограничено, а потому ограничена и максимально достижимая крутизна $S$-кривой.

## 3.7.2. Случайные гиперплоскости и косинусное расстояние

Напомним (см. раздел 3.5.4), что косинусное расстояние между векторами определяется как угол между ними. На рис. 3.12 изображены два вектора $x$ и $y$, образующие угол θ. Отметим, что векторы могут находиться в пространстве любого числа измерений, но всегда определяют плоскость, и угол измеряется в этой плоскости. На рис. 3.12 показан «взгляд сверху» на плоскость, содержащую $x$ и $y$.

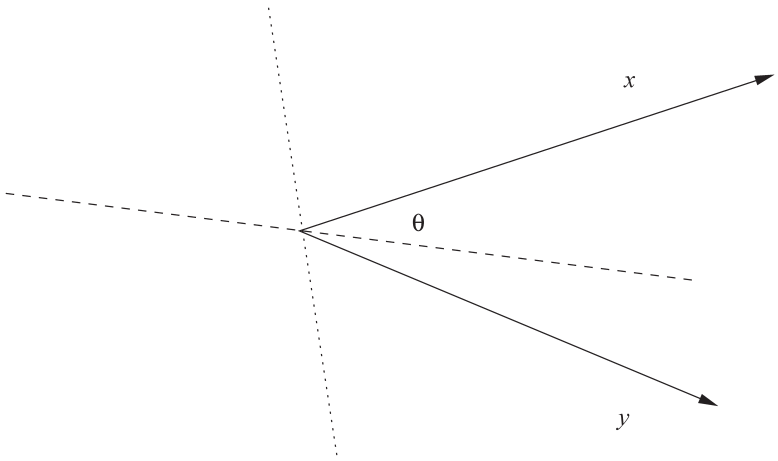



**Рис. 3.12.** Два вектора, образующие угол θ

Выберем какую-нибудь гиперплоскость, проходящую через начало координат. Она пересекает плоскость, определяемую векторами $x$ и $y$, по прямой. На рис. 3.12 показаны линии пересечения с двумя гиперплоскостями: одна штриховая, другая пунктирная. Чтобы выбрать случайную гиперплоскость, мы на самом деле выбираем нормальный к ней вектор, скажем $v$. Тогда гиперплоскость будет множеством точек, для которых скалярное произведение с $v$ равно 0.

Сначала рассмотрим вектор $v$, нормальный к гиперплоскости, проекция которой представлена штриховой линией на рис. 3.12; в этом случае $x$ и $y$ находятся по разные стороны гиперплоскости. Если предположить, что проекция вектора $v$ на плоскость $(x, y)$ расположена выше штриховой линии, то скалярное произведение $v \cdot x$ положительно, а $v \cdot y$ отрицательно. Но нормальный вектор $v$ может быть направлен и противоположно, т. е. находиться ниже штриховой линии. Тогда $v \cdot x$ отрицательно, а $v \cdot y$ положительно, но знаки все равно различаются.

С другой стороны, случайно выбранный вектор $v$ мог бы оказаться нормальным к гиперплоскости, представленной пунктирной линией на рис. 3.12. Тогда знаки $v \cdot x$ и $v \cdot y$ совпадают. Если проекция $v$ оказывается правее, то оба скалярных произведения положительны, а если левее – то отрицательны.

Какова вероятность, что случайно выбранный вектор будет нормален к гиперплоскости такой, как представленная штриховой, а не пунктирной линией? Все углы для прямой, по которой случайная гиперплоскость пересекает плоскость $(x, y)$, равновероятны. Следовательно, гиперплоскость будет выглядеть как штриховая линия с вероятностью $\theta/180$ и как пунктирная – с вероятностью $1 - \theta/180$.

Таким образом, каждая хэш-функция $f$ в нашем LSH-семействе **F** строится по случайно выбранному вектору $v_f$. Если даны два вектора $x$ и $y$, то мы говорим, что $f(x) = f(y)$ тогда и только тогда, когда скалярные произведения $v_f \cdot x$ и $v_f \cdot y$ имеют одинаковый знак. Мы определили LSH-семейство для косинусного расстояния. Параметры по существу такие же, как для семейства на основе расстояния Жаккара из раздела 3.6.2, только величина расстояния изменяется от 0 до 180, а не от 0 до 1. То есть **F** – $(d_1, d_2, (180 - d_1)/180, (180 - d_2)/180)$-чувствительное семейство функций. Его можно расширить точно так же, как семейство минхэш-функций.

## 3.7.3 Эскизы

Мы можем выбирать случайный вектор не из всего множества векторов, а только из тех, что состоят из элементов +1 и –1. Скалярное произведение произвольного вектора $x$ на вектор $v$ из +1 и –1 вычисляется сложением элементов $x$, соответствующих положительным элементам $v$, и вычитанием остальных элементов $x$ – тех, которым в $v$ соответствует –1.

Взяв набор случайных векторов, например $v_1, v_2, ..., v_n$, мы сможем применить его к произвольному вектору $x$ следующим образом: вычислить $v_1 \cdot x, v_2 \cdot x, ..., v_n \cdot x$, а затем заменить положительные значения на +1, а отрицательные – на –1. Результат называется *эскизом* (sketch) вектора $x$. Нулевые произведения можно обрабатывать произвольно, например, случайным образом заменяя на +1 или –1. Поскольку вероятность получения нулевого скалярного произведения пренебрежимо мала, любое принятое решение не отразится на результате.

**Пример 3.21.** Пусть пространство состоит из 4-мерных векторов, и мы случайно выбрали три вектора: $v_1 = [+1,–1,+1,+1]$, $v_2 = [–1,+1,–1,+1]$ и $v_3 = [+1,+1,–1,–1]$. Эскиз вектора $x = [3, 4, 5, 6]$ равен $[+1,+1,–1]$. Действительно, $v_1 \cdot x = 3–4+5+6 = 10$. Поскольку результат положителен, первый элемент эскиза равен +1. Аналогично $v_2 \cdot x = 3$ и $v_3 \cdot x = –4$, поэтому второй элемент равен +1, а третий –1.

Рассмотрим вектор $y = [4, 3, 2, 1]$. Его эскиз равен $[+1,–1,+1]$. Поскольку эскизы $x$ и $y$ совпадают в 1/3 позиций, угол между ними можно оценить как 120 градусов. Таким образом, случайно выбранная гиперплоскость будет выглядеть как штриховая линия на рис. 3.12 с вероятностью в два раза больше, чем как пунктирная.

Этот вывод очень далек от действительности. Мы можем вычислить косинус угла между $x$ и $y$ как скалярное произведение:

$$x \cdot y = 6 \times 1 + 5 \times 2 + 4 \times 3 + 3 \times 4 = 40$$

поделенное на произведение модулей обоих векторов. Модули равны соответственно

$$\sqrt{6^2 + 5^2 + 4^2 + 3^2} = 9.274 \quad \text{и} \quad \sqrt{1^2 + 2^2 + 3^2 + 4^2} = 5.477 .$$

Следовательно, косинус угла между $x$ и $y$ равен 0.7875, а сам угол составляет примерно 38 градусов. Однако, рассмотрев все 16 возможных векторов длины 4 с элементами +1 и –1, мы обнаружим, что среди них есть лишь четыре, для которых знаки скалярных произведений с $x$ и $y$ различаются, а именно $v_2$, $v_3$ и их дополнения [+1,–1,+1,–1] и [–1,–1,+1,+1]. Значит, если бы мы выбрали для формирования эскиза все 16 векторов, то оценка угла составила бы 180/4 = 45 градусов.

## 3.7.4. LSH-семейства для евклидова расстояния

Теперь обратимся к евклидову расстоянию (см. раздел 3.5.2) и разработаем для него LSH-семейство хэш-функций. Начнем с двумерного евклидова пространства. Каждая хэш-функция $f$ в нашем семействе **F** будет ассоциирована со случайно выбранной прямой в этом пространстве. Выберем константу $a$ и разобьем прямую на отрезки длины $a$, как показано на рис. 3.13, где «случайная» прямая горизонтальна.

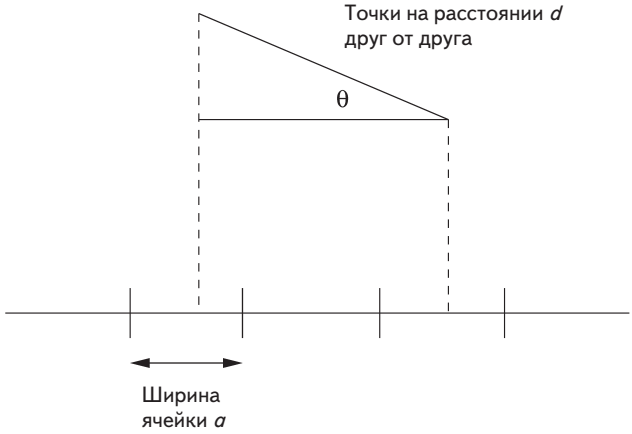



**Рис. 3.13.** У двух точек, отстоящих друг от друга на расстояние $d \gg a$, мало шансов быть хэшированными в одну и ту же ячейку

Отрезки прямой – это ячейки, в которые функция $f$ хэширует точки. Точка хэшируется в ту ячейку, в которую попадает ее проекция на прямую. Если расстояние $d$ между двумя точками мало по сравнению с $a$, то с большой вероятностью обе точки попадут в одну и ту же ячейку, и тогда хэш-функция $f$ объявит их равными. Например, если $d = a/2$, то с вероятностью не менее 50 % обе точки попадут

в одну ячейку. На самом деле, если угол θ между случайно выбранной прямой и прямой, соединяющей две точки, велик, то вероятность попадания в одну ячейку даже выше. Например, если θ равен 90 градусам, то обе точки наверняка окажутся в одной ячейке.

Предположим, однако, что $d$ больше $a$. Чтобы у двух точек был хоть какой-то шанс попасть в одну ячейку, должно быть выполнено условие $d \cos\theta \leq a$. Из рис. 3.12 видно, почему это так. Отметим, что даже если $d \cos\theta \ll a$, все равно нет гарантии, что обе точки окажутся в одной ячейке. Но можно гарантировать следующее. Если $d \geq 2a$, то вероятность попадания в одну ячейку не превышает 1/3. Поясним. Чтобы cos θ был меньше 1/2, угол θ должен находиться в диапазоне от 60 до 90 градусов. Если θ находится в диапазоне от 0 до 60 градусов, то cos θ больше 1/2. Но так как θ – меньший из двух углов, образованных двумя случайно выбранными прямыми на плоскости, то вероятность, что θ попадет в диапазон от 0 до 60, в два раза больше вероятности попадания в диапазон от 60 до 90.

Мы заключаем, что только что описанное семейство хэш-функций **F** является $(a/2, 2a, 1/2, 1/3)$-чувствительным. То есть, если расстояние не превышает $a/2$, то две точки, разделенные таким расстоянием, с вероятностью не ниже 1/2 попадут в одну ячейку, а если расстояние не меньше $2a$, то вероятность попадания в одну ячейку не больше 1/3. Это семейство можно расширить, как и любое другое из рассмотренных выше LSH-семейств хэш-функций.

## 3.7.5. Другие примеры LSH-семейств в евклидовых пространствах

Семейство хэш-функций из раздела 3.7.4 не вполне удовлетворительно. Во-первых, мы описали методику его построения только для двумерных пространств. А что, если данные представлены точками в многомерном пространстве? Во-вторых, для расстояния Жаккара и косинусного расстояния мы смогли разработать LSH-семейства для любой пары расстояний $d_1$ и $d_2$, удовлетворяющих условию $d_1 < d_2$. А в разделе 3.7.4 наложено более сильное условие $d_1 < 4d_2$.

Однако мы утверждаем, что существует LSH-семейство хэш-функций для любых $d_1 < d_2$ и любого числа измерений. Хэш-функции по-прежнему порождаются случайными прямыми в пространстве и длиной отрезка $a$, являющегося одной ячейкой. Мы, как и раньше, хэшируем точки, проецируя их на прямую. Имея лишь условие $d_1 < d_2$, мы не можем вычислить вероятность $p_1$ хэширования двух точек, разделенных расстоянием $d_1$, в одну ячейку, но можем быть уверены, что она больше $p_2$ – вероятности попадания в одну ячейку двух точек, отстоящих друг от друга на расстояние $d_2$. И это понятно – ведь вероятность растет с убыванием расстояния. Таким образом, даже не имея простого способа вычисления $p_1$ и $p_2$, мы знаем, что существует $(d_1, d_2, p_1, p_2)$-чувствительное семейство хэш-функций для любых $d_1 < d_2$ и любого числа измерений.

Применяя технику расширения (см. раздел 3.6.3), мы можем развести вероятности настолько далеко, насколько пожелаем. Разумеется, чем дальше мы хотим их развести, тем больше базовых хэш-функций из **F** придется использовать.

## 3.7.6. Упражнения к разделу 3.7

**Упражнение 3.7.1.** Допустим, мы строим базовое семейство из шести учитывающих близость функций для векторов длины 6. Для каждой пары векторов 000000, 110011, 010101, 011100 ответьте, какая из шести функций делает ее парой-кандидатом?

**Упражнение 3.7.2.** Будем вычислять эскизы с помощью следующих «случайно» выбранных векторов:

$$v_1 = [+1,+1,+1,-1] \quad v_2 = [+1,+1,-1,+1]$$
$$v_3 = [+1,-1,+1,+1] \quad v_4 = [-1,+1,+1,+1]$$

Вычислите эскизы следующих векторов.

(*а*) [2, 3, 4, 5]

(*б*) [–2, 3,–4, 5]

(*в*) [2,–3, 4,–5]

Для каждой пары векторов оцените угол между ними на основе эскизов. А каково истинное значение угла?

**Упражнение 3.7.3.** Допустим, что эскизы строятся на основе всех шестнадцати векторов длины 4 с элементами +1 или –1. Вычислите эскизы трех векторов из упражнения 3.7.2. Как соотносятся оценки углов между каждой парой векторов с истинными значениями?

**Упражнение 3.7.4.** Допустим, что эскизы строятся на основе четырех векторов из упражнения 3.7.2.

**!**(а) Какие ограничения нужно наложить на $a$, $b$, $c$, $d$, чтобы эскиз вектора $[a, b, c, d]$ был равен [+1,+1,+1,+1]?

**!!**(б) Рассмотрим два вектора $[a, b, c, d]$ и $[e, f, g, h]$. Какие условия нужно наложить на $a$, $b$, ..., $h$, чтобы эскизы обоих векторов совпадали?

**Упражнение 3.7.5.** Возьмем три точки в трехмерном евклидовом пространстве: $p_1 = (1, 2, 3)$, $p_2 = (0, 2, 4)$, $p_3 = (4, 3, 2)$. Рассмотрим три хэш-функции, определяемые тремя осями координат (чтобы упростить вычисления). Пусть длина ячейки равна $a$, одна ячейка совпадает с полуинтервалом $[0, a)$ (множество таких точек $x$, для которых $0 \le x < a$), следующая – с полуинтервалом $[a, 2a)$, предыдущая – с полуинтервалом $[-a, 0)$ и так далее.

(*а*) Для каждой из трех прямых определите, в какую ячейку попадает каждая точка, в предположении, что $a = 1$.

(*б*) Тот же вопрос для $a = 2$.

(*в*) Каковы пары-кандидаты в случаях $a = 1$ и $a = 2$?

**!**(*г*) Для каждой пары точек ответьте, при каких значениях $a$ эта пара будет кандидатом?

# 3.8. Применения хэширования с учетом близости

В этом разделе мы рассмотрим три практических применения LSH. В каждом случае изученные методы придется модифицировать с учетом ограничений задачи.

1. *Отождествление объектов* (entity resolution). Этим термином обозначается отождествление записей данных, относящихся к одному и тому же объекту реального мира, например человеку. Основная проблема здесь заключается в том, что сходство записей нельзя точно описать в терминах модели похожих множеств или похожих векторов, на базе которой построена теория.
2. *Сравнение отпечатков пальцев*. Отпечатки пальцев можно представить в виде множеств. Но мы не будем прибегать к минхэш-функциям, а исследуем другое семейство функций, учитывающих близость.
3. *Сравнение газетных статей*. Здесь мы рассмотрим другое понятие шинглов, в котором внимание акцентируется на основной статье на сайте газеты, а дополнительные материалы – рекламные объявления, реквизиты издания и прочее – игнорируются.

## 3.8.1. Отождествление объектов

Часто бывает, что имеется несколько наборов данных и известно, что все они относятся к одним и тем же объектам. Например, разные библиографические источники содержат информацию об одних и тех же книгах или статьях. В общем случае имеются записи, описывающие объекты некоторого типа, например людей или книги. Формат записей может быть одинаковым или различным, состав информации также может различаться.

Есть много причин, почему информация об объекте может различаться, даже если контекст один и тот же. Например, в разных записях имена могут быть записаны по-разному из-за ошибок, отсутствия среднего инициала, использования краткого имени и т. д. Скажем, имена «Bob S. Jomes» и «Robert Jones Jr.» могут относиться к одному человеку или к разным людям. Если записи поступили из разных источников, может также отличаться состав полей. В одном источнике может присутствовать поле «age» (возраст), а в другом – нет. Зато во втором источнике может быть поле «date of birth» (дата рождения), а может и вообще не быть сведений о возрасте.

## 3.8.2. Пример отождествления объектов

Мы рассмотрим реальный пример использования LSH для отождествления объектов. Компания *A* нанята компанией *B* для привлечения клиентов. Компания *B* обязуется выплачивать *A* вознаграждение по итогам года, если клиент не перестал быть ее абонентом. Впоследствии компании рассорились и не смогли прийти к соглашению о том, сколько клиентов *A* привела в *B*. У каждой имеется примерно 1 000 000 записей, и некоторые из них описывают одинаковых людей: тех,

которых привлекла *A*. Состав полей в записях различен, но, к сожалению, нигде нет поля «это клиент, которого *A* привлекла в *B*». Следовательно, стоит задача сопоставить записи из двух наборов и понять, представляет ли пара одного и того же человека.

В каждой записи есть поля «имя», «адрес» и «телефон». Однако значения этих полей по разным причинам могут не совпадать. И дело не только в орфографических ошибках и других различиях в написании имен, упомянутых в разделе 3.8.1. Есть и другие возможности для расхождения. Например, клиент мог сообщить компании *A* свой домашний телефон, а компании *B* – сотовый. Или клиент мог переехать и сообщить об этом *B*, но не *A* (потому что с *A* больше не поддерживает деловых отношений). Иногда также меняются коды зон, а, значит, и номера телефонов.

Стратегия отождествления записей подразумевала оценивание различий в трех полях: имя, адрес и телефон. Для вычисления оценки вероятности того, что две записи, одна из *A*, другая из *B*, описывают одно и то же лицо, каждому полю был назначен вес 100 баллов, поэтому для пары записей, в которых все три поля в точности совпадают, оценка равна 300. Однако расхождение в каждом поле штрафуется. В качестве первого приближения использовалось редакционное расстояние (раздел 3.5.5), но штраф рос квадратично с увеличением расстояния. Тогда стали использовать имеющиеся в свободном доступе таблицы, позволяющие в некоторых ситуациях уменьшить штраф. Например, считалось, что имена «Bill» и «William» различаются только в одной букве, хотя редакционное расстояние между ними равно 5.

**Когда можно отождествлять записи?**

Конечно, каждый случай имеет свои особенности, но интересно знать, какие результаты эксперимент, описанный в разделе 3.8.3, дал в применении к данным из раздела 3.8.2. Для оценок, не меньших 185, значение *x* было очень близко к 10, т. е. такая оценка показывала, что вероятность, что две записи представляют одно лицо, практически равна 1. Отметим, что в этом примере оценка 185 означает, что значения в одном поле точно совпадают (а иначе запись вообще не получила бы оценку), во втором поле совершенно различны, а в третьем различаются, но не сильно. Далее, для такой низкой оценки, как 115, значение *x* было заметно меньше 45, т. е. некоторые пары с такой оценкой все же представляли одно лицо. Отметим, что оценка 115 соответствует случаю, когда значения в одном поле одинаковы, а в двух других наблюдается слабое сходство.

Но вычислить оценки для каждой из триллиона пар записей практически невозможно. Поэтому была выбрана простая LSH-стратегия, позволявшая сосредоточиться только на вероятных кандидатах. Использовались три «хэш-функции». Первая сопоставляла записям одну и ту же ячейку, если совпадали имена; вторая

поступала так же для одинаковых адресов, третья – для одинаковых телефонов. На практике никакого хэширования не производилось; записи просто сортировались по имени, так что записи с одинаковыми именами оказывались рядом и получали оценку общего сходства имени, адреса и телефона. Затем записи сортировались по адресу и оценивались записи с одинаковым адресом. В третий раз записи сортировались по телефону и оценивались записи с одинаковым телефоном.

При таком подходе пропускалась пара записей, которые представляли одно лицо, но значения во всех трех полях немного различались. Однако целью было доказать в суде, что две записи относятся к одному человеку, а судья в любом случае вряд ли счел бы такие записи идентичными.

## 3.8.3. Проверка отождествления записей

Осталось понять, насколько высока должна быть оценка, чтобы можно было считать, что две записи действительно представляют одно и то же лицо. В рассматриваемом примере был простой способ принять такое решение, применимый и во многих похожих ситуациях. Было решено анализировать даты создания записей и предполагать, что между датой привлечения в компании A и датой регистрации в компании B никак не могло пройти больше 90 дней. Следовательно, если случайно выбрать две записи, на которые наложено только одно ограничение – интервал между датами в записи B и в записи A должен составлять от 0 до 90 дней, – то среднее запаздывание должна составить 45 дней.

Обнаружилось, что в парах, получивших наивысшую оценку 300, среднее запаздывание было равно 10 дням. Если предположить, что пары с оценкой 300 без сомнения описывают одно лицо, то можно взять множество пар с любой заданной оценкой $s$ и вычислить для них среднее запаздывание. Предположим, что среднее запаздывание равно $x$, а доля правильно отождествленных пар с оценкой $s$ равна $f$. Тогда $x = 10f + 45(1 - f)$, или $x = 45 - 35f$. Решив это уравнение относительно $f$, найдем, что доля правильно отождествленных пар с оценкой $s$ равна $(45 - x)/35$.

Такой прием проходит во всех случаях, когда выполняются два условия:

1. Существует система оценивания, позволяющая вычислить вероятность того, что две записи представляют один и тот же объект.
2. Существует поле, не используемое при оценивании, на основании которого можно вывести показатель, отличающий в среднем правильно и неправильно отождествленные пары.

Предположим, к примеру, что в записях обеих компаний A и B есть поле «рост». Мы можем вычислить среднюю разницу в росте для пар случайных записей, а затем для записей с наивысшей оценкой (которые заведомо представляют одно и то же лицо). Для заданной оценки $s$ можно вычислить среднюю разницу в росте для пар с такой оценкой и оценить вероятность того, что записи представляют одно лицо. То есть, если $h_0$ – средняя разница в росте для точных совпадений, $h_1$ – средняя разница в росте для случайных пар, а $h$ – средняя разница в росте для пар

с оценкой $s$, то доля правильно отождествленных пар с оценкой $s$ равна $(h_1 - h)/(h_1 - h_0)$.

## 3.8.4. Сравнение отпечатков пальцев

При сравнении отпечатков пальцев компьютером в качестве представления обычно выступает не изображение, а множество *минуций*. Минуцией называется точка, где меняется рисунок папиллярных линий, например, две линии сливаются или одна заканчивается. Если наложить на отпечаток сетку, то его можно будет представить множеством ячеек, в которых находятся минуции.

В идеале перед наложением сетки отпечатки пальцев нормализуются, т. е. приводятся к единому размеру и ориентации, так чтобы, взяв два изображения одного и того же пальца, мы нашли бы минуции в одних и тех же ячейках. Мы здесь не будем рассматривать наилучшие способы нормализации изображений. Предположим, что в результате комбинирования различных приемов, в том числе выбора размера сетки и размещения минуции в нескольких соседних ячейках, если она оказалась близко к границе ячейки, можно выдвинуть обоснованное предположение о том, что вероятность совпадения ячеек, полученных из двух изображений, с точки зрения наличия или отсутствия минуции, гораздо выше, если это изображения одного, а не разных пальцев.

Таким образом, отпечатки пальцев можно представлять множествами ячеек сетки – тех, в которых находятся минуции, – сравнивать, как любые другие множества, используя коэффициент или расстояние Жаккара. Однако у задачи сравнения отпечатков есть две разновидности.

- Задача *многие-с-одним* встречается чаще. На пистолете обнаружены отпечатки пальцев и мы хотим сравнить их со всеми отпечатками в большой базе данных.
- Задача *многие-с-многими* заключается в том, чтобы найти во всей базе данных пары отпечатков, принадлежащих одному и тому же человеку.

Вариант многие-с-многими соответствует модели, которой мы руководствовались при поиске похожих объектов, но ту же технику можно применить и для ускорения поиска в задаче многие-с-одним.

## 3.8.5. LSH-семейство для сравнения отпечатков пальцев

Мы могли бы вычислить минхэши множеств, представляющих отпечатки пальцев, и воспользоваться стандартной техникой LSH, описанной в разделе 3.4. Но поскольку множества выбираются из сравнительно небольшого числа ячеек сетки (порядка 1000), то непонятно, стоит ли сворачивать их в более короткие сигнатуры. В этом разделе мы изучим еще один вид хэширования с учетом близости, который хорошо работает для данных обсуждаемого вида.

Предположим, к примеру, что вероятность обнаружить минуцию в случайной ячейке случайного отпечатка равна 20 %. Предположим также, что если два от-

печатка принадлежат одному пальцу, и на одном из них в данной ячейке сетки есть минуция, то вероятность, что она есть там и на другом равна 80 %. Определим KSH-семейство хэш-функций следующим образом. Каждая функция $f$ в семействе **F** определяется тремя ячейками сетки. Функция $f$ возвращает для двух отпечатков «да», если на обоих имеются минуции во всех трех ячейках, в противном случае $f$ возвращает «нет». Иначе говоря, $f$ сопоставляет один и тот же хэш-код всем отпечаткам, для которых имеются минуции во всех трех точках сетки, определяющих $f$, а всем остальным отпечаткам сопоставляет уникальные хэш-коды. Далее мы будем называть хэш-кодом для $f$ только первый из указанных хэш-кодов, а остальные игнорировать.

При решении задачи многие-с-одним мы можем использовать много функций из семейства **F** и заранее вычислить хэш-коды отпечатков, для которых они возвращают «да». Тогда, получив новый отпечаток, который требуется сравнить, мы определяем, какие хэш-коды ему соответствует, и сравниваем его со всеми отпечатками, у которых такие же хэш-коды. Для решения задачи многие-со-многими мы вычисляем хэш-коды для каждой функции и сравниваем все отпечатки с одинаковыми хэш-кодами.

Посчитаем, сколько функций нужно, чтобы получить разумную вероятность обнаружения совпадения, не сравнивая отпечаток пальца на пистолете с каждым из миллионов отпечатков в базе. Прежде всего, вероятность того, что функция $f$ из **F** вычислит одинаковые хэш-коды для отпечатков двух разных пальцев, равна $(0.2)^6 = 0.000064$. Это так, потому что одинаковыми хэш-коды будут только тогда, когда на каждом отпечатке есть минуции во всех трех точках сетки, ассоциированных с $f$, а вероятность каждого из этих шести независимых событий равна 0.2.

Теперь посчитаем вероятность того, что для двух отпечатков одного пальца $f$ вычислит одинаковые хэш-коды. Вероятность того, что на первом отпечатке есть минуции во всех трех ячейках, ассоциированных с $f$, равна $(0.2)^3 = 0.008$. Но если это так, то вероятность того, что и на другом отпечатке имеются те же минуции, равна $(0.8)^3 = 0.512$. Таким образом, если два отпечатки сняты с одного пальца, то с вероятностью $0.008 \times 0.512 = 0.004096$ $f$ вычислит для них одинаковые хэш-коды. Это не много, всего один шанс из 200. Но если использовать много функций из **F** (но не слишком много), то мы сможем получить приемлемую вероятность отождествления отпечатков одного пальца с не слишком большим количеством ложноположительных результатов – одинаковых отпечатков, которые не отождествились

**Пример 3.22.** Случайным образом выберем 1024 функции из **F**. Затем построим новое семейство $\mathbf{F}_1$, применив к **F** 1024-путевое OR-построение. Тогда вероятность того, что хотя бы одна функция из $\mathbf{F}_1$ вычислит одинаковые хэш-коды для отпечатков одного и того же пальца, равна $1 - (1 - 0.004096)^{1024} = 0.985$. С другой стороны, вероятность того, что для отпечатков разных пальцев будут вычислены одинаковые хэш-коды, равна $(1 - (1 - 0.000064)^{1024} = 0.063$. Следовательно, мы получим примерно 1.5 % ложноотрицательных и примерно 6.3 % ложноположительных результатов.

Результат примера 3.22 – не лучшее, что можно сделать. Хотя шансов, что мы не сможем идентифицировать отпечаток пальца на пистолете всего 1.5 %, зато в 6.3 % случаев нам придется просматривать всю базу данных. Если увеличить количество функций из **F**, то увеличится количество ложноположительных результатов, а количество ложноотрицательных опустится едва-едва ниже 1.5 %. С другой стороны, можно использовать также AND-построение, это заметно сократит вероятность ложноположительного результата и лишь немного увеличит частоту ложноотрицательных. Например, можно было бы взять 2048 функций из **F**, разбив их на две группы по 1024. Вычислим хэш-коды с помощью каждой функции. Затем, имея отпечаток $P$, сделаем следующее:

1. Найдем хэш-коды из первой группы, совпадающие с вычисленными для $P$, и возьмем их объединение.
2. То же самое сделаем для второй группы.
3. Вычислим пересечение объединений.
4. Сравним $P$ только с отпечатками, попавшими в пересечение.

Отметим, что все равно приходится вычислять объединения и пересечения больших множеств отпечатков, но сравниваем мы только малую долю. А время уходит, прежде всего, на сравнение отпечатков; на шагах (1) и (2) отпечатки можно представить их целочисленными индексами в базе данных.

Если воспользоваться этой схемой, то вероятность найти совпадающий отпечаток будет равна $(0.985)^2 = 0.970$, т. е. мы получаем примерно 3 % ложноотрицательных результатов. Зато вероятность ложноположительного результата равна $(0.063)^2 = 0.00397$. То есть нам нужно будет просмотреть только 1/250 часть базы данных.

## 3.8.6. Похожие новости

И напоследок мы рассмотрим проблему организации большого репозитория новостей путем группировки веб-страниц, в основе которых лежит один и тот же базовый текст. Многие организации, например агентство Associated Press, направляют подготовленные ими новости в редакции различных газет. Каждая редакция помещает новость в онлайновое издание, но окружает его собственной информацией, например названием газеты, адресом редакции, ссылками на родственные материалы и на рекламные объявления. Кроме того, редакции часто модифицируют исходный материал, скажем, убирают последние несколько абзацев или даже удаляют текст из середины. В результате одна новость выглядит на сайтах разных газет совершенно по-разному.

Проблема представляется очень похожей на рассмотренную в разделе 3.4: найти такие документы, что коэффициент Жаккара для множеств их шинглов достаточно велик. Отметим, однако, что она отличается от задачи поиска новостей об одном и том же событии. В последнем случае требуются другие методы, обычно исследование множества важных слов в документах (о чем мы кратко упоминали в разделе 1.3.1) и их кластеризация с целью собрать в одну группу статьи на одну тему.

Однако для данных описанного типа оказалась более эффективной интересная модификация шинглов. Проблема в том, что при таком разбиении на шинглы, как описано в разделе 3.2, все части документа считаются равнозначными. Но мы-то хотим игнорировать некоторые части, например рекламу и заголовки других новостей, на которые редакция добавила ссылки, но которые к данной новости не относятся. Как выясняется, существует заметное различие между основным текстом новости и окружающими его рекламными объявлениями и заголовками других новостей. В основном тексте гораздо чаще встречаются стоп-слова, такие как «the» или «and». Общее число слов, считающихся стоп-словами, зависит от приложения, но, как правило, используется список, содержащий несколько сотен наиболее употребительных слов.

> **Пример 3.23.** Типичное объявление может быть весьма лаконичным: «Buy Sudzo» (купи Sudzo). С другой стороны, развернутое изложение той же мысли в тексте новости, возможно, выглядит так: «I recommend that you buy Sudzo for your laundry» (рекомендую вам покупать для стирки белья Sudzo). В этом предложении было бы естественно считать «I», «that», «you», «for» и «your» стоп-словами.

Определим шингл как стоп-слово, за которым следуют еще два слова. Тогда в объявлении «Buy Sudzo» нет ни одного шингла, поэтому оно не будет отражено в представлении объемлющей веб-страницы. С другой стороны, развернутое предложение было бы представлено пятью шинглами: «I recommend that», «that you buy», «you buy Sudzo», «for your laundry», и «your laundry $x$», где $x$ – слово, следующее за этим предложением.

Пусть есть две веб-страницы, состоящие наполовину из текста новости и наполовину из рекламы или других материалов с низкой плотностью стоп-слов. Если тексты новостей одинаковы, а окружающий материал различен, то можно ожидать, что значительная часть шинглов обеих страниц будет совпадать. Положим, что коэффициент Жаккара для них составляет 75 %. Если же окружающие материалы одинаковы, а содержание новостей различается, то количество общих шинглов будет мало, скажем 25 %. Если бы мы использовали стандартное разбиение на шинглы, когда шинглом считается, к примеру, последовательность из 10 соседних символов, то следовало бы ожидать совпадения половины шинглов документов (т. е. коэффициент Жаккара был бы равен 1/3) независимо от того, что совпадает: текст новости или окружающие материалы.

## 3.8.7. Упражнения к разделу 3.8

**Упражнение 3.8.1**. Предположим, что мы пытаемся отождествить библиографические ссылки и оцениваем пары ссылок, исходя из сходства названия, списка авторов и места публикации. Предположим еще, что в каждой ссылке имеется год публикации, который с равной вероятностью может быть любым из последних десяти лет. Предположим также, что для пар ссылок с максимальной

оценкой среднее расхождение в годе публикации равно 0.1[6]. Допустим, что для пар ссылок с некоторой оценкой *s* среднее расхождение в годе публикации равно 2. Какая доля пар с такой оценкой *s* действительно представляет одну и ту же публикацию? *Примечание:* было бы ошибкой предполагать, что среднее расхождение в датах публикации между случайными парами равно 5 или 5.5. Вы должны вычислить его точно, и у вас для этого достаточно информации.

**Упражнение 3.8.2.** Пусть используется семейство **F** функций, описанное в разделе 3.8.5, где вероятность присутствия минуции в ячейке сетки равна 20 %, а вероятность, что для другого отпечатка того же пальца в той же ячейке будет присутствовать минуция, равна 80 %. И пусть каждая функция в **F** строится из трех ячеек. В примере 3.22 мы строили семейство $\mathbf{F}_1$, применяя процедуру OR-построения к 1024 членам **F**. Теперь рассмотрим семейство $\mathbf{F}_2$, полученное применением OR-построения к 2048 членам **F**.

(*а*) Вычислите частоты ложноположительных и ложноотрицательных результатов для $\mathbf{F}_2$.

(*б*) Как эти частоты соотносятся с теми, что получаются при организации тех же 2048 функций в виде двухпутевого AND-построения, примененного к членам $\mathbf{F}_1$, как было описано в конце раздела 3.8.5?

**Упражнение 3.8.3.** Пусть статистическое распределение отпечатков пальцев такое же, как в упражнении 3.8.2, но в качестве базового семейства $\mathbf{F}'$ используется семейство, аналогичное **F**, но каждая функция строится только по двум случайно выбранным ячейкам сетки. Постройте по $\mathbf{F}'$ другое множество функций $\mathbf{F}'_1$, выполнив *n*-путевое OR-построение. Выразите частоты ложноположительных и ложноотрицательных результатов для $\mathbf{F}'_1$ в виде функции от *n*.

**Упражнение 3.8.4.** Пусть используется семейство функций $\mathbf{F}_1$ из примера 3.22, но мы хотим решить задачу многие-с-многими.

(*а*) Какова вероятность, что сравнение двух отпечатков одного и того же пальца завершится неудачно (т. е. какова частота ложноотрицательных результатов)?

(*б*) Какая доля двух отпечатков разных пальцев, признанных одинаковыми (т. е. какова частота ложноположительных результатов)?

**! Упражнение 3.8.5.** Пусть есть два множества функций **F**, как в упражнении 3.8.2, и мы строим новое множество функций $\mathbf{F}_3$ путем применения *n*-путевого OR-построения к функциям из **F**. Для какого значения *n* сумма частот ложноположительных и ложноотрицательных результатов достигает минимума?

# 3.9. Методы для высокой степени сходства

Построенные на основе LSH методы наиболее эффективны, когда степень сходства относительно мала. Для поиска почти идентичных множеств существуют дру-

[6] Можно было ожидать расхождения 0, но на практике год публикации иногда указывается неверно.

гие, более быстрые, методы. К тому же, эти методы точны в том смысле, что находят все пары объектов с требуемой степенью сходства. Ложноотрицательных результатов, как в случае LSH, вообще не бывает.

## 3.9.1. Поиск одинаковых объектов

Крайним случаем является поиск одинаковых объектов, например веб-страниц, совпадающих с точностью до символа. Нетрудно сравнить два документа и узнать, совпадают они или нет, но все равно необходимо избежать сравнения каждой пары документов. Первое, что приходит в голову, – хэшировать документы на основе нескольких первых символов и сравнивать только те, что попали в одну ячейку. И эта схема должна работать неплохо, если только все документы не начинаются одинаково, например HTML-заголовком.

Следующая идея – использовать хэш-функцию, которая просматривает весь документ. Это будет работать и, если взять достаточно, ячеек, то ситуация, когда два разных документа попадают в одну ячейку, будет встречаться редко. Недостаток в том, что нужно обрабатывать каждый символ каждого документа. Если же ограничиться только небольшим числом символов, то не придется исследовать уникальный документ, который оказался в своей ячейке в одиночестве.

Более разумный подход – выбрать фиксированный набор случайных позиций для всех документов и только их и использовать для хэширования. Тогда мы, с одной стороны, обойдем проблему общего префикса у всех или большинства документов, а, с другой, не будем просматривать целиком документы, оказавшиеся в своей ячейке в одиночестве. Правда, есть одна проблема: в коротких документах символы могут присутствовать не во всех выбранных позициях. Но если мы ищем очень похожие документы, то сравнивать документы, заметно отличающиеся по длине, все равно не имеет смысла. Эту идею мы разовьем в разделе 3.9.3.

## 3.9.2. Представление множеств в виде строк

Теперь займемся более трудной задачей: поиском в больших наборах множеств всех пар с высоким коэффициентом Жаккара, например 0.9. Мы можем представить множество следующим образом: отсортировать элементы универсального множества в каком-то фиксированном порядке и перечислить элементы данного множества в этом порядке. Такой список, по существу, является строкой «символов», где под символами понимаются элементы универсального множества. Но эти строки необычны в двух отношениях.

1. Каждый символ входит в строку не более одного раза.
2. Два символа, встречающиеся в двух разных строках, расположены в одном и том же порядке.

**Пример 3.24.** Пусть универсальное множество состоит из 26 строчных букв латинского алфавита и используется обычный алфавитный порядок. Тогда множество {*d*, *a*, *b*} представляется строкой `abd`.

Далее мы будем предполагать, что любая строка представляет некоторое множество описанным выше способом. Поэтому будем говорить о коэффициенте Жаккара строк, имея в виду сходство представленных этими строками множеств. Кроме того, говоря о длине строке, мы будем иметь в виду количество элементов в соответствующем множестве.

Отметим, что документы, обсуждавшиеся в разделе 3.9.1, не точно согласуются с этой моделью, хотя их и можно рассматривать. Чтобы устранить рассогласование, мы можем разбить документы на шинглы, приписать шинглам некоторый порядок и представить каждый документ списком его шинглов в выбранном порядке.

## 3.9.3. Фильтрация на основе длины строки

Воспользоваться строковым представлением из раздела 3.9.2 проще всего, отсортировав строки по длине. Тогда каждая строка $s$ сравнивается только с теми строками $t$, которые следуют за ней в списке, но не являются слишком длинными. Допустим, что задана верхняя граница $J$ для расстояния Жаккара между двумя строками. Обозначим $L_x$ длину строки $x$. Отметим, что $L_s \leq L_t$. Пересечение множеств, представленных строками $s$ и $t$, не может содержать больше $L_s$ элементов, а их объединение содержит не менее $L_t$ элементов. Следовательно, коэффициент Жаккара для строк $s$ и $t$, который мы обозначим SIM($s$, $t$), не больше $L_s/L_t$. То есть сравнивать $s$ и $t$ нам придется, только если $1 - J \leq L_s/L_t$, или $L_t \leq L_s/(1 - J)$.

> **Пример 3.25.** Пусть $s$ – строка длины 9, и мы ищем строки, для которых коэффициент Жаккара не меньше 0.9. Тогда $s$ нужно сравнивать только со следующими (относительно порядка сортировки по длине) за ней строками, длины которых не больше 9/0.9 = 10. То есть мы сравниваем $s$ со следующими за ней строками длины 9 и со всеми строками длины 10. Сравнивать с чем-то еще нет нужды.
>
> Предположим теперь, что длина $s$ равна 8. Тогда $s$ следовало бы сравнивать со всеми строкам длины не более 8/0.9 = 8.89. Это значит, что строки длиной 9 слишком длинные, т. е. коэффициент Жаккара для них будет заведомо меньше 0.9. Поэтому сравнивать нужно только со строками длины 8, следующими за $s$ в порядке сортировки.

## 3.9.4. Префиксное индексирование

Помимо длины, есть и другие свойства строк, которыми можно воспользоваться, чтобы ограничить количество сравнений, необходимых для выявления всех пар похожих строк. Простейший способ – создать индекс для каждого символа; напомним, что символами строк называются элементы универсального множества. Для каждой строки $s$ возьмем ее префикс, состоящий из первых $p$ символов. Насколько большим брать $p$, зависит от $L_s$ и $J$, нижней границы расстояния Жаккара. Строка $s$ добавляется в индекс для каждого из ее первых $p$ символов.

По сути дела, индекс для каждого символа представляет собой ячейку, которая содержит строки, подлежащие сравнению. Мы должны быть уверены, что любая

строка $t$ такая, что SIM($s$, $t$) $\geq J$, содержит в префиксе хотя бы один символ, встречающийся также в префиксе $s$.

Предположим, что это не так, т. е. SIM($s$, $t$) $\geq J$, но в $t$ нет ни одного из первых $p$ символов $s$. Тогда наибольший возможный коэффициент Жаккара строк $s$ и $t$ достигается, когда $t$ является суффиксом $s$, содержащим любые символы, кроме первых $p$ символов $s$. В таком случае коэффициент Жаккара будет равен $(L_s - p)/L_s$. Мы можем быть уверены в том, что $s$ не нужно сравнивать с $t$, только если $J > (L_s - p)/L_s$. Иначе говоря, $p$ должно быть не меньше $\lfloor (1 - J)L_s \rfloor + 1$. Разумеется, нас интересует наименьшее возможное $p$, чтобы не включать строку $s$ в больше индексных ячеек, чем необходимо. Таким образом, в качестве длины индексируемого префикса мы выбираем $p = \lfloor (1 - J)L_s \rfloor + 1$.

### Улучшенное упорядочение символов

Вместо использования очевидного порядка на элементах универсального множества, например лексикографического упорядочения шинглов, мы можем произвести упорядочение по количеству вхождений. То есть определяем, сколько раз каждый элемент встречается в наборе множеств, и упорядочиваем по этому числу, начиная с наименьшего. Это удобно тем, что символы, входящие в префиксы, обычно встречаются редко. Поэтому соответствующая строка будет помещена в индексные ячейки со сравнительно небольшим числом членов. Впоследствии, когда нужно будет устанавливать сходство строк, мы найдем не так много кандидатов для сравнения.

**Пример 3.26.** Пусть $J = 0.9$. Если $L_s = 9$, то $p = \lfloor 0.1 \times 9 \rfloor + 1 = \lfloor 0.9 \rfloor + 1 = 1$. Это означает, что $s$ нужно индексировать только по первому символу. Для любой строки $t$, которая не содержит первый символ $s$ в такой позиции, что $t$ индексируется по этому символу, коэффициент Жаккара между ней и $s$ будет меньше 0.9. Пусть $s$ равна `bcdefghij`. Тогда $s$ индексируется только по `b`. Предположим, что $t$ начинается не с `b`. Необходимо рассмотреть два случая.

1. Если $t$ начинается с `a` и SIM($s$, $t$) $\geq 0.9$, то $t$ может быть равна только `abcdefghij`. Но если это так, то $t$ будет индексирована по `a` и по `b`. Объясняется это тем, что $L_t = 10$, поэтому $t$ будет индексирована по символам своего префикса длины $\lfloor 0.1 - 10 \rfloor + 1 = 2$.
2. Если $t$ начинается с `c` или одной из следующих за ней букв, то максимум SIM($s$, $t$) достигается, когда $t$ равна `cdefghij`. Но тогда SIM($s$, $t$) = 8/9 < 0.9.

В общем случае при $J = 0.9$ строки длины не более 9 индексируются по первому символу, строки длины от 10 до 19 – по первым двум символам, строки длины от 20 до 29 – по первым трем символам и т. д.

Эту схему индексирования можно использовать двумя способами в зависимости от того, какую задачу мы решаем: многие-с-многими или многие-с-одним

(определение было дано в разделе 3.8.4). Для задачи многие-с-одним создается индекс для всей базы данных. Когда поступает запрос на идентификацию нового множества $S$, мы преобразуем множество в строку $s$, называемую *пробной*. Определяем, префиксы какой длины нужно просматривать, т. е. вычисляем $\lfloor (1 - J)L_s \rfloor + 1$. Для каждого символа, встречающегося в одной из префиксных позиций $s$, находим в индексе соответствующую этому символу ячейку и сравниваем $s$ со всеми попавшими в нее строками.

Если мы хотим решить задачу многие-со-многими, то начать надо с пустой базы строк и индексов. Каждое множество $S$ рассматривается как новое множество для задачи многие-с-одним. Преобразуем $S$ в строку $s$, считая ее пробной строкой в задаче многие-с-одним. Но после исследования индексной ячейки мы еще и добавляем $s$ в эту ячейку, так что $s$ будет сравниваться со всеми последующими строками, которые потенциально могли бы сопоставиться.

## 3.9.5. Использование информации о позиции

Рассмотрим строки $s$ = `acdefghijk` и $t$ = `bcdefghijk` и предположим, что $J = 0.9$. Поскольку длины обеих строк равны 10, то они индексируются по первым двум символам. То есть $s$ индексируется по `a` и `c`, а $t$ – по `b` и `c`. Какая бы строка ни была добавлена последней, другая будет найдена в ячейке для `c`, и эти строки будут сравниваться. Но если `c` – второй символ обеих строк, то мы знаем, что существуют два символа, в данном случае `a` и `b`, которые входят в объединение, но не входят в пересечение множеств. Действительно, хотя $s$ и $t$ совпадают, начиная с символа `c`, их пересечение содержит 9 символов, а объединение – 11; следовательно, $\text{SIM}(s, t) = 9/11$, а это меньше 0.9.

Если при построении индекса учитывать не только символ, но и позицию символа внутри строки, то можно будет избежать описанного выше сравнения $s$ и $t$. Это означает, что в индексе имеется ячейка для каждой пары $(x, i)$, содержащая строки, для которых в позиции $i$ префикса находится символ $x$. Имея строку $s$ и предполагая, что $J$ – минимальное желаемое расстояние Жаккара, мы просматриваем префикс $s$, т. е. позиции от 1 до $\lfloor (1 - J)L_s \rfloor + 1$. Если символ в позиции $i$ префикса равен $x$, то добавляем $s$ в индексную ячейку для $(x, i)$.

Теперь рассмотрим $s$ как пробную строку. С какими ячейками ее необходимо сравнить? Мы будем просматривать символы префикса $s$ слева направо и воспользуемся тем фактом, что искать возможное совпадение $t$ нужно только в том случае, когда ни в одной из ранее просмотренных ячеек нет $t$. То есть мы должны найти кандидата на совпадение только один раз. Таким образом, если оказалось, что $i$-ый символ $s$ равен $x$, то нужно просматривать ячейки $(x, j)$ для небольших значений $j$.

Чтобы вычислить верхнюю границу $j$, предположим, что строка $t$ такова, что ни один из ее первых $j - 1$ символов не совпадает ни с одним символом $s$, но $j$-й символ $t$ совпадает с $i$-ым символом $s$. $\text{SIM}(s, t)$ достигает максимума, когда $s$ и $t$ совпадают после $i$-го и $j$-го символов соответственно, как показано на рис. 3.14. Если это так, то размер их пересечения равен $L_s - i + 1$, поскольку это и есть количество символов $s$, которые могли бы встречаться и в $t$. Размер объединения не меньше $L_s + j - 1$.

Это означает, что $s$ заведомо вносит $L_s$ символов в объединение и существует как минимум $j - 1$ символов $t$, не входящих в $s$. Поскольку отношение размеров пересечения и объединения должно быть не меньше $J$, имеем:

$$\frac{L_S - i + 1}{L_S + i - 1} \geq J .$$

Разрешая это неравенство относительно $j$, получаем: $j \leq (L_s(1 - J) - i + 1 + J)/J$.

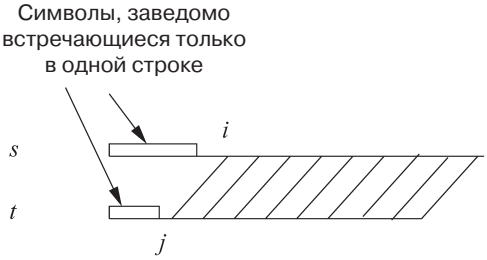



**Рис. 3.14.** Строки $s$ и $t$ начинаются соответственно $i - 1$ и $j - 1$ уникальными символами, а затем совпадают

**Пример 3.27.** Рассмотрим строку $s$ = `acdefghijk` при $J = 0.9$, как в начале этого раздела. Предположим, что $s$ – пробная строка. Мы уже установили, что нужно просматривать только первые две позиции, т. е. $i$ может быть равно 1 или 2. Пусть $i = 1$. Тогда $j \leq (10 \times 0.1 - 1 + 1 + 0.9)/0.9$. Следовательно, нам нужно только сравнить символ `a` со строками в ячейках (`a`, $j$) для $j$ 2.11, т. е. $j$ может быть равно 1 или 2, но не больше.

Пусть теперь $i = 2$. Тогда требуется, чтобы $j \leq (10 \times 0.1 - 2 + 1 + 0.9)/0.9$, т. е. $j \leq 1$. Стало быть, нужно заглянуть только в ячейки (`a`, 1), (`a`, 2) и (`c`, 1) и ни в какие другие. Для сравнения, если использовать ячейки из раздела 3.9.4, то придется просматривать ячейки для `a` и `c`, что эквивалентно просмотру ячеек (`a`, $j$) и (`c`, $j$) для всех $j$.

## 3.9.6. Использование позиции и длины в индексах

Рассматривая верхнюю границу $j$ в предыдущем разделе, мы предполагали, что символы в позициях строк $s$ и $t$ с номерами, большими $i$ и $j$ соответственно, в точности совпадают (см. рис. 3.14). Мы не хотим включать в индекс все символы строк, потому что это неподъемная работа. Но мы можем добавить в индекс сводную информацию о том, что следует за индексируемыми позициями. При этом количество ячеек увеличится не катастрофически, зато мы сможем исключить многих кандидатов, не сравнивая строки целиком. Идея в том, чтобы каждая ячейка индекса соответствовала тройке: символ, позиция и *длина суффикса*, т. е. число символов после рассматриваемой позиции.

**Пример 3.28.** Строка $s$ = acdefghijk при $J$ = 0.9 попадет в индексные ячейки (a, 1, 9) и (c, 2, 8). Это означает, что первый символ $s$ – это a, а длина его суффикса равна 9. Во второй позиции находится символ c с длиной суффикса 8.

На рис. 3.14 предполагается, что суффиксы для позиции $i$ строки $s$ и позиции $j$ строки $t$ имеют одинаковую длину. Если это не так, то мы можем либо уменьшить верхнюю границу размера пересечения $s$ и $t$ (если $t$ короче), либо увеличить нижнюю границу размера объединения (если $t$ длиннее). Пусть длина суффикса $s$ равна $p$, а длина суффикса $t$ равна $q$.

**Случай 1.** $p \geq q$. Тогда максимальный размер пересечения равен

$$L_s - i + 1 - (p - q).$$

Поскольку $L_s = i + p$, мы можем переписать это выражение в виде $q + 1$. Минимальный размер объединения равен $L_s + j - 1$, как и было, когда мы не принимали в расчет длину суффикса. Следовательно, мы требуем выполнения условия

$$\frac{q+1}{L_S + i - 1} \geq J$$

при всех $p \geq q$.

**Случай 2.** $p < q$. Тогда максимальный размер пересечения равен $L_s - i + 1$, как было, когда длина суффикса не учитывалась. Однако минимальный размер объединения теперь равен $L_s + j - 1 + q - p$. Снова воспользовавшись соотношением $L_s = i + p$, мы сможем заменить $L_s - p$ на $i$ и выразить размер объединения в виде $i + j - 1 + q$. Если коэффициент Жаккара не меньше $J$, то

$$\frac{L_S - i + 1}{i + j - 1 + q} \geq J$$

при всех $p < q$.

**Пример 3.29.** Пусть снова $s$ = acdefghijk, но для демонстрации некоторых деталей возьмем $J$ = 0.8, а не 0.9. Мы знаем, что $L_s$ = 10. Поскольку $\lfloor (1 - J) L_s \rfloor + 1 = 3$, то далее необходимо будет рассматривать префиксные позиции $i$ = 1, 2, 3. Как и раньше, пусть $p$ – длина суффикса $s$, а $q$ – длина суффикса $t$.

Сначала рассмотрим случай $p \geq q$. Дополнительное ограничение на $q$ и $j$ имеет вид $(q + 1)/(9 + j) \geq 0.8$. Мы можем следующим образом перебрать пары значений $j$ и $q$ для каждого $i$ от 1 до 3.

$i$ = 1: здесь $p$ = 9, значит, $q \leq 9$. Рассмотрим все возможные значения $q$:
- $q$ = 9: должно быть $10/(9 + j) \geq 0.8$. Следовательно, возможны значения $j$ = 1, $j$ = 2, $j$ = 3. Отметим, что для $j$ = 4 имеем $10/13 > 0.8$.
- $q$ = 8: должно быть $9/(9 + j) \geq 0.8$. Следовательно, возможны значения $j$ = 1 или $j$ = 2. Для $j$ = 3 имеем $9/12 > 0.8$.
- $q$ = 7: должно быть $8/(9+j) \geq 0.8$. Этому неравенству удовлетворяет только $j$ = 1.
- $q$ = 6: нет ни одного возможного значения $j$, потому что $7/(9 + j) > 0.8$ для всех целых положительных $j$. То же самое справедливо для меньших значений $q$.

$i = 2$: здесь $p = 8$, значит, мы требуем, чтобы $q \le 8$. Поскольку ограничение $(q+1)/(9+j) \ge 0.8$ не зависит от $i$,[7] мы можем воспользоваться анализом из предыдущего случая, исключив значение $q = 9$. Следовательно, при $i = 2$ возможны только такие значения $j$ и $q$:

1. $q = 8$; $j = 1$.
2. $q = 8$; $j = 2$.
3. $q = 7$; $j = 1$.

$i = 3$: теперь $p = 7$ и ограничения имеют вид $q \le 7$ и $(q+1)/(9+j) \ge 0.8$. Возможен только один вариант: $q = 7$ и $j = 1$.

Далее следует рассмотреть случай $p < q$. Дополнительное ограничение имеет вид

$$\frac{11-i}{i+j+q-1} \ge 0.8 .$$

И снова рассмотрим все возможные значения $i$.

$i = 1$: тогда $p = 9$, поэтому мы требуем, чтобы $q \ge 10$ и $10/(q+j) \ge 0.8$. Возможны такие значения $q$ и $j$:

1. $q = 10$; $j = 1$.
2. $q = 10$; $j = 2$.
3. $q = 11$; $j = 1$.

$i = 2$: теперь $p = 10$, поэтому требуем, чтобы $q \ge 11$ и $9/(q+j+1) \ge 0.8$. У этой системы неравенств нет решений, т. к. $j$ должно быть целым положительным числом.

$i = 3$: как и для $i = 2$, решений нет.

Собрав воедино возможные комбинации $i$, $j$ и $q$, мы видим, что множество индексных ячеек, которые необходимо просматривать, образует пирамиду. На рис. 3.15 показано, в каких ячейках $(x, j, q)$ нужно искать. Напомним, что так обозначается ячейка, для которой $i$-й символ строки $s$ равен $x$, $j$ – ассоциированная с ячейкой позиция, а $q$ – длина суффикса.

| | $q$ | $j=1$ | $j=2$ | $j=3$ |
|---|---|---|---|---|
| $i=1$ | 7 | x | | |
| | 8 | x | x | |
| | 9 | x | x | x |
| | 10 | x | x | |
| | 11 | x | | |
| $i=2$ | 7 | x | | |
| | 8 | x | x | |
| | 9 | x | | |
| $i=3$ | 7 | x | | |

**Рис. 3.15.** Буквой x помечены ячейки, которые необходимо просматривать для поиска возможных совпадений со строкой $s$ = `acdefghijk` при $J = 0.8$

[7] Отметим, что $i$ влияет на значение $p$ и опосредованно ограничивает величину $q$.

### 3.9.7. Упражнения к разделу 3.9

**Упражнение 3.9.1.** Допустим, что универсальным множеством является множество строчных букв и что принят следующий порядок: сначала гласные, в алфавитном порядке, затем согласные в обратном алфавитном порядке. Представьте в виде строк следующие множества:

(*а*) {*q*,*w*, *e*, *r*, *t*, *y*}.
(*б*) {*a*, *s*, *d*, *f*, *g*, *h*, *j*, *u*, *i*}.

**Упражнение 3.9.2.** Допустим, что пары-кандидаты фильтруются только на основе длины, как в разделе 3.9.3. Если *s* – строка длины 20, то с какими строками сравнивается *s*, если нижняя граница коэффициента Жаккара *J* принимает следующие значения: (*а*) $J = 0.85$, (*б*) $J = 0.95$, (*в*) $J = 0.98$?

**Упражнение 3.9.3.** Пусть имеется строка *s* длины 15, и мы хотим индексировать ее префикс, как в разделе 3.9.4.

(*а*) Сколько позиций будет в префиксе, если $J = 0.85$?
(*б*) Сколько позиций будет в префиксе, если $J = 0.95$?
! (*в*) При каком диапазоне значений *J* строка *s* будет индексирована по первым четырем символам, но не более?

**Упражнение 3.9.4.** Пусть *s* – строка длины 12. С какими парами символ-позиция будет сравниваться *s* при использовании описанного в разделе 3.9.5 подхода к индексированию, когда: (*а*) $J = 0.75$, (*б*) $J = 0.95$?

! **Упражнение 3.9.5.** Предположим, что при построении индекса использована информация о позиции, как описано в разделе 3.9.5. Строки *s* и *t* случайно выбираются из универсального множества, содержащего 100 элементов. Пусть $J = 0.9$. С какой вероятностью будут сравниваться *s* и *t*, если:

(*а*) длина *s* и *t* равна 9, (*b*) длина *s* и *t* равна 10.

**Упражнение 3.9.6.** Пусть используются индексы по позиции и длине суффикса, как в разделе 3.9.6. Если *s* – строка длины 20, то с какими тройками символ-позиция-длина сопоставится *s*, если: (*а*) $J = 0.8$, (*б*) $J = 0.9$?

## 3.10. Резюме

- *Коэффициент Жаккара*. Коэффициентом Жаккара двух множеств называется отношение размера их пересечения к размеру объединения. Эта мера сходства удобна во многих приложениях, в том числе для оценки текстуального сходства документов и пристрастий покупателей.
- *Разбиение на шинглы*. *k*-шинглом называется любая последовательность *k* соседних символов документа. Если представить документ множеством его *k*-шинглов, то коэффициент Жаккара таких множеств измеряет текстуальное сходство документов. Иногда полезно хэшировать шинглы, заменяя их битовой строкой меньшей длины, и использовать для представления документов множества хэш-кодов.

- *Минхэши*. Минхэш-функция на множествах основана на перестановке элементов универсального множества. Если имеется такая перестановка, то значение минхэша множества равно тому элементу множества, который находится на первом месте в порядке перестановки.
- *Минхэш-сигнатуры*. Чтобы представить множество, можно выбрать какой-то набор перестановок и для каждого множества вычислить его минхэш-сигнатуру, т. е. последовательность минхэш-значений, полученную применением каждой перестановки из этого набора к множеству. Если даны два множества, то ожидаемая доля перестановок, которые приводят к одинаковым минхэш-значениям, в точности равна коэффициенту Жаккара этих множеств.
- *Эффективное вычисление минхэшей*. Поскольку генерировать случайные перестановки практически нереально, обычно перестановку моделируют, выбирая случайную хэш-функцию и принимая в качестве минхэш-значения множества наименьший хэш-код, вычисленный по всем его элементам.
- *Хэширование сигнатур с учетом близости*. Эта техника позволяет избежать вычисления сходства каждой пары множеств или их минхэш-сигнатур. Если известны сигнатуры множеств, то их можно разбить на полосы и измерять сходство пары множеств только в том случае, когда они совпадают хотя бы в одной полосе. Путем правильного подбора размера полос мы можем исключить из рассмотрения большинство пар со сходством меньше порогового.
- *Метрики*. Метрикой, или расстоянием называется функция, определенная на парах точек пространства, удовлетворяющая некоторым аксиомам. Расстояние между точками равно 0, если они совпадают, и положительно в противном случае. Расстояние симметрично, т. е. не зависит от порядка точек. Метрика должна удовлетворять неравенству треугольника: расстояние между двумя точками не больше суммы расстояний от каждой из этих точек до какой-то третьей.
- *Евклидово расстояние*. Самый известный пример расстояния – евклидово расстояние в $n$-мерном пространстве, иногда называемое $L_2$-нормой. Оно равно квадратному корню из суммы квадратов разностей между координатами точек по каждому измерению. Еще одно расстояние, пригодное для евклидовых пространств, называется манхэттенским расстоянием, или $L_1$-нормой. Оно равно сумме абсолютных величин разностей между координатами точек по каждому измерению.
- *Расстояние Жаккара*. Так называется функция, равная единице минус коэффициент Жаккара.
- *Косинусное расстояние*. Угол между векторами в векторном пространстве называется косинусным расстоянием. Чтобы вычислить косинус этого угла, нужно разделить скалярное произведение векторов на произведение их длин.

- *Редакционное расстояние*. Это расстояние в пространстве строк, оно равно минимальному количеству операций вставки или удаления, необходимых для преобразования одной строки в другую. Редакционное расстояние можно также вычислить как сумму длин строк за вычетом удвоенной длины самой длинной общей подстроки.
- *Расстояние Хэмминга*. Это расстояние между двумя векторами равно количеству позиций, в которых они различаются.
- *Обобщенное хэширование с учетом близости*. Мы можем начать с любого набора функций, например минхэш-функций, которые принимают решение о том, являются ли два объекта кандидатами для сравнения на сходство. Единственное ограничение заключается в том, что такие функции должны ограничивать снизу вероятность ответа «да», если расстояние (согласно какой-то метрике) меньше определенной величины, и ограничивать сверху вероятность ответа «да», если расстояние больше другой, тоже наперед заданной, величины. В таком случае мы можем увеличить вероятность ответа «да» для близких объектов, уменьшив в то же время вероятность ответа «да» для далеких объектов, настолько, насколько захотим, применяя процедуры AND- и OR-построения.
- *Случайные гиперплоскости и LSH-хэширование для косинусного расстояния*. Мы можем получить множество базисных функций для построения обобщенного LSH-семейства функций для косинусного расстояния, определив каждую функцию списком случайных векторов. Чтобы применить такую функцию к заданному вектору $v$, мы вычисляем скалярные произведения $v$ с каждым вектором из списка. Результатом является эскиз, состоящий из знаков этих скалярных произведений (+1 или –1). Доля позиций, в которых эскизы двух векторов совпадают, умноженная на 180, дает оценку угла между векторами.
- *LSH-хэширование для евклидова расстояния*. Набор базисных функций для построения LSH-семейства для евклидова расстояния можно получить, проецируя точки на случайно выбранные прямые. Каждая прямая разбивается на интервалы фиксированной длины, и функция отвечает «да», если пара точки попадает в один и тот же интервал.
- *Определение высокой степени сходства путем сравнения строк*. В случае, когда пороговая величина коэффициента Жаккара близка к 1, можно применить другой подход к поиску похожих объектов, позволяющий избежать вычисления минхэшей и LSH-хэширования. Вместо этого на универсальном множестве определяется некоторый порядок, а множества представляются строками, содержащими их элементы в этом порядке. Чтобы избежать сравнения всех пар множеств или соответствующих им строк, сделаем простое наблюдение: в случае, когда множества очень похожи, их строки имеют примерно одинаковую длину. Если отсортировать строки, то можно будет сравнивать каждую строку только с небольшим числом строк, следующих непосредственно за ней.

- *Индексы по символам.* Если множества представлены строками и порог сходства близок к 1, то можно проиндексировать все строки по нескольким первым символам. Длина индексируемого префикса приблизительно равна длине строки, умноженной на максимальное расстояние Жаккара.
- *Индексы по позиции.* Строки можно индексировать не только по символам префикса, но и по позиции символа внутри префикса. При этом уменьшается количество пар строк, подлежащих сравнению, потому что если у двух строк есть общий символ, который находится не в первой позиции каждой строки, то мы точно знаем, что либо некоторые предшествующие ему символы принадлежат объединению, но не пересечению, либо ему предшествует какой-то символ, встречающийся в обеих строках.
- *Индексы по суффиксу.* Строки можно индексировать не только по символам префикса и их позициям, но и по длине суффикса символа – количеству следующих за ним символов строки. Это позволяет еще уменьшить количество подлежащих сравнению пар, потому что из наличия общего символа с разной длиной суффикса следует, что существуют дополнительные символы, присутствующие в объединении, но не в пересечении.

# 3.11. Список литературы

Считается, что метод разбиения на шинглы впервые был описан в работе [10]. Способ, рассмотренный нами, заимствован из [2]. Идея минхэшей изложена в [3]. Оригинальные работы о хэшировании с учетом близости – [9] и [7]. Работа [1] содержит полезный обзор идей в этой области.

В работе [4] вводится идея использования случайных гиперплоскостей для хэширования элементов с учетом близости относительно косинусного расстояния. В [8] высказано предположение о том, что применение гиперплоскостей совместно с LSH позволит распознавать похожие документы точнее, чем комбинации минхэшей и LSH.

Методы хэширования точек в евклидовом пространстве рассматриваются в [6]. В работе [11] описана техника разбиения на шинглы на основе стоп-слов.

Схемы индексирования по префиксу и по позиции для поиска очень похожих объектов взяты из [5]. Включение в индекс длины суффикса описано в [12].

1. A. Andoni, P. Indyk «Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions», Comm. ACM 51:1, pp. 117–122, 2008.
2. A. Z. Broder «On the resemblance and containment of documents», Proc. Compression and Complexity of Sequences, pp. 21–29, Positano Italy, 1997.
3. A. Z. Broder, M. Charikar, A. M. Frieze, M. Mitzenmacher «Min-wise independent permutations», ACM Symposium on Theory of Computing, pp. 327–336, 1998.
4. M. S. Charikar «Similarity estimation techniques from rounding algorithms», ACM Symposium on Theory of Computing, pp. 380–388, 2002.
5. S. Chaudhuri, V. Ganti, R. Kaushik «A primitive operator for similarity joins in

data cleaning», Proc. Intl. Conf. on Data Engineering, 2006.
6. M. Datar, N. Immorlica, P. Indyk, V. S. Mirrokni «Locality-sensitive hashing scheme based on p-stable distributions», Symposium on Computational Geometry pp. 253–262, 2004.
7. A. Gionis, P. Indyk, R. Motwani «Similarity search in high dimensions via hashing», Proc. Intl. Conf. on Very Large Databases, pp. 518–529, 1999.
8. M. Henzinger «Finding near-duplicate web pages: a large-scale evaluation of algorithms», Proc. 29th SIGIR Conf., pp. 284–291, 2006.
9. P. Indyk, R. Motwani «Approximate nearest neighbor: towards removing the curse of dimensionality», ACM Symposium on Theory of Computing, pp. 604–613, 1998.
10. U. Manber «Finding similar files in a large file system», Proc. USENIX Conference, pp. 1–10, 1994.
11. M. Theobald, J. Siddharth, A. Paepcke «SpotSigs: robust and efficient near duplicate detection in large web collections», 31st Annual ACM SIGIR Conference, July, 2008, Singapore.
12. C. Xiao, W. Wang, X. Lin, J.X. Yu «Efficient similarity joins for near duplicate detection», Proc. WWW Conference, pp. 131-140, 2008.

# ГЛАВА 4.
# Анализ потоков данных

В большинстве описываемых в этой книге алгоритмов предполагается, что анализу подвергается какая-то база данных. То есть когда возникает нужда в данных, они имеются в наличии. В этой главе мы примем другое предположение: данные поступают в виде одного или нескольких потоков и, если не обработать или не сохранить их немедленно, то они пропадут навсегда. Более того, мы предполагаем, что данные поступают так быстро, что практически нереально сохранить их все в активном хранилище (т. е. традиционной базе данных), а обрабатывать позже, когда будет удобно.

Все алгоритмы обработки потоков подразумевают то или иное обобщение данных. Мы начнем с вопроса о том, как сделать полезную выборку из потока и как отфильтровать поток, исключив большинство «нежелательных» элементов. Затем мы покажем, как оценить число различных элементов в потоке, используя гораздо меньше памяти, чем потребовалось бы для хранения списка всех встречавшихся элементов.

Другой подход к обобщению потока заключается в исследовании только «окна» фиксированного размера, содержащего последние $n$ элементов, где $n$ обычно большое число. Затем окно опрашивается так, будто это отношение в базе данных. Если потоков много и/или $n$ велико, то мы не сможем сохранить окна целиком для каждого потока, поэтому обобщать придется даже окна. Мы решим фундаментальную проблему подсчета приблизительного числа единиц в окне битового потока, используя гораздо меньше памяти, чем потребовалось бы для хранения всего окна. Эта техника обобщается на аппроксимацию сумм различного вида.

## 4.1. Потоковая модель данных

Начнем с обсуждения основ потоков и потоковой обработки. Мы объясним, в чем различие между потоками и базами данных, и увидим, какие специфические проблемы возникают при работе с потоками. Будут рассмотрены некоторые типичные приложения, к которым применима потоковая модель.

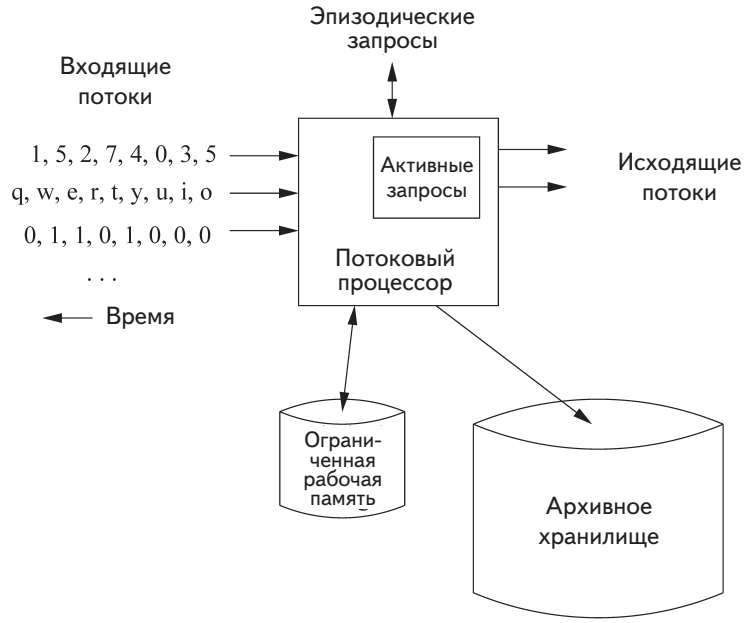



**Рис. 4.1.** Система управления потоками данных

## 4.1.1. Система управления потоками данных

По аналогии с системой управления базами данных мы можем рассматривать потоковый процессор как своего рода систему управления данными. На рис. 4.1 показано, как она организована на верхнем уровне. В систему может входить любое число потоков. Каждый поток может поставлять элементы по своему расписанию; и темп передачи, и типы данных могут быть различны, время между поступлением элементов одного потока тоже не обязательно одинаковое. Тот факт, что скорость поступления элементов потока не контролируется системой, отличает обработку потоков от обработки данных из СУБД, когда система контролирует скорость чтения данных с диска и, следовательно, может не заботиться о том, что данные потеряются, пока она будет выполнять запрос.

Потоки можно архивировать в большом архивном хранилище, но мы предполагаем, что оно не может использоваться для обслуживания запросов. К нему можно обращаться только в особых случаях с помощью долго работающих процессов извлечения данных. Существует также рабочая память, в которую можно помещать обобщенные данные или части потоков; вот она может использоваться для получения ответов на запросы. Рабочая память может располагаться на диске или в оперативной памяти в зависимости от того, как быстро нужно обрабатывать запросы. Но в любом случае ее емкость ограничена, и хранить все данные из всех потоков она не в состоянии.

## 4.1.2. Примеры источников потоков данных

Прежде чем двигаться дальше, рассмотрим, как потоки возникают на практике.

### Данные с датчиков

Представьте себе датчик температуры, который закреплен на якоре в океане и каждый час посылает основной станции сведения о температуре поверхности. Этот датчик порождает поток вещественных чисел – не особенно интересный, потому что темп передачи данных слишком низок. Для современных технологий это никакая не проблема, весь поток можно было бы хранить в оперативной памяти хоть вечно.

Теперь снабдим датчик GPS-модулем и поручим ему сообщать не температуру, а высоту над уровнем поверхности. По сравнению с температурой высота меняется довольно быстро, поэтому датчик может отправлять данные десять раз в секунду. Если всякий раз отправляется 4-байтовое вещественное число, то за день накопится 3,5 мегабайта. Но и в этом случае для заполнения всей оперативной памяти, не говоря уже о диске, потребуется некоторое время.

Но что такое один датчик? Чтобы получить значимую информацию о поведении океана, датчиков должны быть миллионы, и каждый отправляет поток со скоростью 10 элементов в секунду. Миллион датчиков – это не так уж много; по одному на каждые 150 квадратных миль океана. Теперь ежедневно поступает 3,5 терабайта, и, очевидно, пора задуматься, что хранить в рабочей памяти, а что архивировать.

### Изображения

Спутники ежедневно посылают на землю потоки, содержащие многие терабайты изображений. Камеры слежения снимают изображения более низкого разрешения, чем спутники, но их может быть много, и каждая генерирует поток изображений с интервалом порядка 1 секунды. Говорят, что в Лондоне установлено шесть миллионов таких камер, и все они генерируют потоки данных.

### Интернет-трафик и веб-трафик

Коммутатор, связывающий сегменты Интернета, получает потоки IP-пакетов из многих источников и направляет различным получателям. Обычно работа коммутатора заключается в том, чтобы передавать данные, а не сохранять их и уж тем более не предъявлять к ним запросов. Но набирает силу тенденция расширять возможности коммутатора; например, он может обнаруживать DoS-атаки или изменять маршруты пакетов, основываясь на сведениях о заторах в сети.

Веб-сайты получают потоки разных типов. Например, Google каждый день получает несколько сотен миллионов запросов. Yahoo! ежедневно принимает миллиарды «кликов» на своих сайтах. Из этих потоков можно извлечь много интересной информации. Например, увеличение числа запросов вида «боль в горле» позволяет сделать вывод о распространении вируса. Внезапное увеличение часто-

ты кликов по некоторой ссылке может означать, что на странице была размещена какая-то новость, или о том, что ссылка «битая» и ее необходимо срочно починить.

### 4.1.3. Запросы к потокам

Есть два способа предъявить запрос к потоку. На рис. 4.1 показано место внутри процессора, где хранятся активные запросы. В некотором смысле они выполняются постоянно и что-то выводят в подходящее время.

> **Пример 4.1.** С упомянутым в начале раздела 4.1.2 потоком, который порождает датчик температуры на поверхности океана, может быть связан активный запрос, выводящий уведомление, когда температура превышает 25 градусов по Цельсию. На этот запрос легко ответить, потому что ответ зависит только от последнего элемента потока.
>
> Но может быть и такой активный запрос, который при поступлении каждого нового измерения вычисляет среднее арифметическое последних 24 измерений. Этот запрос тоже нетрудно обработать, если хранить последние 24 измерения. При поступлении нового элемента мы можем отбросить 25-й с конца элемента, поскольку больше он не понадобится (если, конечно, нет другого активного запроса, которому он нужен).
>
> Можно также спросить, какую максимальную температуру фиксировал датчик за все время измерений. Чтобы ответить, мы должны сохранить простой агрегат: максимальное из всех когда-либо виденных измерений. При поступлении нового измерения мы сравниваем его с запомненным максимумом и переустанавливаем максимум, если новое измерение больше. В ответ на запрос выводится текущее значение максимума. Аналогично, если нужна средняя температура за все время измерений, то нужно хранить всего два значения: общее количество измерений, отправленных в поток, и сумму этих измерений. Их можно модифицировать при поступлении каждого нового измерения, а в ответ на запрос посылать частное от деления.

Существуют также эпизодические запросы относительно текущего состояния одного или нескольких потоков. Если не хранить все потоки целиком, чего мы обычно не делаем, то нельзя ожидать, что мы сможем ответить на любой запрос о потоках. Если имеется априорное представление о том, какие запросы могут быть заданы в интерфейсе эпизодических запросов, то можно подготовиться к ним, сохранив достаточные части или некоторые агрегаты потоков, как в примере 4.1.

Если нужно средство для предъявления разнообразных эпизодических запросов, то обычно в рабочей памяти сохраняют скользящее окно каждого потока. В таком окне могут храниться последние $n$ элементов потока или все элементы, поступившие за последние $t$ единиц времени, например 1 день. Если рассматривать каждый элемент потока как кортеж, то окно можно считать отношением и формулировать запросы на языке SQL. Разумеется, система управления потоками должна поддерживать актуальность окна, т. е. удалять старые элементы по мере поступления новых.

**Пример 4.2.** Веб-сайты часто сообщают о количестве уникальных пользователей за прошлый месяц. Если считать каждый вход в систему элементом потока, то можно поддержать окно, содержащее все входы за последний месяц. С каждым входом нужно ассоциировать время, чтобы знать, когда он перестает принадлежать окну. Если рассматривать окно как отношение `Logins(name, time)`, то нетрудно посчитать количество уникальных пользователей за последний месяц. SQL-запрос выглядит так:

```
SELECT COUNT(DISTINCT(name))
FROM Logins
WHERE time >= t;
```

Здесь *t* – константа, представляющая момент времени на месяц раньше текущего.

Отметим, что должна быть возможность хранить весь поток входов за прошлый месяц в рабочей памяти. Но даже для самых крупных сайтов объем таких данных не превышает нескольких терабайтов, так что они вполне поместятся на диске.

## 4.1.4. Проблемы обработки потоков

Прежде чем переходить к обсуждению алгоритмов, рассмотрим, какие ограничения накладыв,аются при работе с потоками. Прежде всего, потоки зачастую доставляют элементы очень быстро. Мы должны обрабатывать элементы в реальном масштабе времени, потом такой возможности уже не будет (разве что обратиться к архивному хранилищу). Поэтому важно, чтобы алгоритм обработки потока исполнялся целиком в оперативной памяти, вообще или почти не обращаясь к внешним устройствам. Кроме того, даже если потоки «медленные», как в случае датчика температуры из раздела 4.1.2, их может быть много. И если для обработки каждого отдельного потока хватило бы совсем немного оперативной памяти, то все потоки вместе могут превысить имеющийся объем.

Поэтому многие задачи, относящиеся к потоковым данным, было бы легко решить при наличии достаточной памяти, но, когда требуется обрабатывать потоки реальной частоты на машине реального размера, задача становится трудной, и для ее решения приходится изобретать новые методы. Вот два общих замечания о потоковых алгоритмах, которые стоит иметь в виду при чтении этой главы:

- часто можно гораздо эффективнее получить приближенное, а не точное решение;
- как и в главе 3, могут оказаться полезны различные методы, связанные с хэшированием. Вообще говоря, эти методы вносят полезную случайность в поведение алгоритма, позволяющую получить приближенный ответ, очень близкий к истинному.

# 4.2. Выборка данных из потока

В качестве первого примера управления потоковыми данными рассмотрим получение надежной выборки из потока. Как во многих потоковых алгоритмах, хитрость заключается в необычном хэшировании.

## 4.2.1. Пояснительный пример

Общая проблема состоит в том, чтобы выбрать из потока подмножество такое, что ответы на предъявленные к нему запросы статистически репрезентативны для потока в целом. В случае, когда предъявляемые запросы известны заранее, существует ряд работоспособных методов, но мы хотели бы разрешить произвольные запросы к выборке. Возьмем конкретную задачу и из ее решения выведем общую идею.

В этом разделе мы будем рассматривать следующий пример. Поисковая система получает поток запросов и хотела бы изучить поведение типичных пользователей[1]. Предположим, что поток состоит из кортежей (пользователь, запрос, время), и наша цель – получить ответы на такие вопросы, как «Какова доля запросов типичных пользователей, повторявшихся в течение последнего месяца?». Предположим еще, что готовы хранить только 1/10 часть элементов потока.

Очевидный подход состоит в том, чтобы генерировать случайное целое число от 0 до 9 при получении каждого поискового запроса. Кортеж сохраняется тогда и только тогда, когда это число равно 0. В этом случае будет сохранена в среднем 1/10 всех запросов каждого пользователя. Из-за статистических флуктуаций данные будут немного зашумлены, но если пользователь отправлял много запросов, то по закону больших чисел доля его сохраненных запросов будет близка к 1/10.

Однако при такой схеме мы получим неверный ответ на вопрос о среднем числе повторяющихся запросов для каждого пользователя. Предположим, что в прошлом месяце пользователь отправил $s$ запросов по одному разу, $d$ запросов по два раза и ни одного запроса более двух раз. Если взять выборку, содержащую 1/10 всех запросов данного пользователя, то мы увидим в ней ожидаемые $s/10$ запросов, отправленных по разу. Из $d$ запросов, отправленных дважды, в выборку попадет только $d/100$; эта величина равна $d$, умноженное на вероятность того, что оба запроса попадут в выборку объемом 1/10. Из запросов, встретившихся в полном потоке дважды, $18d/100$ будут присутствовать ровно один раз. Чтобы понять, почему, заметим, что 18/100 – это вероятность того, что один из двух запросов окажется в выбранной 1/10 части потока, а другой – в невыбранной части, составляющей 9/10.

Правильный ответ на вопрос о доле повторяющихся поисковых запросов – $d/(s+d)$. Однако на основе выборки мы получаем ответ $d/(10s+19d)$. Чтобы вывести эту формулу, заметим, что $d/100$ запросов встречаются в выборке дважды, а

[1] Мы говорим «пользователи», но на самом деле поисковая система получает IP-адреса, с которых были отправлены запросы. Мы будем предполагать, что эти адреса однозначно идентифицируют пользователей, хотя это и не совсем так.

$s/10+18d/100$ – один раз. Следовательно, доля запросов, встретившихся в выборке дважды, равна $d/100$, поделенное на $d/100 + s/10 + 18d/100$, т. е. $d/(10s + 19d)$. Ни при каких положительных значениях $s$ и $d$ не может иметь место равенство $d/(s + d) = d/(10s + 19d)$.

## 4.2.2. Получение репрезентативной выборки

На вопрос из раздела 4.2.1, как и на многие вопросы о статистике типичных пользователей, нельзя ответить, взяв выборку поисковых запросов каждого пользователя. Следовательно, мы должны попытаться отобрать 1/10 часть всех пользователей и включить в выборку все их запросы, игнорируя запросы всех прочих пользователей. Если есть возможность сохранить список всех пользователей вместе с признаком их присутствия в выборке, то мы сможем поступить следующим образом. При поступлении нового запроса смотрим, присутствует ли отправивший его пользователь в выборке. Если да, добавляем запрос в выборку, иначе игнорируем. Если же мы не видели данного пользователя раньше, то генерируем случайное целое число от 0 до 9. Если оно равно 0, добавляем пользователя в список с признаком «входит», иначе – с признаком «не входит».

Этот способ работает, если мы можем позволить себе хранить список всех пользователей с признаком «входит-не входит» в оперативной памяти, потому что на обращение к диску при обработке каждого запроса нет времени. Воспользовавшись хэш-функцией, мы сможем отказаться от хранения списка пользователей. Будем хэшировать каждое имя пользователя в одну из 10 ячеек с номерами от 0 до 9. Если пользователь окажется в ячейке 0, включим запрос в выборку, иначе проигнорируем.

Отметим, что мы не храним пользователя в ячейке, там вообще не хранятся данные. По сути дела, хэш-функция используется как генератор случайных чисел, но обладает важным дополнительным свойством: если применить ее к одному и тому же пользователю несколько раз, то всякий раз будем получаться то же самое «случайное» значение. Таким образом, нигде не сохраняя признак входит-не входит, мы можем реконструировать его при получении каждого запроса от данного пользователя.

Более общо, мы можем получить выборку, содержащую долю запросов пользователей, равную любому рациональному числу $a/b$. Для этого нужно хэшировать имена пользователей в $b$ ячеек с номерами от 0 до $b - 1$ и включать запрос в выборку, если хэш-код меньше $a$.

## 4.2.3. Общая постановка задачи о выборке

Рассмотренный пример иллюстрирует следующую общую задачу. Поток состоит из кортежей с $n$ компонентами. Некоторые компоненты являются ключевыми, на них будет основана выборка. В нашем примере было три компоненты – пользователь, запрос и время, а ключевым был только пользователь. Но можно было бы строить выборку, взяв в качестве ключа запрос или даже пару пользователь-запрос – тогда ключевыми были бы обе компоненты.

Чтобы получить выборку размера $a/b$, мы хэшируем ключи кортежей в $b$ ячеек и включаем кортеж в выборку, если хэш-код меньше $a$. Если ключ состоит из нескольких компонент, то хэш-функция должна каким-то образом объединить их, чтобы получился один хэш-код. Результатом будет выборка, состоящая из всех кортежей с некоторыми значениями ключа. В выборку войдет примерно $a/b$ всех встречающихся в потоке значений ключа.

### 4.2.4. Динамическое изменение размера выборки

Часто выборка растет по мере поступления в систему новых данных. В нашем примере мы всегда сохраняем все поисковые запросы выбранной 1/10 части пользователей. Со временем будет накапливаться все больше запросов от одних и тех же пользователей, а, кроме того, будут появляться новые пользователи.

Если имеются ограничения на количество кортежей, сохраняемых в выборке, то доля отбираемых ключей должна динамически уменьшаться со временем. Чтобы гарантировать, что в любой момент выборка содержит все кортежи с ключами из заданного подмножества, выберем хэш-функцию $h$, отображающую значения ключей на очень большой диапазон целых чисел 0, 1, ..., $B - 1$. Мы храним пороговое значение $t$, первоначально равное номеру последней ячейки $B - 1$. В каждый момент времени выборка состоит из кортежей, для которых ключ $K$ удовлетворяет условию $h(K) \leq t$. Новые кортежи из потока включаются в выборку тогда и только тогда, когда удовлетворяют этому же условию.

Как только количество сохраняемых кортежей в выборке начинает превышать выделенную квоту, мы уменьшаем $t$ до $t - 1$ и удаляем из выборки все кортежи, для которых ключ $K$ хэшируется в $t$. Для пущей эффективности можно уменьшать $t$ сразу на величину, большую 1, и удалять кортежи с несколькими старшими значениями ключей. Дополнительно процесс можно ускорить, если поддерживать индекс по хэш-коду, чтобы можно было быстро находить все кортежи, ключи которых хэшируются в заданное значение.

### 4.2.5. Упражнения к разделу 4.2

**Упражнение 4.2.1.** Рассмотрим поток кортежей с такой схемой:

```
Grades(university, courseID, studentID, grade)
```

Предположим, что все университеты уникальны, а идентификатор учебного курса courseID уникален только в пределах одного университета (т. е. в разных университетах могут быть курсы с одинаковыми идентификаторами, например «CS101») и точно так же идентификатор студента studentID уникален только в пределах одного университета. Допустим, что мы хотим получать ответы на некоторые вопросы по выборке, содержащей примерно 1/20 часть данных. Для каждого из вопросов ниже объясните, как бы вы стали строить выборку, т. е. какие выбрали бы ключевые атрибуты.

(*а*) Для каждого университета оценить среднее число студентов, посещающих курс.
(*б*) Оценить долю студентов со средним баллом аттестата не ниже 3.5.
(*в*) Оценить долю курсов, по которым не менее половины студентов получили оценку «А».

# 4.3. Фильтрация потоков

Еще один процесс, часто применяемый к потокам, – фильтрация. Мы хотим отбирать только кортежи, удовлетворяющие некоторому критерию. Прошедшие проверку кортежи передаются потоком другому процессу, а не прошедшие отбрасываются. Если критерием является вычисляемое свойство кортежа (например, первая компонента меньше 10), то произвести фильтрацию легко. Задача усложняется, когда критерий подразумевает поиск в некотором множестве, особенно если множество не помещается в оперативной памяти. В этом разделе мы обсудим метод «фильтра Блума», позволяющий исключить большинство кортежей, не отвечающих критерию.

## 4.3.1. Пояснительный пример

Снова возьмем пример, иллюстрирующий проблему, и посмотрим, что в нем можно сделать. Пусть имеется множество $S$, содержащее миллиард разрешенных почтовых адресов, с которых мы готовы принимать почту, потому что считаем, что они не рассылают спам. Поток состоит из пар: адрес и само сообщение. Поскольку длина типичного сообщения больше 20 байтов, неразумно хранить $S$ в оперативной памяти. Следовательно, принимая решение о том, пропустить элемент потока или нет, мы должны либо обращаться к диску, либо придумать метод, который не будет требовать больше памяти, чем есть в наличии, но при этом сможет отфильтровать большую часть нежелательных элементов.

Предположим для определенности, что имеется оперативная память объемом 1 ГБ. В фильтре Блума оперативная память трактуется как битовый массив. В данном случае у нас есть место для восьми миллиардов битов, т. к. один байт состоит из восьми битов. Требуется придумать хэш-функцию $h$, которая отображает почтовые адреса на восемь миллиардов ячеек. Каждый элемент множества $S$ будет хэшироваться в номер бита, и этот бит будет установлен в 1. Остальные биты будут равны 0.

Поскольку в $S$ миллиард элементов, примерно 1/8 часть всех битов будет установлена в 1. В действительности единичных битов будет чуть меньше, потому что два разных элемента $S$ могут хэшироваться в один бит. Какова точная доля единиц, мы обсудим в разделе 4.3.3. Для каждого поступившего элемента потока мы хэшируем его почтовый адрес. Если соответствующий бит равен 1, то сообщение пропускается. Если же он равен 0, то мы точно знаем, что адрес не принадлежит $S$, и потому отбрасываем сообщение.

К сожалению, некоторые спамные сообщения все же просочатся. Приблизительно 1/8 часть элементов потока с адресами, не принадлежащими $S$, хэшируется в единичные биты и, значит, пропускается фильтром. Тем не менее, поскольку большинство почтовых сообщений – это спам (примерно 80 % согласно некоторым отчетам), исключение 7/8 спама – большое достижение. Если же мы хотим исключить весь спам, то должны будет проверять лишь принадлежность к $S$ тех хороших и плохих адресов, которые прошли через фильтр. Для выполнения таких проверок придется обратиться к самому множеству $S$ во внешней памяти. Но существуют и другие возможности, о которых мы узнаем в ходе обсуждения общего метода фильтрации Блума. Например, можно было бы использовать каскад фильтров, каждый из которых отсеивает 7/8 оставшегося спама.

## 4.3.2. Фильтр Блума

Фильтр Блума состоит из следующих частей:

1. Массив $n$ бит, первоначально равных 0.
2. Набор хэш-функций $h_1$, $h_2$, ..., $h_k$. Каждая функция отображает значения «ключей» в $n$ ячеек, соответствующих $n$ битам массива.
3. Множество $S$, содержащее $m$ ключей.

Назначение фильтра Блума – пропускать все элементы потока, ключи которых принадлежат $S$, и отбрасывать большинство элементов с ключами, не принадлежащими $S$.

Инициализируем битовый массив, обнулив все биты. Применим к каждому ключу в $S$ каждую из $k$ хэш-функций. Установим в 1 биты, номера которых совпадают с $h_i(K)$ для некоторой хэш-функции $h_i$ и некоторого ключа $K$ из $S$.

При обработке ключа $K$, поступившего из потока, проверяем, что значения всех битов с номерами

$$h_1(K), h_2(K), ..., h_k(K)$$

равны 1. Если это так, пропускаем элемент. Если хотя бы один бит равен 0, то ключ $K$ не может принадлежать $S$, поэтому отбрасываем элемент.

## 4.3.3. Анализ фильтра Блума

Если значение ключа принадлежит $S$, то элемент, безусловно, будет пропущен фильтром Блума. Но элемент может пройти и тогда, когда его ключ отсутствует в $S$. Нам нужно понять, как выразить вероятность ложноположительного результата в виде функции от длины битового массива $n$, количества $m$ элементов множества $S$ и количества хэш-функций $k$.

В качестве модели воспользуемся бросанием дротиков в мишени. Пусть имеется $x$ мишеней и $y$ дротиков. Каждый дротик с равной вероятностью поражает любую мишень. Каково математическое ожидание количества мишеней, пораженных хотя бы один раз, после бросания всех дротиков? Анализ проводится так же, как в разделе 3.4.2.

- Вероятность, что данный дротик не поразит данную мишень, равна $(x - 1)/x$.
- Вероятность, что ни один из $y$ дротиков не поразит данную мишень, равна $((1 - x)/x)^y$. Это выражение можно переписать в виде $(1 - 1/x)^{x(y/x)}$.
- Воспользовавшись аппроксимацией $(1 - \varepsilon)^{1/\varepsilon} = 1/e$ для малых $\varepsilon$ (см. раздел 1.3.5), заключаем, что вероятность того, что ни один из $y$ дротиков не поразит заданную мишень, равна $e^{-y/x}$.

**Пример 4.3.** Рассмотрим пример из раздела 4.3.1. Приведенное выше вычисление можно использовать для получения истинного математического ожидания количества единиц в битовом массиве. Будем считать каждый бит мишенью, а каждый элемент $S$ дротиком. Тогда вероятность того, что заданный бит будет равен 1, – не что иное как вероятность того, что соответствующая мишень будет поражена хотя бы одним дротиком. В $S$ миллиард элементов, значит, у нас есть $y = 10^9$ дротиков. Битов 8 миллиардов, следовательно, имеется $8 \times 10^9$ мишеней. Таким образом, вероятность непоражения заданной мишени равна $e^{-y/x} = e^{-1/8}$, а вероятность ее поражения равна $1 - e^{-1/8}$, т. е. примерно 0.1175. В разделе 4.3.1 мы предположили, что 1/8 = 0.125 – хорошее приближение, и это действительно так, но теперь мы имеем и точный результат.

Это правило можно применить и к более общей ситуации, когда множество $S$ содержит $m$ элементов, массив состоит из $n$ битов и имеется $k$ хэш-функций. Количество мишеней равно $x = n$, а количество дротиков – $y = km$. Следовательно, вероятность того, что некий бит останется равным 0, равна $e^{-km/n}$. Мы хотим, чтобы доля нулей была достаточно большой, иначе вероятность того, что элемент, не принадлежащий $S$, хотя бы один раз будет хэширован в 0, окажется слишком маленькой, а, значит, будет слишком много ложноположительных результатов. Например, можно выбрать количество хэш-функций $k$ равным $n/m$ или меньше. Тогда вероятность получить 0 не меньше $e^{-1}$, или 37 %. В общем случае вероятность ложноположительного результата, т. е. вероятность получить единичный бит, равна $(1 - e^{-km/n})^k$.

**Пример 4.4.** В примере 4.3 мы нашли, что доля единиц в нашем массиве, т. е. вероятность ложноположительного результата, составляет 0.1175, Это означает, что элемент, не принадлежащий $S$, будет пропущен фильтром, если он хэшируется в 1, что происходит с вероятностью 0.1175.

Допустим, что используется то же множество $S$ и тот же массив, но две разных хэш-функции. Эта ситуация соответствует бросанию двух миллиардов дротиков в восемь миллиардов мишеней, и вероятность, что заданный бит останется нулевым, равна $e^{-1/4}$. Чтобы получить ложноположительный результат, элемент, не принадлежащий $S$, должен быть дважды хэширован в 1, вероятность чего составляет $(1 - e^{-1/4})^2$, или приблизительно 0.0493. Следовательно, добавление второй хэш-функции в нашем примере является улучшением, благодаря которому частота ложноположительных результатов сокращается с 0.1175 до 0.0493.

### 4.3.4. Упражнения к разделу 4.3

**Упражнение 4.3.1.** В описанной ситуации (8 миллиардов битов, 1 миллиард элементов в множестве $S$) вычислите частоту ложноположительных результатов при использовании трех хэш-функций. А если их четыре?

! **Упражнение 4.3.2.** Пусть объем доступной памяти составляет $n$ битов, а множество содержит $m$ элементов. Вместо того чтобы использовать $k$ хэш-функций, мы могли бы распределить $n$ битов по $k$ массивам и произвести хэширование по одному разу в каждый массив. Выразите вероятность ложноположительного результата в виде функции от $n$, $m$, $k$. Как это соотносится с использованием $k$ функций, хэширующих в один массив?

!! **Упражнение 4.3.3.** Выразите в виде функции от $n$ – числа битов и $m$ – числа элементов множества $S$ количество хэш-функций, при котором достигается минимум частоты ложноположительных результатов.

## 4.4. Подсчет различных элементов в потоке

В этом разделе мы рассмотрим третий вид обработки потока. Как и в предыдущих случаях – выборке и фильтрации – для получения желаемого результата в условиях ограничений на объем оперативной памяти придется исхитриться. Мы будем использовать вариант хэширования и рандомизированный алгоритм, чтобы получить приближение к истинному значению, потребляя не слишком много памяти на каждый поток.

### 4.4.1. Проблема Count-Distinct

Пусть элементы потока выбираются из какого-то универсального множества. Нам хочется знать, сколько различных элементов встречалось в потоке, считая либо от начала потока, либо от заданного момента в прошлом.

> **Пример 4.5.** В качестве полезного примера этой задачи рассмотрим сайт, который собирает статистику о количестве уникальных пользователей в каждом месяце. В данном случае универсальное множество – это множество заходов на сайт, а элемент потока генерируется при каждом входе в систему. Такой показатель подходит для сайтов типа Amazon, где типичный пользователь заходит со своим уникальным логином.
>
> Похожая задача возникает на сайтах типа Google, которые не требуют вводить логин для отправки поискового запроса и умеют идентифицировать пользователей только по IP-адресу. Существует около 4 миллиардов IP-адресов[2], универсальным множеством являются последовательности из четырех 8-битовых байтов.

[2] По крайней мере, до тех пор пока IPv6 не станет нормой.

Очевидный подход к решению этой задачи – хранить в оперативной памяти список всех уже встречавшихся элементов потока. Причем хранить в структуре данных, обеспечивающей эффективный поиск, например, в хэш-таблице или в дереве поиска, так чтобы можно было быстро добавлять новые элементы и проверять, встречался ли ранее только что поступивший из потока элемент. Пока количество различных элементов не слишком велико, эта структура помещается в оперативной памяти, и мы без всяких проблем можем узнать, сколько различных элементов встречалось в потоке.

Но если различных элементов становится слишком много или приходится обрабатывать сразу много потоков (например, Yahoo! хочет знать, сколько уникальных пользователей заходило на каждую страницу в каждом месяце), то мы не сможем сохранить все необходимые данные в оперативной памяти. Вариантов решения несколько. Можно использовать несколько машин, каждая из которых будет отвечать за один или несколько потоков. Можно хранить большую часть данных во внешней памяти и собирать элементы потока в пакет, а при перемещении блока с диска в оперативную память производить много проверок и обновлений данных в этом блоке. Или можно использовать обсуждаемую в этом разделе стратегию, когда мы получаем лишь оценку количества различных элементов, зато потребляем памяти гораздо меньше, чем было бы необходимо для их хранения.

## 4.4.2. Алгоритм Флажоле-Мартена

Количество различных элементов можно оценить путем хэширования элементов универсального множества в достаточно длинную битовую строку. Длина должна быть такой, чтобы возможных значений хэш-функции было больше, чем элементов в универсальном множестве. Например, 64 битов достаточно для хэширования URL-адресов. Мы выберем много разных хэш-функций и будем с их помощью хэшировать каждый элемент потока. У хэш-функции есть важное свойство – если несколько раз применить ее к одному элементу, то всякий раз мы будем получать один и тот же результат. Отметим, что это свойство было важно и в методе получения выборки из раздела 4.2.

Идея алгоритма Флажоле-Мартена заключается в том, что чем больше мы увидим элементов потока, тем больше будет получено различных хэш-кодов. А чем больше различных хэш-кодов, тем выше вероятность, что один из них будет «необычен». Конкретное интересующее нас необычное свойство – большое число нулей в конце, хотя существуют и другие варианты.

Если применить хэш-функцию $h$ к элементу потока $a$, то в конце битовой строки $h(a)$ будет какое-то количество нулей, быть может, ни одного. Назовем его *хвостовой длиной* для $a$ и $h$. Пусть $R$ – максимальная хвостовая длина для любого элемента $a$, встречавшегося до сих пор в потоке. Тогда в качестве оценки числа различных виденных элементов возьмем $2^R$.

У этой оценки интуитивно понятный смысл. Вероятность того, что значение $h(a)$ для данного элемента $a$ оканчивается по крайней мере $r$ нулями, равна $2^{-r}$. Предположим, что в потоке $m$ различных элементов. Тогда вероятность, что ни для

одного их них хвостовая длина не достигает $r$, равна $(1 - 2^{-r})^m$. Выражения такого вида нам уже знакомы. Мы можем переписать его в виде $((1 - 2^{-r})^{2^r})^{m2^{-r}}$. В предположении, что $r$ достаточно велико, внутреннее выражение имеет вид $(1 - \varepsilon)^{1/\varepsilon}$, что приближенно равно $1/e$. Таким образом, вероятность не найти элемента потока, для которого хэш-код заканчивается $r$ нулями, равна $e^{-m2^{-r}}$. Мы можем сделать следующий вывод:

1. Если $m$ много больше $2^r$, то вероятность найти хвост длиной не менее $r$ близка к 1.
2. Если $m$ много меньше $2^r$, то вероятность найти хвост длиной не менее $r$ близка к 0.

Отсюда следует, что предложенная оценка $m - 2^R$ (напомним, что $R$ максимальная хвостовая длина любого элемента потока) – с большой вероятностью не является ни сильно завышенной, ни сильно заниженной.

## 4.4.3. Комбинирование оценок

К сожалению, имеется подвох, относящийся к стратегии комбинирования оценок количества различных элементов $m$, получаемых применением многих разных хэш-функций. В качестве первой попытки можно предположить, что если взять среднее значений $2^R$, получаемых от каждой функции, то результат будет тем ближе к истинному значению $m$, чем больше хэш-функций. Однако это не так, и причина связана с влиянием завышения оценки на среднее.

Рассмотрим значение $r$ такое, что $2^r$ много больше $m$. Пусть $p$ – вероятность того, что $r$ – наибольшее количество нулей в конце хэш-кода любого из $m$ элементов потока. Тогда вероятность обнаружить, что наибольшее количество нулей равно $r + 1$, не меньше $p/2$. Однако если увеличить на 1 количество нулей в конце хэш-кода, то значение $2^R$ удвоится. Следовательно, вклад каждого возможного большого $R$ в ожидаемое значение $2^R$ растет с ростом $R$, и ожидаемое значение $2^R$ фактически бесконечно[3].

Можно комбинировать оценки и по-другому – взять их медиану. На медиану не влияют случайные выбросы $2^R$, поэтому высказанные выше опасения для среднего на медиану не распространяются. К сожалению, медиане свойствен другой дефект: она всегда является степенью 2. Следовательно, сколько бы хэш-функций ни взять, если истинное значение $m$ лежит между двумя степенями 2, скажем равно 400, получить близкую к нему оценку невозможно.

Но у задачи все же есть решение. Можно применить комбинацию обоих методов. Сначала объединим хэш-функции в небольшие группы и вычислим их средние. Затем возьмем медиану средних. Да, случайные выбросы $2^R$ приводят к асимметрии некоторых групп, в результате чего их среднее оказывается слишком большим. Но, взяв медиану групповых средних, мы сводим влияние этого эффекта почти к нулю. Кроме того, если сами группы достаточно велики, то среднее может быть

[3] Строго говоря, поскольку хэш-код – битовая строка конечной длины, те $R$, которые больше длины хэш-кода, не дают вклада в $2^R$. Однако этого недостаточно, чтобы обесценить вывод о том, что ожидаемое значение $2^R$ слишком велико.

практически любым числом, что позволяет приблизиться к истинному значению $m$, если хэш-функций достаточно много. Для гарантированного получения любого возможного среднего размер групп должен быть не меньше небольшого кратного $\log_2 m$.

### 4.4.4. Требования к памяти

Отметим, что при чтении потока необязательно сохранять встретившиеся элементы. В оперативной памяти нужно хранить только одно целое число на каждую хэш-функцию – текущую максимальную хвостовую длину элементов потока для данной функции. Если обрабатывается всего один поток, то можно использовать миллионы хэш-функций, а это гораздо больше, чем необходимо для получения близкой оценки. И лишь если одновременно обрабатывается много потоков, то объем оперативной памяти начинает ограничивать количество функций, ассоциируемых с каждым. Впрочем, на практике количество хэш-функций ограничено, скорее, временем вычисления хэш-кодов элементов потока.

### 4.4.5. Упражнения к разделу 4.4

**Упражнение 4.4.1.** Предположим, что поток состоит из чисел 3, 1, 4, 1, 5, 9, 2, 6, 5. Все хэш-функции будут иметь вид $h(x) = ax + b \bmod 32$ для некоторых $a$ и $b$. Результат следует рассматривать как 5-разрядное двоичное целое. Определите хвостовую длину каждого элемента потока и итоговую оценку количества различных элементов для следующих хэш-функций:

(*а*) $h(x) = 2x + 1 \bmod 32$.
(*б*) $h(x) = 3x + 7 \bmod 32$.
(*в*) $h(x) = 4x \bmod 32$.

**! Упражнение 4.4.2.** Видите ли вы какую-нибудь проблему в выборе хэш-функций в упражнении 4.4.1? Какой совет вы дали бы человеку, собирающемуся использовать хэш-функцию вида $h(x) = ax + b \bmod 2^k$?

## 4.5. Оценивание моментов

В этом разделе мы обсудим обобщение задачи подсчета различных элементов в потоке. Проблема вычисления «моментов» относится к распределению частоты различных элементов в потоке. Мы определим моменты всех порядков, но сосредоточимся на вычислении вторых моментов, а алгоритм для моментов любого порядка можно будет получить простым обобщением.

### 4.5.1. Определение моментов

Предположим, что поток содержит элементы, выбранные из универсального множества. Предположим еще, что универсальное множество упорядочено, т. е. можно говорить об $i$-ом элементе для любого $i$. Пусть $m_i$ – количество вхождений $i$-го

элемента. Тогда моментом $k$-ого порядка (или просто $k$-ым моментом) потока называется сумма $(m_i)^k$ по всем $i$.

> **Пример 4.6.** Нулевой момент – это сумма единиц для всех $m_i$, больших 0.[4] Следовательно, нулевой момент – это количество различных элементов в потоке. Для оценивания нулевого момента потока можно воспользоваться методом из раздела 4.4.
>
> Первый момент – это сумма всех $m_i$, т. е. длина потока. Следовательно, первый момент вычисляется особенно легко, нужно просто подсчитать количество всех виденных элементов. Иногда его называют *мерой неожиданности* (surprise number), потому что он измеряет неравномерность распределения элементов в потоке. Предположим, что имеется поток длины 100, в котором встречаются одиннадцать различных элементов. При максимально равномерном распределении один элемент встретился бы 10 раз, а все остальные – по 9. При этом мера неожиданности будет равна $10^2 + 10 \times 9^2 = 910$. Другой крайний случай возникает, когда один элемент встречается 90 раз, а остальные десять – по разу. Тогда мера неожиданности равна $90^2 + 10 \times 1^2 = 8110$.

Как и в разделе 4.4, вычисление моментов любого порядка не представляет проблемы, если можно хранить в оперативной памяти счетчики вхождений всех элементов. Но если столько памяти нет, то необходимо оценивать $k$-ый момент на основе ограниченного количества значений, хранящихся в памяти. При подсчете различных элементов мы хранили в качестве таких значений максимальные хвостовые длины, порождаемые хэш-функциями. Ниже будет рассмотрен другой вид значений, полезный для вычисления моментов второго и более высоких порядков.

## 4.5.2. Алгоритм Алона-Матиаса-Сегеди для вторых моментов

Предположим пока, что длина потока фиксирована и равна $n$. Как обрабатывать растущие потоки, мы покажем в следующем разделе. Предположим, что для хранения $m_i$ для всех элементов потока не хватает памяти. Тем не менее, мы можем оценить второй момент, и чем больше памяти имеется, тем точнее будет оценка. Мы вычисляем некоторое число *переменных*. Для каждой переменной $X$ мы храним:

1. Определенный элемент универсального множества, который будем обозначать *X.element*.
2. Целое число *X.value*, являющееся значением переменной. Чтобы определить значение переменной $X$, мы выбираем в потоке позицию с номером от 1 до $n$, случайно и с равномерным распределением. Положим *X.element* равным находящемуся в этой позиции элементу и инициализируем *X.value*

[4] Строго говоря, поскольку $m_i$ может быть равно 0 для некоторых элементов универсального множества, необходимо явно указать в определении момента, что $0^0$ считается равным 0. Для моментов порядка 1 и выше вклад $m_i$, равных 0, очевидно нулевой.

значением 1. По мере чтения потока будем прибавлять 1 к *X.value* всякий раз, как встретится *X.element*.

**Пример 4.7:** Пусть поток состоит из элементов *a*, *b*, *c*, *b*, *d*, *a*, *c*, *d*, *a*, *b*, *d*, *c*, *a*, *a*, *b*. Длина потока равна $n = 15$. Поскольку *a* встречается 5 раз, *b* – 4 раза, а *c* и *d* – по 3 раза, то второй момент этого потока равен $5^2+4^2+3^2+3^2 = 59$. Допустим, что мы храним три переменные $X_1$, $X_2$, $X_3$, и пусть для определения этих переменных «случайно» выбраны позиции 3, 8 и 13.

Дойдя до позиции 3, мы прочитаем элемент *c*, поэтому устанавливаем $X_1.element = c$ и $X_1.value = 1$. В позиции 4 находится *b*, поэтому $X_1$ не изменяется. При чтении позиций 5 и 6 тоже ничего не происходит. В позиции 7 мы снова находим *c*, поэтому переустанавливаем $X_1.value = 2$.

В позиции 8 находится *d*, так что $X_2.element = d$ и $X_2.value = 1$. В позициях 9 и 10 находятся *a* и *b*, поэтому ни на $X_1$, ни на $X_2$ они не влияют. В позиции 11 находится *d*, так что переустанавливаем $X_2.value = 2$, а в позиции 12 находится *c*, так что $X_1.value = 3$. В позиции 13 мы видим элемент *a*, поэтому $X_3.element = a$ и $X_3.value = 1$. Затем в позиции 14 мы снова встречаем *a*, поэтому $X_3.value = 2$. Находящийся в позиции 15 элемент *b* не влияет ни на одну из переменных, так что в конце мы имеем $X_1.value = 3$, $X_2.value = X_3.value = 2$.

Вывести оценку второго момента можно по любой переменной *X*. Она равна $n(2X.value - 1)$.

**Пример 4.8.** Рассмотрим три переменные из примера 4.7. На основе $X_1$ выводим оценку $n(2X_1.value - 1) = 15 \times (2 \times 3 - 1) = 75$. Две другие переменные, $X_2$ и $X_3$, в конце имели значение 2, поэтому из них получаются оценки $15 \times (2 \times 2 - 1) = 45$. Напомним, что истинное значение второго момента этого потока равно 59. С другой стороны, среднее всех трех оценок равно 55, т. е. аппроксимация довольно близкая.

## 4.5.3. Почему работает алгоритм Алона-Матиаса-Сегеди

Можно доказать, что математическое ожидание любой переменной, построенной в разделе 4.5.2, равно второму моменту соответствующего потока. Для удобства рассуждений введем некоторые обозначения. Пусть $e(i)$ – элемент потока в позиции $i$, а $c(i)$ – сколько раз этот элемент встречается в позициях $i, i + 1, \ldots, n$.

**Пример 4.9.** В потоке из примера 4.7 $e(6) = a$, т. к. шестой элемент равен *a*. Кроме того, $c(6) = 4$, потому что, помимо позиции 4, *a* встречается еще в позициях 9, 13 и 14. Тот факт, что *a* встречается также в позиции 1, не оказывает влияния на $c(6)$.

Математическое ожидание величины $n(2X.value - 1)$ равно среднему значению $n(2c(i) - 1)$ по всем позициям $i$ от 1 до $n$, т. е.

$$E\left(n(2X.value-1)\right)=\frac{1}{n}\sum_{i=1}^{n}n(2c(i)-1)\cdot$$

Это выражение можно упростить, сократив $1/n$ и $n$, и в результате получим:

$$E\left(n(2X.value-1)\right)=\sum_{i=1}^{n}n(2c(i)-1)\cdot$$

Но чтобы прояснить смысл этой формулы, мы изменим порядок суммирования, сгруппировав позиции, в которых находятся одинаковые элементы. Например, возьмем элемент $a$, встречающийся в потоке $m_a$ раз. Член суммы, соответствующий последней позиции, в которой встречается $a$, должен быть равен $2 \times 1 - 1 = 1$. Член, соответствующий предпоследней позиции, равен $2 \times 2 - 1 = 3$. Предшествующие позиции, содержащие $a$, дают слагаемые 5, 7 и т. д. вплоть до $2m_a - 1$, т. е. слагаемого, соответствующего первой позиции, где встречается $a$. Таким образом, выражение для математического ожидания $2X.value - 1$ можно переписать в виде:

$$E(n(2X.value - 1)) = \sum_a 1 + 3 + 5 + ... + (2m_a - 1).$$

Но $1 + 3 + 5 + ... + (2m_a - 1) = (m_a)^2$. Доказательство легко проводится индукцией по числу слагаемых. Следовательно, $E(n(2X.value - 1)) = \sum_a (m_a)^2$, а это и есть определение второго момента.

## 4.5.4. Моменты высших порядков

Для $k > 2$ $k$-й момент оценивается по существу так же, как второй. Единственное различие заключается в том, как из переменной выводится оценка. В разделе 4.5.2 для преобразования количества $v$ вхождений элемента потока $a$ в оценку второго момента применялась формула $n(2v - 1)$. В разделе 4.5.3 мы объяснили, почему эта формула работает: сумма чисел $2v - 1$ для $v = 1, 2, ..., m$ равна $m^2$, где $m$ – количество вхождений $a$ в поток.

Заметим, что $2v - 1$ – это разность между $v^2$ и $(v - 1)^2$. Пусть требуется вычислить не второй, а третий момент. Тогда нужно всего лишь заменить $2v - 1$ на $v^3 - (v - 1)^3 = 3v^2 - 3v + 1$. Поскольку $\sum_{v=1}^{m} 3v^2 - 3v + 1 = m^3$, мы можем взять в качестве оценки третьего момента величину $n(3v^2 - 3v + 1)$, где $v = X.value$ – значение, ассоциированное с некоторой переменной $X$. Вообще, $k$-й момент для любого $k \geq 2$ получается преобразованием $v = X.value$ в $n(v^k - (v - 1)^k)$.

## 4.5.5. Обработка бесконечных потоков

Строго говоря, при получении оценок второго и последующих моментов мы предполагали, что длина потока $n$ – константа. На практике $n$ со временем увеличивается. Сам по себе этот факт не вызывает проблем, потому что мы храним только значения переменных и умножаем некоторую функцию от этого значения на $n$, когда приходит время оценить момент. Если подсчитывать, сколько элементов потока мы видели, и хранить это значение, для чего требуется всего $\log n$ битов, то $n$ будет доступно, когда бы ни потребовалось.

Более серьезная проблема – как выбирать позиции переменных. Если зафиксировать выбор раз и навсегда, то с ростом потока мы будем отдавать предпочтение ранним позициям, и оценка момента окажется завышенной. С другой стороны, если откладывать выбор позиций на слишком долгий срок, то в начале потока переменных будет недостаточно, и оценка окажется ненадежной.

Правильная тактика – всегда хранить столько переменных, сколько позволяет память, и отбрасывать некоторые по мере роста потока. Отброшенные переменные заменяются новыми таким образом, что в каждый момент времени вероятность выбора любой конкретной позиции для переменной равна вероятности выбора любой другой позиции. Предположим, что памяти достаточно для хранения $s$ переменных. Тогда позиции для $s$ переменных случайно выбираются из первых $s$ позиций потока.

По индукции, предположим, что мы уже видели $n$ элементов потока, и вероятность, что какая-то позиция выбрана в качестве позиции переменной, подчиняется равномерному распределению, т. е. равна $s/n$. При поступлении $(n+1)$-ого элемента выбираем эту позицию с вероятностью $s/(n+1)$. Если она не выбрана, то все $s$ переменных сохраняют прежние позиции. Если же $(n+1)$-ая позиция выбрана, то отбрасываем одну из имеющихся $s$ переменных, выбирая ее случайно. Заменяем отброшенную переменную новой, элемент которой находится в позиции $n$, значение равно 1.

Естественно, вероятность того, что позиция $n + 1$ выбрана для переменной, равна, как и положено, $s/(n + 1)$. Однако вероятность любой другой позиции тоже равна $s/(n + 1)$, и мы докажем это индукцией по $n$. По предположению индукции, перед поступлением $(n + 1)$-ого элемента потока эта вероятность была равна $s/n$. С вероятностью $1 - s/(n + 1)$ позиция $n + 1$ не будет выбрана, и вероятность каждой из первых $n$ позиций останется равна $s/n$. Но с вероятностью $s/(n + 1)$ позиция $n + 1$ все же будет выбрана, и тогда вероятность каждой из первых $n$ позиций уменьшится в $(s - 1)/s$ раз. С учетом этих двух случаев вероятность выбора каждой из первых $n$ позиций равна

$$\left(1-\frac{s}{n+1}\right)\left(\frac{s}{n}\right)+\left(\frac{s}{n+1}\right)\left(\frac{s-1}{s}\right)\left(\frac{s}{n}\right).$$

Это выражение можно упростить:

$$\left(1-\frac{s}{n+1}\right)\left(\frac{s}{n}\right)+\left(\frac{s-1}{n+1}\right)\left(\frac{s}{n}\right)$$

затем

$$\left(\left(1-\frac{s}{n+1}\right)+\left(\frac{s-1}{n+1}\right)\right)\left(\frac{s}{n}\right)$$

и наконец

$$\left(\frac{n}{n+1}\right)\left(\frac{s}{n}\right)=\frac{s}{n+1}.$$

Таким образом, индукцией по длине потока $n$ мы доказали, что для любой позиции вероятность ее выбора в качестве позиции переменной равна $s/n$.

**Общая задача о выборке из потока**

Отметим, что описанная в разделе 4.5.5 методика решает более общую задачу. Она позволяет в любой момент времени хранить выборку, включающую $s$ элементов потока, так что вероятность включения в нее любого элемента одинакова.

В качестве полезного примера применения этой техники напомним, что в разделе 4.2 нам нужно было отобрать из потока все кортежи, для которых значение ключа принадлежит случайно выбранному подмножеству. Предположим, что со временем с одним ключом оказывается ассоциировано слишком много кортежей. Мы сможем ограничить количество кортежей для любого ключа $K$ фиксированной константой $s$, если воспользуемся описанной техникой при поступлении каждого нового кортежа для ключа $K$.

## 4.5.6. Упражнения к разделу 4.5

**Упражнение 4.5.1.** Вычислите меру неожиданности (второй момент) для потока 3, 1, 4, 1, 3, 4, 2, 1, 2. Чему равен третий момент этого потока?

**! Упражнение 4.5.2.** Пусть в потоке $n$ элементов и $m$ из них различны. Выразите минимальное и максимальное значение меры неожиданности в виде функции от $m$ и $n$.

**Упражнение 4.5.3.** Пусть к потоку из упражнения 4.5.1 применяется алгоритм Алона-Матиаса-Сегеди для оценки меры неожиданности. Для каждого возможного значения $i$ пусть $X_i$ – переменная с начальной позицией $i$. Каково тогда значение $X_i.value$?

**Упражнение 4.5.4.** Повторите упражнение 4.5.3, если переменные предназначены для вычисления третьего момента. Каким будет значение каждой переменной в конце? Какова оценка третьего момента, полученная по каждой переменной? Насколько среднее значение этих оценок близко к истинному значению третьего момента?

**Упражнение 4.5.5.** Индукцией по $m$ докажите, что $1 + 3 + 5 + ... + (2m - 1) = m^2$.

**Упражнение 4.5.6.** Как следовало бы преобразовать $X.value$ для получения оценки четвертого момента?

# 4.6. Подсчет единиц в окне

Теперь обратимся к задачам о подсчете для потоков. Пусть имеется окно длины $N$, перемещающееся по двоичному потоку. Мы хотим, чтобы в любой момент времени можно было ответить на вопросы вида «Сколько единиц среди последних $k$ битов?» для любого $k \leq N$. Как и в предыдущих разделах, нас будет интересовать ситуация, когда для хранения всего окна недостаточно памяти. Продемонстрировав алгоритм для двоичного случая, мы затем обсудим, как обобщить его идею на вычисление суммы чисел.

## 4.6.1. Стоимость точного подсчета

Для начала предположим, что мы хотим иметь возможность точно подсчитать количество единиц в последних $k$ битах для любого $k \leq N$. Тогда мы утверждаем, что обязательно хранить все $N$ битов окна, поскольку любое представление с меньшим числом битов может оказаться непригодным. Для доказательства предположим, что мы нашли какой-то способ использовать меньше $N$ битов для представления $N$ битов окна. Поскольку существует $2^N$ последовательностей $N$ битов, но меньше $2^N$ представлений, обязательно найдутся две разных битовых строки $w$ и $x$ с одинаковым представлением. Поскольку $w \neq x$, они должны различаться хотя бы в одном бите. Пусть последние $k - 1$ битов $w$ и $x$ совпадают, а в $k$-ом бите, считая справа, последовательности различаются.

> **Пример 4.10.** Если $w = 0101$ и $x = 1010$, то $k = 1$, поскольку при просмотре справа налево последовательности различаются уже в позиции 1. Если $w = 1001$ и $x = 0101$, то $k = 3$, поскольку последовательности различаются в третьей позиции справа.

Допустим, что данные, представляющие содержимое окна, – это та последовательность, которая является общим представлением $w$ и $x$. Зададим вопрос «сколько единиц в последних $k$ битах?». Алгоритм ответа на этот вопрос должен давать один и тот же ответ, вне зависимости от того, содержит окно $w$ или $x$, потому что он видит только представления. Но очевидно, что правильные ответы для этих двух битовых строк различаются. Таким образом, мы доказали, что для ответа на вопросы о последних $k$ битах для любого $k$ необходимо использовать по меньшей мере $N$ битов.

На самом деле, $N$ битов нужно, даже если разрешено задавать только вопрос «сколько единиц во всем окне длины $N$?». Доказательство почти такое же. Допустим, что для представления окна используется меньше $N$ битов и потому можно найти такие $w$, $x$ и $k$, как описано выше. Может случиться, что количество единиц в $w$ и $x$ одинаково, как в обоих случаях из примера 4.10. Но если продолжить текущее окно любыми $N - k$ битами, то мы будем иметь ситуацию, когда истинное содержимое окон, полученных из последовательностей $w$ и $x$, совпадает всюду, кроме самого левого бита, и, значит, количество единиц не равно. Но поскольку

представления $w$ и $x$ совпадают, то будут совпадать и представления окон для последовательностей, полученных дописыванием в конец $w$ и $x$ одной и той же последовательности битов. Таким образом, мы можем сделать так, что ответ на вопрос «сколько единиц в окне?» будет неверным для содержимого одного из двух окон.

## 4.6.2. Алгоритм Датара-Гиониса-Индыка-Мотвани

Мы представим простейший случай алгоритма DGIM. В нем используется $O(\log^2 N)$ битов для представления $N$-битового окна, а погрешность оценки количества единиц в окне не превышает 50 %. Впоследствии мы обсудим усовершенствование этого метода, позволяющее ограничить погрешность произвольным числом $\varepsilon > 0$ при тех же $O(\log^2 N)$ битах (хотя постоянный множитель растет с уменьшением $\varepsilon$).

Прежде всего, отметим, что у каждого бита потока есть временная метка – его позиция. Временная метка первого бита равна 1, второго – 2 и т. д. Поскольку нам нужно различать позиции только в пределах окна длины $N$, мы будем представлять временные метки по модулю $N$, так что для представления хватит $\log_2 N$ битов. Если хранить также общее количество битов в потоке (т. е. самую последнюю временную метку) по модулю $N$, то, зная временную метку по модулю $N$, мы сможем определить, где в текущем окне находится бит с такой временной меткой.

Разобьем окно на *интервалы*, состоящие из:

1. Временной метки правого (более позднего) конца.
2. Количества единиц в интервале. Это число, которое должно быть степенью 2, будем называть *размером* интервала.

Для представления интервала нам нужно $\log_2 N$ битов, чтобы представить временную метку (по модулю $N$) его правого конца. Для представления количества 1 хватит всего $\log_2 \log_2 N$ битов. Дело в том, что, как нам известно, это число $i$ является степенью 2, скажем $2^j$, так что $i$ можно представить двоичной записью $j$. Поскольку $j$ не больше $\log_2 N$, в его двоичной записи будет не больше $\log_2 \log_2 N$ битов. Таким образом, $O(\log N)$ битов достаточно для представления интервала.

Для представления потока интервалами нужно соблюдать шесть правил.

- Правый конец интервала всегда находится в позиции, содержащей 1.
- Каждая позиция, содержащая 1, принадлежит некоторому интервалу.
- Ни одна позиция не принадлежит сразу нескольким интервалам.
- Существует один или два интервала любого заданного размера, не превосходящего некоторого максимума.
- Все размеры должны быть степенями 2.
- Размеры интервалов не могут уменьшаться при движении влево (назад во времени).

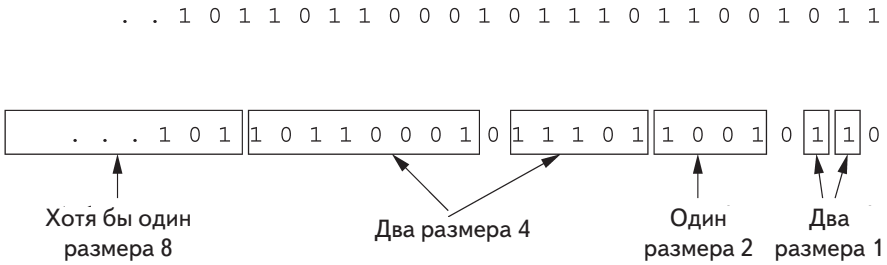



**Рис. 4.2.** Битовый поток, разделенный на интервалы согласно правилам DGIM

**Пример 4.11.** На рис. 4.2 показан битовый поток, разделенный на интервалы, как предписывают правила DGIM. Справа (последние по времени) мы видим два интервала размера 1. Слева от них находится один интервал размера 2. Отметим, что он покрывает четыре позиции, но только в двух из них находятся 1. Двигаясь дальше влево, мы видим два интервала размера 4, и можно предположить, что слева от них расположен интервал размера 8.

Отметим, что нулям разрешено находиться между интервалами. Кроме того, обратите внимание, что интервалы на рис. 4.2 не пересекаются; существует один или два интервала любого размера вплоть до максимального, и при движении справа налево размер никогда не уменьшается.

В следующих разделах мы объясним следующие особенности алгоритма DGIM.

1. Почему количество интервалов в представлении окна должно быть небольшим.
2. Как оценить количество единиц в последних $k$ битах для любого $k$ с погрешностью не более 50 %.
3. Как обеспечить выполнение условий DGIM при поступлении новых битов из потока.

## 4.6.3. Требования к объему памяти для алгоритма DGIM

Мы видели, что каждый интервал можно представить $O(\log N)$ битами. Если длина окна равна $N$, то, конечно, единиц не может быть больше $N$. Предположим, что размер наибольшего интервала равен $2^j$. Тогда $j$ не превосходит $\log_2 N$, иначе в этом интервале было бы больше единиц, чем во всем окне. Таким образом, существует не более двух интервалов каждого размера от $\log_2 N$ до 1 и ни одного интервала большего размера.

Отсюда мы заключаем, что имеется $O(\log N)$ интервалов. Поскольку каждый интервал можно представить $O(\log N)$ битами, общий объем памяти, необходимый для всех интервалов, представляющих окно размера $N$, равен $O(\log^2 N)$.

## 4.6.4. Ответы на вопросы в алгоритме DGIM

Пусть нам задан вопрос, сколько единиц в последних $k$ битах окна для некоторого $1 \le k \le N$. Найдем интервал $b$ с самой ранней временной меткой, включающий хотя бы некоторые из $k$ последних битов. Количество единиц оценим как сумму размеров всех интервалов справа (т. е. более поздних) от интервала $b$ плюс половина размера самого $b$.

> **Пример 4.12.** Рассмотрим поток на рис. 4.2 и пусть $k = 10$. Тогда спрашивается, сколько единиц среди последних десяти битов 0110010110. Пусть текущая временная метка (метка самого правого бита) равна $t$. тогда два интервала с одной единицей, имеющие временные метки $t - 1$ и $t - 2$, включаются в ответ целиком. Интервал размера 2 с временной меткой $t - 4$ также включается целиком. Однако самый правый интервал размера 4 с временной меткой $t - 8$ включается лишь частично. Мы знаем, что это последний интервал, привносящий вклад в ответ, потому что интервал слева от него имеет временную метку меньше $t - 9$ и потому находится целиком вне окна. С другой стороны, мы знаем, что интервалы справа от него целиком попадают в диапазон, указанный в вопросе, т. к. слева от них существует интервал с временной меткой не менее $t - 9$.
>
> Наша оценка количества единиц в последних десяти позициях – 6. Это число складывается из двух интервалов размера 1, одного интервала размера 2 и половины интервала размера 4, который попадает в диапазон частично. Правильный ответ, как легко видеть, 5.

Предположим, что в описанную выше оценку включен интервал $b$ размера $2^j$, частично находящийся вне диапазона, указанного в вопросе. Рассмотрим отклонение оценки от правильного ответа $c$. Возможны два случая: оценка больше или меньше $c$.

- *Случай 1.* Оценка меньше $c$. В худшем случае все единицы, входящие в $b$, на самом деле находятся внутри диапазона, поэтому оценка не учитывает половину интервала $b$, т. е. $2^{j-1}$ единиц. Но в этом случае $c$ никак не меньше $2^j$; на самом деле, даже $2^{j+1} - 1$, потому что существует по меньшей мере один интервал каждого из размеров $2^{j-1}$, $2^{j-2}$, ..., 1. Следовательно, наша оценка не меньше 50 % $c$.
- *Случай 2.* Оценка больше $c$. В худшем случае только самый правый бит интервала $b$ находится внутри диапазона и существует лишь один интервал каждого размера, меньшего $b$. Тогда $c = 1 + 2^{j-1} + 2^{j-2} + ... + 1 = 2^j$, а данная нами оценка равна $2^{j-1} + 2^{j-1} + 2^{j-2} + ... + 1 = 2^j + 2^{j-1} - 1$. Как видим, оценка превышает $c$ не более чем на 50 %.

## 4.6.5. Поддержание условий DGIM

Пусть окно длины $N$ правильно представлено интервалами, удовлетворяющими условиям DGIM. При поступлении нового бита мы должны модифицировать ин-

тервалы, так чтобы они и дальше представляли окно, не нарушая этих условий. Сначала мы должны

- Проверить самый левый (самый ранний) интервал. Если его временная метка стала меньше текущей временной метки минус $N$, то в этом интервале больше нет ни одной единицы, принадлежащей окну. Поэтому мы удаляем его из списка интервалов.

Далее смотрим, чему равен новый бит: 0 или 1. Если он равен 0, то больше ничего делать с интервалами не нужно, а если 1, то необходимо произвести еще несколько изменений. Во-первых:

- Создать новый интервал с текущей временной меткой и размером 1.

Если ранее существовал только один интервал размера 1, то больше ничего делать не надо. Но если в результате получилось три интервала размера 1, то необходимо объединить два самых левых из них.

- Для объединения любых двух соседних интервалов одного размера мы заменяем их одним интервалом удвоенного размера. Временная метка нового интервала равна временной метке правого (более позднего по времени) из двух объединяемых интервалов.

При объединении двух интервалов размера 1 может образоваться третий интервал размера 2. В таком случае объединяем два самых левых интервала размера 2 в интервал размера 4. Это, в свою очередь, может привести к образованию третьего интервала размера 3, и тогда мы объединим два самых левых в интервал размера 8. Этот процесс может распространиться на интервалы нескольких размеров, но всего размеров не более $\log_2 N$, а объединение двух соседних интервалов занимает постоянное время. Поэтому для обработки нового бита потребуется время порядка $O(\log N)$.

**Пример 4.13.** Пусть уже имеются интервалы, показанные на рис. 4.2, и поступает новый бит 1. Видно, что левый интервал не вышел за пределы окна, поэтому никакие интервалы не удаляются. Мы создаем новый интервал размера 1 с текущей временной меткой, скажем $t$. Теперь появилось три интервала размера 1, поэтому объединяем два левых, заменяя их интервалом размера 2. Его временная метка равна $t - 2$, временной метке правого из объединенных интервалов (т. е. самого правого из присутствующих на рис. 4.2).

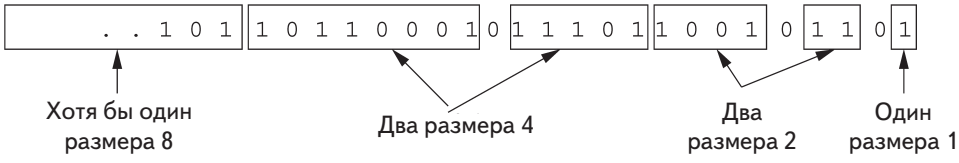



**Рис. 4.3.** Результат модификации интервалов после поступления нового бита 1

Теперь имеется два интервала размера 2, но это правила DGIM допускают. Таким образом, после добавления бита 1 получается последовательность интервалов, показанная на рис. 4.3.

## 4.6.6. Уменьшение погрешности

Допустим, что мы разрешаем не один или два интервала каждого размера, а $r - 1$ или $r$ интервалов, размеры которых образуют геометрическую прогрессию 1, 2, 4, ..., для некоторого целого $r > 2$. Чтобы представить любое возможное число единиц, мы должны ослабить это условие для интервалов размера 1 и максимального из присутствующих размеров: таковых может быть любое число от 1 до $r$.

Правило объединения интервалов практически такое же, как в разделе 4.6.5. Если образовалось $r + 1$ интервалов размера $2^j$, то объединяем два самых левых в интервал размера $2^{j+1}$. При этом может образоваться $r + 1$ интервалов размера $2^{j+1}$, и тогда мы продолжаем объединение.

Доказательство из раздела 4.6.4 проходит и здесь. Но поскольку интервалов маленьких размеров больше, можно получить более сильную оценку погрешности. Мы видели, что наибольшая относительная погрешность возникает, когда внутри диапазона находится только одна единица из самого левого интервала $b$. Пусть размер $b$ равен $2^j$. Тогда истинное количество единиц не меньше $1 + (r - 1)(2^{j-1} + 2^{j-2} + ... + 1) = 1 + (r - 1)(2^j - 1)$. Оценка превышает эту величину на $2^{j-1} - 1$. Поэтому относительная погрешность равна

$$\frac{2^{j-1}-1}{1+(r-1)(2^j-1)}.$$

Каково бы ни было $j$, эта дробь ограничена сверху величиной $1/(r - 1)$. Следовательно, выбрав достаточно большое $r$, мы сможем ограничить погрешность произвольном малым $\varepsilon > 0$.

**Размеры интервалов и каскадные сумматоры**

При выполнении базового алгоритма из раздела 4.6.5 можно проследить закономерность в распределении размеров интервалов. Представьте, что два интервала размера $2^j$ – это «1» в позиции $j$, а один интервал размера $2^j$ – «0» в той же позиции. Тогда последовательность размеров интервалов можно интерпретировать как двоичное число. А случающиеся иногда длинные цепочки объединений интервалов при поступлении из потока единицы – как «волну» переносов при переходе, например, от числа 101111 к 110000.

## 4.6.7. Обобщения алгоритма подсчета единиц

Возникает естественный вопрос: «А нельзя ли обобщить описанный выше алгоритм на выполнение более общих операций агрегирования, чем подсчет количе-

ства единиц в двоичном потоке?». Очевидное направление обобщения – рассмотреть потоки целых чисел и попробовать оценить сумму последних $k$ чисел для любого $1 \le k \le N$, где $N$, как обычно, размер окна.

Подход DGIM вряд ли удастся распространить на потоки, содержащие как положительные, так и отрицательные числа. Ведь поток может содержать очень большие положительные и отрицательные числа, сумма которых очень близка к 0. Любая неточность в оценке значений этих больших чисел окажет огромное влияние на оценку суммы, так что относительная погрешность может быть неограниченной.

Например, предположим, что мы разбили поток на интервалы, но представляем интервал суммой входящих в него целых чисел, а не счетчиком единиц. Если интервал $b$ частично выходит за границы диапазона, то может оказаться, что в первой половине $b$ находятся только очень большие отрицательные числа, а во второй половине – равные им по абсолютной величине положительные числа, так что общая сумма равна 0. Вклад $b$, оцениваемый половиной суммы, равен 0. Но фактический вклад той части $b$, которая находится внутри диапазона, может быть любым числом от 0 до суммы всех положительных чисел. Это отклонение может оказаться гораздо больше истинного результата, так что оценка становится бессмысленной.

Но некоторые обобщения на целые числа все же существуют. Предположим, что поток состоит только из положительных целых чисел в диапазоне от 1 до $2^m$ для некоторого $m$. Можно рассматривать каждый из $m$ битов каждого целого как отдельный поток и применить алгоритм DGIM для подсчета единиц в каждом бите. Обозначим $c_i$ такой счетчик для $i$-го бита (в предположении, что биты нумеруются от младшего к старшему, начиная с 0). Тогда сумма всех целых чисел равна

$$\sum_{i=0}^{m-1} c_i 2^i .$$

Если воспользоваться описанным в разделе 4.6.6 методом для оценки каждого $c_i$ с относительной погрешностью не больше $\varepsilon$, то погрешность оценки истиной суммы тоже не будет превосходить $\varepsilon$. Худший случай имеет место, когда оценки всех $c_i$ завышены или занижены на одну и ту же величину.

## 4.6.8. Упражнения к разделу 4.6

**Упражнение 4.6.1.** Пусть имеется окно, показанное на рис. 4.2. Оцените количество единиц в последних $k$ позициях, если $k$ = (*а*) 5, (*б*) 15. Насколько оценка отличается от истинного значения в каждом случае?

**! Упражнение 4.6.2.** Существует несколько способов разбить битовый поток 1001011011101 на интервалы. Найдите все.

**Упражнение 4.6.3.** Опишите, что произойдет с интервалами, если в окно на рис. 4.3 поступят из потока еще три единицы. Можете считать, что ни одна из показанных единиц не покидает пределы окна.

# 4.7. Затухающие окна

Мы предполагали, что в скользящем окне хранится тот или иной хвост потока: либо последние $N$ элементов для фиксированного $N$, либо все элементы, поступившие позже некоторого момента в прошлом. Иногда мы не хотим четко различать недавние элементы и элементы из отдаленного прошлого, но недавним элементам хотели бы приписать больший вес. В этом разделе мы рассмотрим «экспоненциально затухающие окна» и приложение, в котором они весьма полезны: поиск самых частых «недавних» элементов.

## 4.7.1. Задача о самых частых элементах

Пусть элементами потока являются билеты в кино, проданные по всему миру, и частью элемента является название фильма. Мы хотим знать, какие фильмы наиболее популярны «сейчас». Понятие «сейчас» расплывчато, но интуитивно понятно, что мы хотели бы снизить популярность «Звездных войн, эпизод 4», на который было продано много билетов, но по большей части много десятилетий назад. С другой стороны, фильм, на который продано $n$ билетов в каждую из последних 10 недель, очевидно более популярен, чем фильм, на который было продано $2n$ билетов в последнюю неделю, но ни одного раньше.

Одно из возможных решений – представить каждый фильм битовым потоком. Значение $i$-го бита равно 1, если $i$-й билет продан на этот фильм, и 0 в противном случае. Возьмем окно размера $N$, где $N$ – количество недавно проданных билетов, учитываемое при вычислении популярности. Затем воспользуемся методом из раздела 4.6 для оценки количества билетов, проданных на каждый фильм, и ранжируем фильмы по найденным оценкам. Эта техника вполне может оказаться работоспособной для фильмов, поскольку их всего несколько тысяч, но неприменима для оценки популярности товаров на сайте Amazon или частоты отправки пользователями сообщений в Twitter, поскольку и товаров, и пользователей слишком много. Кроме того, она дает только приближенные ответы.

## 4.7.2. Определение затухающего окна

Альтернативный подход – переформулировать задачу так, чтобы речь не шла о подсчете единиц в окне. Вместо этого будем вычислять гладкий агрегат всех виденных в потоке единиц с убывающими весами, т. е. чем раньше встречалась единица, тем меньший вес ей приписывается. Формально, пусть поток состоит из элементов $a_1, a_2, ..., a_t$, где $a_1$ – первый поступивший элемент, а $a_t$ – текущий. Пусть $c$ – малая константа, скажем $10^{-6}$ или $10^{-9}$. Определим *экспоненциально затухающее окно* для этого потока как сумму

$$\sum_{i=0}^{t-1} a_{t-i}(1-c)^i .$$

Смысл этого определения заключается в том, чтобы распространить веса элементов на все время существования потока. Напротив, в случае фиксированного

окна с той же суммой весов $1/c$ мы приписали бы одинаковый вес 1 каждому из последних $1/c$ элементов и вес 0 – всем предшествующим. Это различие показано на рис. 4.4.

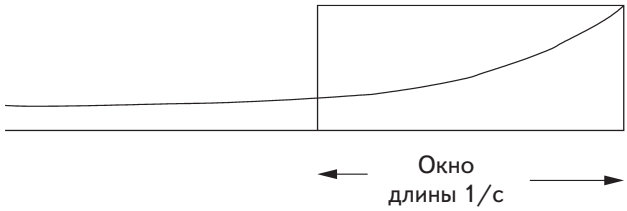



**Рис. 4.4.** Затухающее окно и окно фиксированной длины с одинаковыми весами

Гораздо проще пересчитать сумму в экспоненциально затухающем окне, чем в скользящем окне фиксированной длины. В скользящем окне при поступлении каждого нового элемента мы должны думать о выпадающем из окна элементе. Поэтому приходится хранить не только сумму, но и точное количество элементов или применять приближенные схемы типа DGIM. А в случае затухающего окна при поступлении элемента $a_{t+1}$ нужно всего лишь:

1. Умножить текущую сумму на $1 - c$.
2. Прибавить $a_{t+1}$.

Этот способ работает, потому что каждый предшествующий элемент теперь сдвинулся на одну позицию дальше от текущего, поэтому его вес умножается на $1 - c$. Кроме того, вес текущего элемента равен $(1 - c)^0 = 1$, поэтому вклад $a_{t+1}$ учтен правильно.

## 4.7.3. Нахождение самых популярных элементов

Вернемся к задаче о нахождении самых популярных фильмов в потоке данных о проданных билетах[5]. Мы воспользуемся экспоненциально затухающим окном с константой $c$, которую будем считать равной $10^{-9}$. Иначе говоря, аппроксимируем скользящее окно, содержащее данные о последнем миллиарде проданных билетов. Каждый фильм представим отдельным воображаемым потоком, в котором единицы находятся в позициях, соответствующих проданным на него билетам в реальном потоке, а нули – позициям, соответствующим другим фильмам. Затухающая сумма единиц измеряет текущую популярность фильма.

Вообразим, что количество различных фильмов в потоке очень велико, так что хранить данные о непопулярных фильмах не хочется. Поэтому установим порог, скажем 1/2, такой, что если оценка популярности опускается ниже, то она исключается из подсчета. По причинам, которые вскоре станут понятны, порог должен быть меньше 1. При поступлении нового билета сделаем следующее:

[5] Этот пример следует воспринимать с долей скептицизма, потому что, как мы отмечали, число фильмов недостаточно велико, чтобы оправдать применение этой техники. Если хотите, представьте, что фильмов очень много, так что точно подсчитать, сколько продано билетов на каждый, невозможно.

1. Для каждого фильма, оценка которого еще подсчитывается, умножим оценку на $(1 - c)$.
2. Пусть новый билет продан на фильм $M$. Если существует оценка для $M$, прибавить к ней 1, иначе создать оценку для $M$ и инициализировать ее значением 1.
3. Если оценка оказывается меньше $1/2$, исключить ее.

Сразу не очевидно, почему количество оцениваемых фильмов в любой момент ограничено. Заметим, однако, что сумма всех оценок равна $1/c$. Поэтому не может существовать более $2/c$ фильмов с оценкой $1/2$ или выше. Следовательно, никогда не может быть более $2/c$ фильмов с подсчитываемыми оценками. Конечно, на практике в каждый момент времени продаются билеты на сравнительно небольшое число фильмов, гораздо меньшее $2/c$.

# 4.8. Резюме

- *Потоковая модель данных*. В этой модели предполагается, что данные поступают обрабатывающей системе в таком темпе, что сохранить все в активном хранилище практически нереально. Одна из стратегий работы с потоками – хранить обобщенные сведения о потоках, достаточные для ответа на ожидаемые запросы. Другой подход – поддержать скользящее окно, содержащее недавние данные.
- *Выборка из потока*. Чтобы создать выборку, пригодную для некоторого класса запросов, мы выделяем множество ключевых атрибутов. Если хэшировать ключ каждого поступающего элемента, то мы сможем использовать хэш-код, чтобы принять непротиворечивое решение о том, следует ли включать в выборку элементы с таким ключом.
- *Фильтр Блума*. Этот метод позволяет фильтровать поток, так что элементы, принадлежащие некоторому множеству, пропускаются, а большинство остальных отбрасывается. Для каждого элемента выбранного множества вычисляется хэш-код, равный номеру бита в массиве, и эти биты устанавливаются в 1. Чтобы проверить, принадлежит ли поступивший из потока элемент множеству, нужно применить к нему каждую из набора хэш-функций и пропускать элемент, только если все хэш-коды соответствуют битам, равным 1.
- *Подсчет различных элементов*. Чтобы оценить количество различных элементов в потоке, мы можем хэшировать его элементы в целые числа, интерпретируемые как двоичные. Возводим 2 в степень, равную длине самой длинной последовательности нулей, встретившейся в хэш-кодах всех элементов потока, и считаем получившуюся величину оценкой количества различных элементов. Если взять много хэш-функций и объединить оценки, сначала вычислив средние по группам, а затем медианы средних, то получившаяся оценка будет надежной.

- *Моменты потоков.* $k$-ым моментом потока называется сумма $k$-х степеней счетчиков всех элементов, встречавшихся хотя бы один раз. Нулевой момент равен количеству различных элементов, а первый – длине потока.
- *Оценивание второго момента.* Хорошая оценка второго момента, или меры неожиданности, получается следующим образом: выберем случайную позицию в потоке, посчитаем, сколько раз элемент, находящийся в этой позиции, встречается в потоке, начиная с нее и далее, умножим эту величину на два, вычтем из результата 1 и умножим на длину потока. Чтобы получить надежную оценку второго момента, объединим много таких случайных переменных так же, как делали при подсчете числа различных элементов.
- *Оценивание моментов высшего порядка.* Метод, примененный для вторых моментов, работает и для $k$-х моментов, если заменить выражение $2x - 1$ (где $x$ – количество вхождений элемента, начиная с выбранной позиции) на $x^k - (x-1)^k$.
- *Оценивание количества единиц в окне.* Чтобы оценить количество единиц в окне, содержащем нули и единицы, мы можем сгруппировать единицы в интервалы. Количество единиц в каждом интервале – степень 2. Существует один или два интервала каждого размера и при движении назад во времени размер не убывает. Если хранить только позиции и размеры интервалов, то содержимое окна размера $N$ можно представить в памяти объемом порядка $O(\log^2 N)$.
- *Ответы на вопросы о количестве единиц.* Чтобы узнать приблизительное количество единиц среди $k$ последних элементов двоичного потока, мы можем найти самый ранний интервал $B$, который хотя бы частично пересекается с последними $k$ позициями окна, а затем вычислить количество единиц как сумму размеров каждого следующего за ним интервала и прибавить половину размера $B$. Эта оценка никогда не отклоняется от истинного числа единиц более чем на 50 %.
- *Более точные аппроксимации количества единиц.* Мы можем изменить правило, диктующее, сколько интервалов данного размера может присутствовать в представлении двоичного окна, разрешив присутствие $r$ или $r - 1$ интервалов. Тогда получится аппроксимация истинного числа 1 с относительной погрешностью не выше $1/r$.
- *Экспоненциально затухающие окна.* Вместо окна фиксированного размера мы можем представить, что окно содержит все элементы, когда-либо поступавшие из потока, но элементу, поступившему $t$ единиц времени назад, приписывается вес $e^{-ct}$ для некоторой константы $c$. Это позволяет легко вычислять некоторые обобщенные сведения об экспоненциально затухающем окне. Например, чтобы пересчитать взвешенную сумму элементов при поступлении нового, нужно умножить предыдущую сумму на $1 - c$ и прибавить новый элемент.

- *Хранение частых элементов в экспоненциально затухающем окне*. Вообразим, что каждый элемент представлен двоичным потоком, в котором 0 означает, что в данный момент времени поступил не этот элемент, а 1 – что этот. Мы можем найти элементы, для которых сумма их двоичных потоков не меньше 1/2. При поступлении нового элемента умножаем все хранящиеся суммы на $1 - c$, прибавляем 1 к счетчику, соответствующему поступившему элементу, и перестаем хранить те элементы, для которых сумма опустилась ниже 1/2.

# 4.9. Список литературы

Многие идеи, относящиеся к управлению потоками, впервые появились в «модели исторических данных» из работы [8]. Один из первых обзоров исследований в области систем управления потоками имеется в работе [2]. Этой теме посвящена также недавно вышедшая книга [6].

Метод выборки из потока, описанный в разделе 4.2, взят из работы [7]. Фильтр Блума обычно связывают с работой [3], хотя такая же, по существу, техника под названием «суперпозиционные коды» изложена в [9].

Алгоритм подсчета различных элементов в основном заимствован из [5], хотя конкретный описанный нами метод опубликован в [1]. Оттуда же взят алгоритм вычисления моментов второго и более высоких порядков. Однако техника поддержания равномерно распределенной выборки позиций в потоке, называемая «выборка с резервуаром» (reservoir sampling), взята из работы [10].

Метод приближенного подсчета количества единиц в окне описан в [4].

1. N. Alon, Y. Matias, M. Szegedy «The space complexity of approximating frequency moments» 28th ACM Symposium on Theory of Computing, pp. 20–29, 1996.
2. B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom «Models and issues in data stream systems» Symposium on Principles of Database Systems, pp. 1–16, 2002.
3. B. H. Bloom «Space/time trade-offs in hash coding with allowable errors» Comm. ACM 13:7, pp. 422–426, 1970.
4. M. Datar, A. Gionis, P. Indyk, R. Motwani «Maintaining stream statistics over sliding windows» SIAM J. Computing 31, pp. 1794–1813, 2002.
5. P. Flajolet, G.N. Martin, «Probabilistic counting for database applications» 24th Symposium on Foundations of Computer Science, pp. 76–82, 1983.
6. M. Garofalakis, J. Gehrke, R. Rastogi (editors) *Data Stream Management*, Springer, 2009.
7. P. B. Gibbons «Distinct sampling for highly-accurate answers to distinct values queries and event reports» Intl. Conf. on Very Large Databases, pp. 541–550, 2001.
8. H. V. Jagadish, I.S. Mumick, A. Silberschatz «View maintenance issues for the chronicle data model» Proc. ACM Symp. on Principles of Database Systems, pp. 113–124, 1995.

9. W. H. Kautz, R.C. Singleton «Nonadaptive binary superimposed codes» IEEE Transactions on Information Theory 10, pp. 363–377, 1964.
10. J. Vitter «Random sampling with a reservoir» ACM Transactions on Mathematical Software 11:1, pp. 37–57, 1985.

# ГЛАВА 5.
# Анализ ссылок

Одно из важнейших изменений в нашей жизни за десять лет нового века – появление эффективного и точного поиска в вебе благодаря таким поисковым системам, как Google. Хотя поисковики существовали и до Google, именно она впервые поставила заслон на пути спамеров, из-за которых поиск стал почти бесполезным. Заслугой Google является также нетривиальное техническое достижение, получившее название «PageRank». Мы начнем эту главу с объяснения того, что такое PageRank и как его эффективно вычислить.

Тем не менее, война между теми, кто хочет сделать веб полезным инструментом, и тем, кто стремится эксплуатировать его в собственных корыстных целях, никогда не закончится. Когда алгоритм PageRank утвердился в качестве основного метода в поисковых системах, спамеры нашли способы манипулировать рангом веб-страницы. Эту технику часто называют ссылочным спамом[1]. В ответ был изобретен алгоритм TrustRank и другие способы противодействия атакам на PageRank. Мы обсудим TrustRank и другие подходы к обнаружению ссылочного спама.

Наконец, в этой главе рассматриваются и несколько вариаций на тему PageRank. К ним относятся тематический PageRank (topic-sensitive PageRank), который также можно приспособить для борьбы со ссылочным спамом, и подход на основе «хабов и авторитетов» (HITS) к оценке страниц в веб.

## 5.1. PageRank

Немного углубимся в историю поисковых систем, чтобы понять мотивы определения PageRank[2], инструмента для оценки важности веб-страниц, который трудно одурачить. Понять причины эффективности PageRank нам поможет понятие «случайного пользователя». Затем мы познакомимся с техникой «телепортации», или рециркуляции пользователей, которая позволяет избежать некоторых структур в вебе, представляющих проблемы для простого варианта PageRank.

---

[1] Иногда ссылочные спамеры пытаются замаскировать безнравственность своей деятельности, называя ее «поисковой оптимизацией».

[2] Термин PageRank изобретен Ларри Пейджем (Larry Page), автором идеи и сооснователем Google.

### 5.1.1. Ранние поисковые системы и спам термов

До Google было много поисковых систем. В большинстве своем они занимались обходом веба и сохранением найденных на странице термов (слов или других строк непробельных символов) в инвертированном индексе. Инвертированный индекс – это структура данных, которая по терму позволяет быстро находить указатели на все места, где этот терм встречается.

Получив поисковый запрос (список термов), система извлекала из инвертированного индекса страницы, содержащие указанные термы, и ранжировала их с учетом использования терма на странице. Так, если терм встречался в заголовке, то страница считалась более релевантной, чем та, что содержала терм в обычном тексте. Если терм встречался много раз, то это увеличивало релевантность страницы

Когда поисковые системы стали набирать популярность, бессовестные люди увидели для себя возможность обмануть поисковую систему и направить аудиторию на свою страницу. Так, продавцу рубашек было важно заманить человека на свой сайт вне зависимости от того, что он искал. Например, можно поместить на свою страницу слово «фильм», повторив его тысячу раз, чтобы поисковая система думала, что это очень важная страница о кино. И если пользователь отправит запрос со словом «фильм», то система поставит вашу страницу на первое место. А чтобы тысяча слов «фильм» не была видна, их можно нарисовать тем же цветом, что и фон. А если одного лишь добавления слова «фильм» окажется недостаточно, то можно отправить поисковой системе запрос «фильм» и посмотреть, какая страница окажется первой. После чего скопировать ее на свою собственную, закрасив цветом фона.

Техника обмана поисковых систем с целью заставить их поверить, что некая страница является не тем, что представляет собой на самом деле, называется *спамом термов*. Из-за простоты ее применения ранние поисковые системы стали почти бесполезными. Для борьбы со спамом термов Google ввела два новшества.

1. Алгоритм PageRank использовался для моделирования того, где в конечном итоге соберутся пользователи веба, если начнут блуждание со случайной страницы и будут следовать по случайно выбранным исходящим ссылкам с текущей страницы, повторяя это действие много раз. Страницы, на которые заходит много пользователей, считаются более «важными», чем те, которые посещаются редко. Решая, какие страницы поставить на первое место в списке результатов запроса, Google отдает предпочтение важным.
2. Содержимое страницы оценивалось не только по встречающимся на ней термам, но и по термам, встречающимся в ссылках на эту страницу или рядом с ними. Отметим, что спамер легко может добавить фальшивые термы на страницу, которую контролирует, но вот проделать то же самое с чужими страницами, которые на нее ссылаются, гораздо сложнее.

**Упрощенный PageRank не работает**

Как мы увидим, вычисление PageRank путем моделирования случайных пользователей отнимает очень много времени. Можно было бы подумать, что простой подсчет количества ссылок, ведущих на каждую страницу, является неплохой аппроксимацией мест скопления случайных пользователей. Но если бы мы этим и ограничились, то гипотетический продавец рубашек мог бы просто создать «спам-ферму», содержащую миллионы страниц, ведущих на страницу его магазина. Тогда страница о рубашках показалась бы системе чрезвычайно важной, и она осталась бы в дураках.

В совокупности эти два приема чрезвычайно затрудняют гипотетическому продавцу рубашек обман Google. Он, конечно, может добавить на свою страницу слово «фильм», но Google верит тому, что говорят о нем другие страницы, а не он сам, поэтому фальшивые термы проигнорирует. Продавец рубашек может предпринять очевидную контрмеру – создать много других страниц и сослаться из них на страницу о рубашках, включив в ссылку слово «фильм». Но PageRank не придаст этим страницам большой важности, потому что на них не ссылаются другие страницы. Продавец рубашек мог бы создать много перекрестных ссылок между своими страницами, но ни одну из них алгоритм PageRank не сочтет достаточно важной, так что спамер не сумеет убедить Google в том, что его страница посвящена фильмам.

Разумно спросить, почему моделирование случайных пользователей позволяет аппроксимировать интуитивное представление о «важности» страниц. У этого подхода есть два взаимосвязанных обоснования.

- Авторы веб-страниц «голосуют ногами». Они обычно размещают ссылки на страницы, которые считают хорошими или полезными, а не на плохие и бесполезные.
- Поведение случайного пользователя определяет, какие страницы пользователи посетят с наибольшей вероятностью. Пользователь скорее зайдет на полезную страницу, чем на бесполезную.

Но каковы бы ни были причины, показатель PageRank эмпирически доказал свою пригодность, поэтому изучим подробнее, как он вычисляется.

## 5.1.2. Определение PageRank

PageRank – это функция, которая сопоставляет вещественное число каждой веб-странице (или, по крайней мере, тем, которые робот нашел по ссылкам). Идея в том, что чем выше PageRank страницы, тем она «важнее». Не существует единственного фиксированного алгоритма вычисления PageRank, различные варианты базовой идеи могут изменять относительные ранги любой пары страниц. Начнем с определения базового, идеализированного PageRank, а затем рассмо-

трим модификации, необходимые для решения некоторых реальных проблем, связанных со структурой веба.

Будем рассматривать веб как ориентированный граф, в котором вершинами являются страницы и вершина $p_1$ соединена ребром с $p_2$, если с $p_1$ на $p_2$ ведет хотя бы одна ссылка. На рис. 5.1 приведен пример крохотного фрагмента веба, в котором есть всего четыре страницы. На странице $A$ имеются ссылки на остальные три страницы, на странице $B$ – только на $A$ и $D$, на странице $C$ – только на $A$, а на странице $D$ – только на $B$ и $C$.

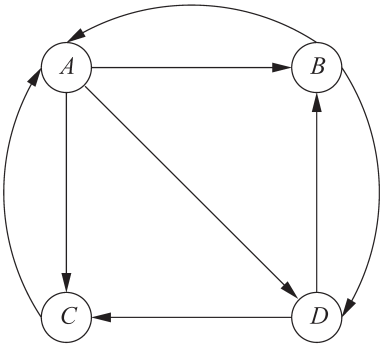



**Рис. 5.1.** Гипотетический пример веба

Допустим, что случайный посетитель начинает блуждание со страницы $A$. С нее ведут ссылки на $B$, $C$ и $D$, поэтому пользователь может оказаться на каждой из этих страниц с вероятностью 1/3, а вероятность попасть на страницу $A$ равна нулю. На следующем шаге, оказавшись в $B$, случайный пользователь с вероятностью 1/2 попадет в $A$, с вероятностью 1/2 – в $D$ и с вероятностью 0 – в $B$ или $C$.

В общем случае мы можем определить *матрицу переходов в вебе*, которая описывает, что происходит со случайными пользователями на каждом шаге. Если имеется $n$ страниц, то эта матрица $M$ будет состоять из $n$ строк и $n$ столбцов. Элемент $m_{ij}$ на пересечении строки $i$ и столбца $j$ равен $1/k$, если из вершины $j$ исходит $k$ ребер, и одно из них ведет в вершину $i$. В противном случае $m_{ij} = 0$.

**Пример 5.1.** Матрица переходов для фрагмента веба на рис. 5.1.

$$M = \begin{bmatrix} 0 & 1/2 & 1 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

В этой матрице сохранен естественный порядок страниц: $A$, $B$, $C$, $D$. Таким образом, первый столбец выражает тот факт, что случайный пользователь, находящийся в $A$, на следующем шаге с вероятностью 1/3 попадет на любую из трех других страниц. Второй столбец говорит, что пользователь, находящийся в $B$, с вероятностью 1/2 перейдет в $A$ или в $D$, третий столбец – что пользователь из $C$ непременно попадет в $A$. И последний столбец означает, что пользователь, находящийся в $D$, с вероятностью 1/2 окажется в $B$ или в $C$.

Распределение вероятностей местонахождения случайного пользователя можно описать вектором, $j$-й элемент которого содержит вероятность того, что пользователь окажется на странице $j$. Эта вероятность и есть (идеализированная) функция *PageRank*.

Предположим, что случайный пользователь с равной вероятностью может начать блуждание с любой из $n$ страниц веба. Тогда все элементы начального вектора $\mathbf{v}_0$ будут равны $1/n$. Если $M$ – матрица переходов в вебе, то после одного шага вектор распределения вероятностей станет равен $M\mathbf{v}_0$, после двух – $M(M\mathbf{v}_0) = M^2\mathbf{v}_0$ и т. д. Вообще, распределение вероятностей после $i$ шагов получается путем $i$-кратного умножения $M$ на начальный вектор $\mathbf{v}_0$.

Чтобы понять, почему произведение $M$ на вектор $\mathbf{v}$ дает распределение на следующем шаге, будем рассуждать следующим образом. Вероятность $\mathbf{x}_i$, что случайный пользователь на следующем шаге окажется в вершине $i$ равна $\sum_j m_{ij}\mathbf{v}_j$. Здесь $m_{ij}$ – вероятность, что пользователь, находящийся в вершине $j$, перейдет в вершину $i$ (часто она равна 0, потому что из $j$ в $i$ нет ссылок), а $\mathbf{v}_j$ – вероятность, что на предыдущем шаге пользователь находился в вершине $j$.

Такое поведение – пример старой теории *марковских процессов*. Известно, что распределение вероятностей для случайного пользователя стремится к предельному распределению $\mathbf{v}$, удовлетворяющему уравнению $\mathbf{v} = M\mathbf{v}$, если выполнены два условия:

1. Граф является *сильно связным*, т. е. из любой вершины можно попасть в любую другую.
2. Не существует *тупиков* – вершин, из которых не исходит ни одно ребро.

Граф на рис. 5.1 удовлетворяет обоим условиям.

Предел достигается, когда после очередного умножения вектора распределения на $M$ вектор не изменяется. Иными словами, предельное распределение $\mathbf{v}$ – это собственный вектор $M$ (*собственным вектором* матрицы $M$ называется такой вектор $\mathbf{v}$, что $\mathbf{v} = \lambda M\mathbf{v}$ для некоторого постоянного *собственного значения* $\lambda$). На самом деле, поскольку $M$ – *стохастическая* матрица, т. е. сумма элементов в каждом столбце равна 1, $\mathbf{v}$ является *главным* собственным вектором (с ним ассоциировано максимальное собственное значение). Заметим также, что поскольку матрица $M$ стохастическая, собственное значение, ассоциированное с главным собственным вектором, равно 1.

Главный собственный вектор $M$ говорит о том, где случайный пользователь с наибольшей вероятностью окажется спустя длительное время. Напомним, что за алгоритмом PageRank стоит интуитивное соображение – чем выше вероятность оказаться в некоторой странице, тем важнее эта страница. Для вычисления главного собственного вектора $M$ мы можем взять начальный вектор $\mathbf{v}_0$ и умножать его на $M$, пока вектор не перестанет заметно изменяться. Для реального веба 50–75 итераций достаточно для сходимости при вычислениях с двойной точностью.

**Пример 5.2.** Применим описанный выше процесс к матрице $M$ из примера 5.1. Поскольку в графе всего четыре вершины, начальный вектор $\mathbf{v}_0$ будет состоять из четырех элементов, равных 1/4. Последовательные приближения, получаемые умножением на $M$, выглядят так:

$$\begin{bmatrix}1/4\\1/4\\1/4\\1/4\end{bmatrix}, \begin{bmatrix}9/24\\5/24\\5/24\\5/24\end{bmatrix}, \begin{bmatrix}15/48\\11/48\\11/48\\11/48\end{bmatrix}, \begin{bmatrix}11/32\\7/32\\7/32\\7/32\end{bmatrix}, \ldots, \begin{bmatrix}3/9\\2/9\\2/9\\2/9\end{bmatrix}$$

Отметим, что в этом примере вероятности попасть в *B*, *C* и *D* остаются равными. Легко видеть, что для *B* и *C* вероятности должны быть равны на всех итерациях, потому что соответствующие им строки *M* одинаковы. Для доказательства того, что они совпадают с вероятностью для *D*, можно применить метод математической индукции; оставляем это в качестве упражнения для читателя. С учетом того, что последние три элемента предельного вектора должны быть равны, легко найти предел последовательности выше. Первая строка *M* говорит, что вероятность *A* должна быть в 3/2 раза больше других вероятностей, поэтому в пределе вероятность *A* равна 3/9, или 1/3, тогда как остальные три вероятности равны 2/9.

Разница между этими вероятностями не очень велика. Но в реальном вебе с миллиардами страниц существенно различной важности истинная вероятность оказаться на странице типа `www.amazon.com` на много порядков выше, чем для какой-нибудь типичной страницы.

**Решение линейных уравнений**

Взглянув на «веб» с 4 вершинами на рис. 5.2, вы можете подумать, что для решения уравнения $\mathbf{v} = M\mathbf{v}$ нужно применить метод Гаусса. И в этом примере мы действительно нашли предел именно таким способом. Но в реальных ситуациях, когда число вершин исчисляется десятками или сотнями миллиардов, исключение переменных по методу Гауссу неосуществимо, т. к. время работы этого алгоритма кубически зависит от числа уравнений. Поэтому единственный способ решения систем линейных уравнений такого масштаба – предложенная нами последовательность итераций. И хотя на каждом шаге итерация требует квадратичного времени, мы можем ускорить процесс, воспользовавшись тем фактом, что матрица *M* сильно разрежена – в среднем на каждой странице примерно 10 ссылок, т. е. 10 ненулевых элементов в каждом столбце.

Есть и еще одно различие между вычислением PageRank и решением линейных уравнений. Число решений уравнения $\mathbf{v} = M\mathbf{v}$ бесконечно, потому что мы можем взять любое решение $\mathbf{v}$, умножить его на произвольную константу *c* и получить другое решение. Потребовав, чтобы сумма элементов была равна 1 (как мы и сделали), мы получим единственное решение.

## 5.1.3. Структура веба

Как было бы хорошо, если бы граф веба был сильно связным, как на рис. 5.1. Увы, на самом деле это не так. В ранних исследованиях веба было показано, что его

структура выглядит, как на рис. 5.2. Существует большая сильно связная компонента (ССК), а также еще несколько частей, почти не уступающих ей по размеру.

1. *Входящая компонента*, состоящая из страниц, с которых можно попасть в ССК, следуя по ссылкам, но недостижимых из ССК.
2. *Исходящая компонента*, состоящая из страниц, которые достижимы из ССК, но из которых нельзя попасть в ССК.
3. *Завитки* двух типов. Одни завитки состоят из страниц, которые достижимы из входящей компоненты, но из которых невозможно в нее попасть. Из других можно попасть в исходящую компоненту, но не наоборот.

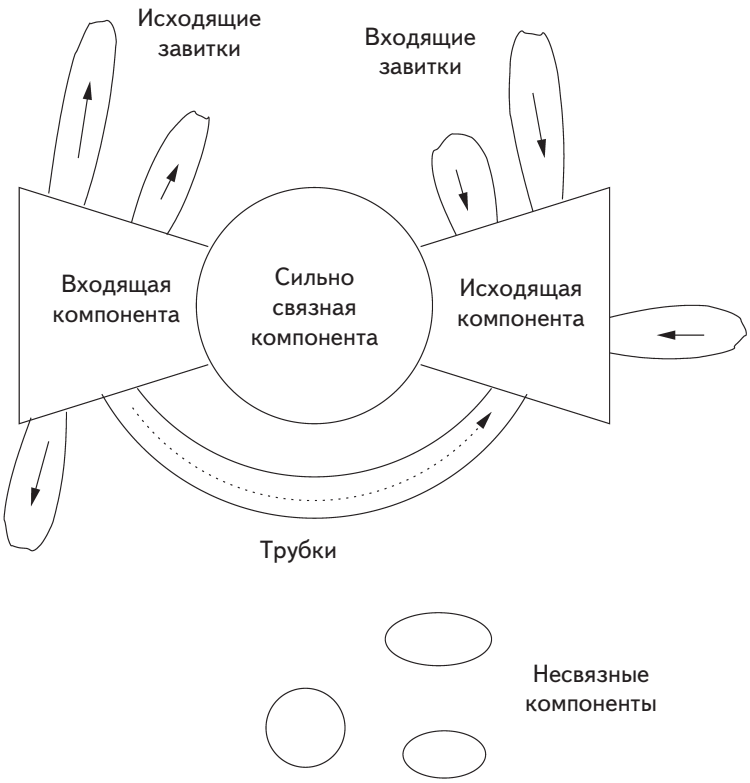



**Рис. 5.2.** Изображение веба в виде галстука-бабочки

Кроме того, сравнительно немного страниц попадают либо в

(*а*) *трубки*, состоящие из страниц, которые достижимы из входящей компоненты, с которых можно попасть в исходящую компоненту, но в которые нельзя перейти из ССК,

либо в

(*б*) изолированные компоненты, которые недостижимы из крупных компонент (ССК, входящая и исходящая) и из которых нельзя в них попасть.

Некоторые из этих структур нарушают предположения, при которых марковский процесс сходится. Например, если случайный пользователь попадет в исходящую компоненту, то уже никогда не сможет ее покинуть. Поэтому пользователи, начавшие блуждание в ССК или входящей компоненте, обречены оказаться в исходящей компоненте или в одном из завитков входящей компоненты. Следовательно, для всех страниц из ССК и входящей компоненты вероятность, что случайный пользователь в конечном итоге попадет в нее, равна нулю. Если интерпретировать эту вероятность, как меру важности страницы, то мы придем к неверному выводу, будто ничто в ССК или входящей компоненте не заслуживает внимания.

Поэтому алгоритм вычисления PageRank обычно модифицируется, чтобы избежать таких аномалий. На самом деле, мы хотим предотвратить две проблемы. Первая – *тупики*, т. е. страницы без исходящих ссылок. Пользователь, попавший на такую страницу, бесследно исчезает, а в пределе это означает, что у любой страницы, с которой достижим тупик, вообще не будет PageRank. Вторая – группы страниц, каждая из которых содержит исходящие ссылки, но все они ведут на какую-то страницу той же группы, а не наружу. Такие структуры называются *паучьими ловушками* (spider traps).[3] Обе проблемы решаются методом «телепортации», когда мы допускаем ненулевую вероятность того, случайный посетитель может исчезнуть на любом шаге и появиться на какой-то другой странице. Мы проиллюстрируем этот процесс ниже.

## 5.1.4. Избегание тупиков

Напомним, что тупиком называется страница без исходящих ссылок. Если разрешить тупики, то матрица переходов в вебе перестанет быть стохастической, поскольку сумма элементов в некоторых столбцах равна не 1, а 0. Матрица, в которой сумма элементов в каждом столбце не больше 1, называется *субстохастической*. Если вычислять $M^i\mathbf{v}$ для возрастающих степеней субстохастической матрицы $M$, то некоторые или все элементы вектора будут стремиться к 0. Таким образом, из веба «вымывается» важность, и мы не получаем никакой информации об относительной важности страниц.

**Пример 5.3.** На рис. 5.3 изображен модифицированный граф, в котором по сравнению с рис. 5.1 удалено ребро, ведущее из *C* в *A*. В результате *C* стала тупиком. В терминах случайных посетителей это означает, что, попав в *C*, посетитель исчезает на следующем шаге. Матрица *M*, описывающая этот граф, выглядит так:

$$M = \begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}.$$

[3] Такое название они получили, потому что программы, которые обходят сеть и сохраняют страницы и ссылки, часто называют «пауками» (spider). Попав в ловушку, паук уже никогда не сможет выбраться.

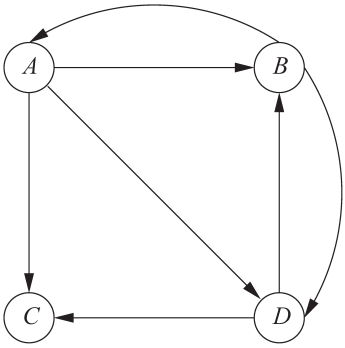



**Рис. 5.3.** Вершина *C* теперь является тупиком

Эта матрица субстохастическая, но не стохастическая, потому что сумма элементов в третьем столбце, соответствующем *C*, равна 0, а не 1. Ниже показана последовательность векторов, получающаяся, если начать с вектора, все элементы которого равны 1/4, и раз за разом умножать его на *M*:

$$\begin{bmatrix}1/4\\1/4\\1/4\\1/4\end{bmatrix}, \begin{bmatrix}3/24\\5/24\\5/24\\5/24\end{bmatrix}, \begin{bmatrix}5/48\\7/48\\7/48\\7/48\end{bmatrix}, \begin{bmatrix}21/288\\31/288\\31/288\\31/288\end{bmatrix}, \ldots, \begin{bmatrix}0\\0\\0\\0\end{bmatrix}.$$

Как видим, вероятность, что пользователь придет хоть куда-нибудь, стремится к 0 при увеличении числа шагов.

К проблеме тупиков есть два подхода.

1. Можно исключить из графа как сами тупики, так и входящие в них ребра. Но при этом могут возникнуть новые тупики, которые тоже придется устранять. И так далее, рекурсивно. Но в конечном итоге мы получим сильно связную компоненту, в которой не будет тупиковых вершин. В терминах рис. 5.2 рекурсивное удаление тупиков приводит к удалению частей исходящей компоненты, завитков и трубок, но оставляет ССК, входящую компоненту и части небольших изолированных компонент[4].
2. Можно модифицировать процесс перемещения случайных пользователей по вебу. Этот метод, называемый «телепортацией», решает заодно и проблему паучьих ловушек, поэтому отложим его обсуждение до раздела 5.1.5.

Если воспользоваться методом удаления тупиков, то оставшийся граф *G* мы разрешим подходящими средствами, включая метод телепортации, если в нем су-

[4] На первый взгляд, кажется, что входящая компонента и завитки будут удалены целиком, но нужно помнить, что внутри них могут существовать меньшие сильно связные компоненты, в том числе паучьи ловушки, которые удалять нельзя.

ществуют паучьи ловушки. После этого восстановим граф, но сохраним значения PageRank, вычисленные для вершин $G$. PageRank вершин, которые не входят в $G$, но для которых в $G$ есть предшественники, можно вычислить, просуммировав по всем предшественникам $p$ ранг PageRank вершины $p$, поделенный на число последователей $p$ в полном графе. При этом могут оказаться другие вершины, не входящие в $G$, но такие, для которых вычислен PageRank всех их предшественников. Их собственный PageRank может быть вычислен тем же процессом. В конечном итоге будут вычислены ранги всех вершин, не входящих в $G$; разумеется, вычисление должно производиться в порядке, обратном порядку, в котором они удалялись.

**Пример 5.4.** На рис. 5.4 изображен граф, в котором по сравнению с рис. 5.3 добавлена вершина $E$, являющаяся последователем $C$. Но $E$ – тупик и после удаления ее самой и ребра, ведущего в нее из $C$, обнаруживается, что теперь стала тупиком вершина $C$. После удаления $C$ других тупиков не останется, поскольку из вершин $A$, $B$ и $D$ исходит по меньшей мере одно ребро. Получившийся граф изображен на рис. 5.5.

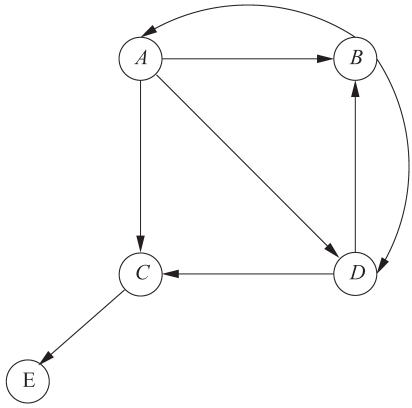



**Рис. 5.4.** Граф с двумя уровнями тупиков

Матрица графа на рис. 5.5 имеет вид

$$M = \begin{bmatrix} 0 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 1/2 & 1/2 & 0 \end{bmatrix}.$$

Строки и столбцы соответствуют $A$, $B$ и $D$ в указанном порядке. Чтобы вычислить PageRank для этой матрицы, начнем с вектора, все элементы которого равны 1/3, и будем повторно умножать его на $M$. Получается такая последовательность векторов:

$$\begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}, \begin{bmatrix} 1/6 \\ 3/6 \\ 2/6 \end{bmatrix}, \begin{bmatrix} 3/12 \\ 5/12 \\ 4/12 \end{bmatrix}, \begin{bmatrix} 5/24 \\ 11/24 \\ 8/24 \end{bmatrix}, \cdots, \begin{bmatrix} 2/9 \\ 4/9 \\ 3/9 \end{bmatrix}.$$

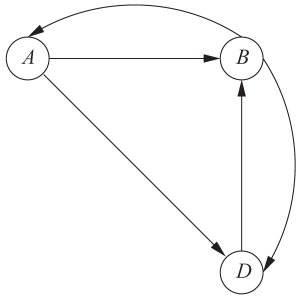



**Рис. 5.5.** Редуцированный граф без тупиков

Теперь мы знаем, что для вершины *A* PageRank равен 2/9, для *B* – 4/9, а для *D* –3/9. Но нам еще нужно вычислить ранги вершин *C* и *E*; сделаем это в порядке, обратном порядку их удаления. Поскольку *C* удалялась последней, мы знаем, что для всех ее предшественников PageRank уже вычислен. Предшественниками являются вершины *A* и *D*. На рис. 5.4 у *A* три предшественника, поэтому она вносит в ранг *C* вклад 1/3. У вершины *D* два предшественника, так что ее вклад в ранг *C* равен 1/2. Следовательно, ранг *C* равен $1/3 \times 2/9 + 1/2 \times 3/9 = 13/54$.

Теперь мы можем вычислить ранг для *E*. У этой вершины только один предшественник *C*, а у *C* только один последователь. Следовательно, ранг *E* такой же, как у *C*. Отметим, что суммы рангов PageRank превышают 1, так что они больше не представляют распределение вероятностей случайного пользователя. Но тем не менее, это хорошие оценки относительной важности страниц.

## 5.1.5. Паучьи ловушки и телепортация

Как уже было сказано, паучьей ловушкой называется множество вершин без тупиков, но и без исходящих наружу ребер. Такие структуры могут образовываться в вебе случайно или намеренно, и в результате оказывается, что PageRank вычислен только для вершин, входящих в паучьи ловушки.

**Пример 5.5.** На рис. 5.6 изображен тот же граф, что на рис. 5.1, только ребро, исходящее из вершины *C*, теперь входит в нее же. При этом *C* становится простой паучьей ловушкой из одной вершины. Отметим, что в общем случае паучьи ловушки включают много вершин, а в разделе 5.4 мы увидим, что спамеры намеренно строят паучьи ловушки, насчитывающие миллионы вершин.

Графу на рис. 5.6 соответствует такая матрица переходов:

$$M = \begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 1 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}.$$

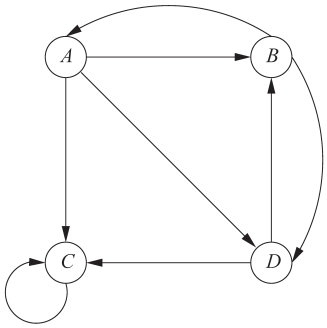



**Рис. 5.6.** Граф с паучьей ловушкой из одного узла

Применив обычную итерацию для вычисления PageRank, получим:

$$\begin{bmatrix}1/4\\1/4\\1/4\\1/4\end{bmatrix}, \begin{bmatrix}3/24\\5/24\\11/24\\5/24\end{bmatrix}, \begin{bmatrix}5/48\\7/48\\29/48\\7/48\end{bmatrix}, \begin{bmatrix}21/288\\31/288\\205/288\\31/288\end{bmatrix}, \ \ldots, \ \begin{bmatrix}0\\0\\1\\0\end{bmatrix}.$$

Как и предполагалось, PageRank отличен от 0 только для вершины *C*, потому что, попав в нее, случайный пользователь уже не сможет выбраться.

Чтобы избежать такой проблемы, мы модифицируем вычисление PageRank, разрешив любому пользователю с небольшой вероятностью телепортироваться на случайную страницу, а не идти по ссылке с текущей страницы. Тогда шаг итерации при вычислении нового вектора оценок PageRank $\mathbf{v}'$ по текущему вектору $\mathbf{v}$ и матрице переходов $M$ выглядит так:

$$\mathbf{v}' = \beta M\mathbf{v} + (1 - \beta)\mathbf{e}/n$$

где $\beta$ – константа, обычно в диапазоне от 0.8 до 0.9, $\mathbf{e}$ – вектор нужной размерности, состоящий только из единиц, а $n$ – количество вершин в графе веба. Член $\beta M\mathbf{v}$ представляет случай, когда с вероятностью $\beta$ случайный пользователь решает пойти по ссылке, исходящей с текущей страницы. Член $(1 - \beta)\mathbf{e}/n$ – вектор, все элементы которого равны $(1 - \beta)/n$, – представляет появление нового случайного пользователя на какой-то случайной странице; это происходит с вероятностью $1 - \beta$.

Отметим, что если в графе нет тупиков, то вероятность появления нового случайного пользователя в точности равна вероятности того, что случайный пользователь решит *не* следовать по ссылке с текущей страницы. В таком случае метафора пользователя, решающего, идти по ссылке или телепортироваться куда-то, выглядит разумной. Но при наличии тупиков существует и третья возможность: пользователь никуда не идет. Поскольку член $(1 - \beta)\mathbf{e}/n$ не зависит от суммы элементов вектора $\mathbf{v}$, всегда какая-то часть пользователя будет находиться в вебе. То есть, если тупики существуют, то сумма элементов $\mathbf{v}$ может быть меньше 1, но никогда не обратится в 0.

**Пример 5.6.** Посмотрим, как новый подход к вычислению PageRank работает в применении к рис. 5.6. Положим $\beta = 0.8$. Тогда итерация описывается таким уравнением:

$$v' = \begin{bmatrix} 0 & 2/5 & 0 & 0 \\ 4/15 & 0 & 0 & 2/5 \\ 4/15 & 0 & 4/5 & 2/5 \\ 4/15 & 2/5 & 0 & 0 \end{bmatrix} v + \begin{bmatrix} 1/20 \\ 1/20 \\ 1/20 \\ 1/20 \end{bmatrix}.$$

Мы включили множитель $\beta$ в матрицу $M$, умножив все ее элементы на 4/5. Все элементы вектора $(1 - \beta)\mathbf{e}/n$ равны 1/20, т. к. $1 - \beta = 1/5$ и $n = 4$. Вот результаты нескольких начальных итераций:

$$\begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}, \begin{bmatrix} 9/60 \\ 13/60 \\ 25/60 \\ 13/60 \end{bmatrix}, \begin{bmatrix} 41/300 \\ 53/300 \\ 153/300 \\ 53/300 \end{bmatrix}, \begin{bmatrix} 543/4500 \\ 707/4500 \\ 2543/4500 \\ 707/4500 \end{bmatrix}, \ \ldots, \ \begin{bmatrix} 15/148 \\ 19/148 \\ 95/148 \\ 19/148 \end{bmatrix}.$$

Будучи паучьей ловушкой, вершина $C$ собрала более половины общего ранга. Но эффект все же оказался ограниченным, и ранги всех остальных вершин тоже ненулевые.

## 5.1.6. Использование PageRank в поисковой системе

Поняв, как вычисляется вектор PageRank для той части Web, которую обошел робот поисковой системы, следует разобраться, как эта информация используется. В каждой поисковой системе имеется секретная формула для определения порядка показа результатов поиска в ответ на запрос, содержащий один или несколько поисковых термов (слов). Говорят, что Google использует свыше 250 свойств страниц, на основе которых вычисляется их линейный порядок.

Прежде всего, чтобы принять участие в ранжировании, страница должна содержать хотя бы один поисковый терм. Обычно свойствам приписываются такие веса, что у страницы, не содержащей поисковых термов, очень мало шансов оказаться в числе первых десяти результатов, который видит пользователь. Для каждой страницы, прошедшей первоначальный отбор, вычисляется оценка, важным компонентом которой является ее PageRank. В оценке учитываются и другие компоненты, например, присутствие поисковых термов в таких значимых местах, как заголовки или ссылки на страницу.

## 5.1.7. Упражнения к разделу 5.1

**Упражнение 5.1.1.** Вычислите PageRank всех страниц на рис. 5.7 в предположении, что телепортации нет.

**Упражнение 5.1.2.** Вычислите PageRank всех страниц на рис. 5.7 в предположении, что $\beta = 0.8$.

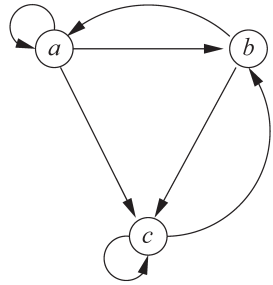



**Рис. 5.7.** Пример графа для упражнений

**! Упражнение 5.1.3.** Предположим, что веб состоит из *клики* (множество вершин, соединенных между собой всеми возможными ребрами), содержащей $n$ вершин, и еще одной дополнительной вершины, являющейся последователем для каждой из вершин клики. На рис. 5.8 показан такой граф для случая $n = 4$. Выразите PageRank каждой страницы в виде функции от $n$ и $\beta$.

**!! Упражнение 5.1.4.** Для каждого целого $n$ постройте такой веб, что в зависимости от $\beta$ любая из $n$ вершин может получить наибольший среди них PageRank. Помимо этих $n$ вершин, в вебе могут быть и другие.

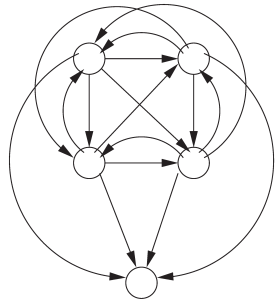


**Рис. 5.8.** Пример графа для упражнения 5.1.3

**! Упражнение 5.1.5.** Индукцией по $n$ докажите, что если второй, третий и четвертый элементы вектора $\mathbf{v}$ равны, и $M$ – матрица переходов из примера 5.1, то второй, третий и четвертый элементы будут равны также и в векторе $M^n\mathbf{v}$ для любого $n \geq 0$.

**Упражнение 5.1.6.** Предположим, что мы рекурсивно удаляем тупики из графа, разрешаем оставшийся граф и оцениваем PageRank для тупиковых страниц, как описано в разделе 5.1.4. Рассмотрим граф, представляющий собой цепочку тупиков, в начале которой находится вершина-петля, как показано на рис. 5.9. Какой PageRank получит в итоге каждая вершина?

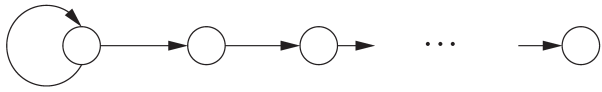


**Рис. 5.9.** Цепочка тупиков

**Упражнение 5.1.7.** Повторите упражнение 5.1.6 для дерева тупиков, показанного на рис. 5.10. Точнее, имеется одна вершина-петля, которая является корнем полного двоичного дерева с $n$ уровнями.

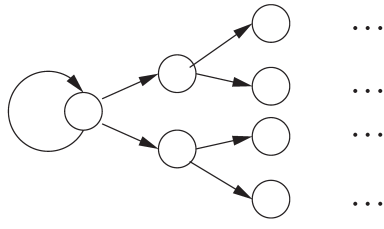


**Рис. 5.10.** Дерево тупиков

# 5.2. Эффективное вычисление PageRank

Чтобы вычислить PageRank для большого графа, представляющего веб, мы должны умножить матрицу на вектор примерно 50 раз, пока вектор не прекратит заметно изменяться после каждой итерации. В первом приближении пригодна методика MapReduce, описанная в разделе 2.3.1. Однако нужно решить две проблемы.

1. Матрица переходов в вебе $M$ сильно разрежена. Поэтому хранить все ее элементы крайне неэффективно. Вместо этого мы хотели бы представить только ненулевые элементы матрицы.
2. Возможно, мы вообще не воспользуемся MapReduce или эффективности ради захотим использовать комбинатор (см. раздел 2.2.4) совместно с распределителями, чтобы сократить объем данных, передаваемых между распределителями и редукторами. В таком случае разбиения на полосы, описанного в разделе 2.3.1, недостаточно, чтобы избежать интенсивного использования диска (пробуксовки).

В этом разделе мы обсудим, как решить обе проблемы.

## 5.2.1. Представление матрицы переходов

Матрица переходов сильно разрежена, потому что в среднем с веб-страницы исходит 10 ссылок. Если мы анализируем граф, состоящий из 10 миллиардов страниц, то лишь один из миллиарда элементов матрицы отличен от 0. Для представления любой разреженной матрицы принято использовать список ненулевых элементов вместе с их значениями. Если координаты элементов представлять 4-байтовыми целыми, а значения – 8-байтовыми числами двойной точности, то на один элемент потребуется 16 байтов. И объем памяти будет линейно зависеть от количества ненулевых элементов, а не квадратично от длины стороны матрицы.

Однако матрицу переходов в вебе можно сжать еще сильнее. Мы знаем, чему равен каждый элемент списка ненулевых элементов столбца; это 1, поделенная на количество ссылок, исходящих с данной страницы (эта величина называется *исходящей степенью* вершины). Тогда столбец можно представить целой исходящей степенью и по одному целому числу для каждого ненулевого элемента, равному номеру строки, в которой этот элемент находится. Следовательно, для представления матрицы переходов нам понадобится чуть больше 4 байтов на каждый ненулевой элемент.

**Пример 5.7.** Снова рассмотрим пример графа на рис. 5.1 со следующей матрицей переходов:

$$M = \begin{bmatrix} 0 & 1/2 & 1 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}.$$

Напомним, что строки и столбцы представляют вершины *A*, *B*, *C*, *D* именно в таком порядке. На рис. 5.11 показано компактное представление этой матрицы[5].

| Вершина | Исходящая степень | Последователи |
|---|---|---|
| *A* | 3 | *B*, *C*, *D* |
| *B* | 2 | *A*, *D* |
| *C* | 1 | *A* |
| *D* | 2 | *B*, *C* |

**Рис. 5.11.** Представление матрицы переходов исходящей степенью каждой вершины и списком ее последователей

Например, степень вершины *A* равна 3, и у нее есть три последователя. Из строки для *A* на рис. 5.11 мы можем заключить, что в столбце *M*, соответствующем *A*, будет стоять 0 в соответствующей *A* строке (поскольку ее нет в списке последователей) и 1/3 в строках, соответствующих *B*, *C* и *D*. Мы знаем, что это значение равно 1/3, потому что исходящая степень *A* равна 3, т. е. из нее выходят три ребра.

## 5.2.2. Итеративное вычисление PageRank с помощью MapReduce

На каждой итерации алгоритма PageRank берется оценка вектора PageRank $\mathbf{v}$ и вычисляется следующая оценка $\mathbf{v}'$:

$$\mathbf{v}' = \beta M\mathbf{v} + (1 - \beta)\mathbf{e}/n$$

Напомним, что $\beta$ – константа, немного меньшая 1, $\mathbf{e}$ – вектор, все элементы которого равны 1, а $n$ – количество вершин в графе, представленном матрицей $M$.

Если $n$ достаточно мало, так что каждая задача-распределитель может полностью разместить в оперативной памяти векторы $\mathbf{v}$ и $\mathbf{v}'$, то самое сложное здесь – умножение матрицы на вектор. Дополнительные шаги – умножение каждого элемента $M\mathbf{v}$ на константу $\beta$ и прибавление $(1 - \beta)/n$ к каждому элементу.

Однако, учитывая размер сегодняшнего веба, весьма вероятно, что вектор $\mathbf{v}$ не поместится в оперативной памяти. В разделе 2.3.1 мы обсуждали метод разбиения на полосы, когда $M$ разбивается на вертикальные (см. рис. 2.4), а $\mathbf{v}$ на горизонтальные полосы. Это позволит эффективно выполнить процесс MapReduce, не требуя, чтобы хотя бы в одном узле распределителя вектор $\mathbf{v}$ целиком помещался в памяти.

[5] Поскольку матрица *M* не разрежена, такое представление для нее не слишком полезно. Но этот пример иллюстрирует общий способ представления матриц, и, чем сильнее разрежена матрица, тем больше памяти удастся сэкономить.

### 5.2.3. Использование комбинаторов для консолидации результирующего вектора

Есть две причины, по которым метод из раздела 5.2.2 может не подойти.

1. Возможно, мы хотим прибавлять какие-то члены к $v'_i$, $i$-му элементу результирующего вектора **v**, в задачах-распределителях. Это улучшение эквивалентно использованию комбинатора, потому что функция Reduce просто складывает члены с общим ключом. Напомним, что при реализации умножения матрицы на вектор с помощью MapReduce ключом является значение $i$, для которого предназначен член $m_{ij}v_j$.
2. Возможно, мы вообще не хотим использовать MapReduce, а предпочитаем выполнять каждый шаг итерации на одной или нескольких машинах.

Предположим, что мы пытаемся реализовать комбинатор в сочетании с распределителем; во втором случае основная идея та же.

Пусть используется метод разбиения на полосы с тем, чтобы выделить части матрицы и векторы, помещающиеся в оперативную память. Тогда вертикальная полоса матрицы $M$ и горизонтальная полоса вектора **v** вносят вклад во все элементы результирующего вектора **v′**. Длина этого вектора такая же, как у **v**, и, значит, он тоже не поместится в память. Кроме того, из соображений эффективности матрица $M$ хранится по столбцам, а каждый столбец может повлиять на любой элемент **v′**. В результате, когда мы захотим прибавить какой-то член к некоторому элементу $v'_i$, этого элемента с большой вероятностью не окажется в оперативной памяти. И следовательно, для прибавления большинства членов придется подгружать страницу с диска. Если такая *пробуксовка* действительно происходит, то времени потребуется на несколько порядков больше, чем мы можем себе позволить.

Альтернативная стратегия – разбивать матрицу на $k^2$ блоков, а векторы – попрежнему на $k$ полос. На рис. 5.12 показано такое разбиение для $k = 4$. Мы не показываем ни умножение матрицы на $\beta$, ни прибавление $(1 - \beta)\mathbf{e}/n$, потому что эти шаги не вызывают сложностей при любой стратегии.

$$\begin{bmatrix} v'_1 \\ v'_2 \\ v'_3 \\ v'_4 \end{bmatrix} = \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix} \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \mathbf{v}_3 \\ \mathbf{v}_4 \end{bmatrix}$$

**Рис. 5.12.** Разбиение матрицы на квадратные блоки

При таком подходе используется $k^2$ задач-распределителей. Каждая задача получает один блок матрицы, например $M_{ij}$, и одну полосу вектора **v**, а именно $\mathbf{v}_j$. Отметим, что каждая полоса вектора посылается $k$ разным распределителям; $\mathbf{v}_j$ посылается задачам, обрабатывающим $M_{ij}$ для каждого из $k$ возможных значений $i$.

Следовательно, **v** передается по сети $k$ раз. Однако каждый блок матрицы передается только один раз. Поскольку можно ожидать, что размер матрицы, представленной, как описано в разделе 5.2.1, в несколько раз превышает размер вектора, то стоимость передачи не намного выше минимально возможной. А поскольку значительный объем вычислений выполняется комбинаторами, реализованными внутри распределителей, то мы экономим на передаче данных от распределителей редукторам.

Преимущество такого подхода заключается в том, что $j$-ю полосу **v** и $i$-ю полосу $\mathbf{v}'$ можно одновременно хранить в оперативной памяти во время обработки $M_{ij}$. Отметим, что все члены, порожденные из $M_{ij}$ и $\mathbf{v}_j$, дают вклад только в $v'_i$ и ни в какую другую полосу $\mathbf{v}'$.

## 5.2.4. Представление блоков матрицы переходов

Поскольку мы представляем матрицы переходов специальным образом, описанным в разделе 5.2.1, необходимо рассмотреть, как представляются блоки, изображенные на рис. 5.12. К сожалению, для хранения столбца блоков (раньше мы называли его «полосой») необходимо больше памяти, чем для полосы, не разбитой на блоки, но ненамного.

Для каждого блока нам нужны данные обо всех столбцах, для которых в блоке есть хотя бы один ненулевой элемент. Если $k$, число полос по каждому измерению, велико, то в большинстве столбцов не будет ничего в большинстве принадлежащих им блоков. Для данного блока мы должны не только перечислить строки, на пересечении которых с данным столбцом находится ненулевой элемент, но и повторно сохранить исходящую степень вершины, представленной этим столбцом. Следовательно, может оказаться, что исходящая степень будет повторена столько раз, чему она равна. Поэтому для хранения блоков полосы может потребоваться вдвое больше памяти, чем для хранения полосы, не разбитой на блоки.

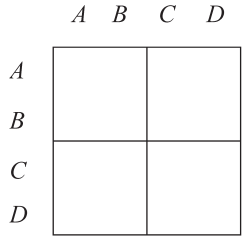



**Рис. 5.13.** Граф с четырьмя вершинами, разбитый на блоки $2 \times 2$

**Пример 5.8.** Предположим, что матрица из примера 5.7 разбита на блоки и $k = 2$. Это означает, что левый верхний квадрант представляет ссылки, ведущие из *A* или *B* в *A* или *B*, правый верхний – ссылки, ведущие из *C* или *D* в *A* или *B*, и т. д. В этом крохотном примере оказывается, что единственный элемент, которого мы можем избежать, – это элемент, соответствующий *C* в $M_{22}$, потому что из *C* не исходят ребра, ведущие в *C* или *D*.

На рис. 5.14 показаны таблицы, представляющие каждый из четырех блоков. Так, на рис. 5.14(*а*) мы видим представление левого верхнего квадранта. Отметим, что степени вершин *A* и *B* такие же, как на рис. 5.11, потому что нам нужно знать общее число последователей, а не число последователей в одном блоке. Однако каждый последователь *A* или *B* представлен лишь на одном из рисунков 5.14(*а*) или 5.14(*в*). Отметим также, что на рис. 5.14(*г*), нет строки для *C*, потому что в нижней половине матрицы (строки *C* и *D*) нет ни одного последователя *C*.

| Вершина | Степень | Последователи |
|---|---|---|
| *A* | 3 | B |
| *B* | 2 | A |

**(*а*)** Представление $M_{11}$, в котором *A* и *B* соединены с *A* и *B*

| Вершина | Степень | Последователи |
|---|---|---|
| *C* | 1 | A |
| *D* | 2 | B |

**(*б*)** Представление $M_{12}$, в котором *C* и *D* соединены с *A* и *B*

| Вершина | Степень | Последователи |
|---|---|---|
| *A* | 3 | *C*, *D* |
| *B* | 2 | D |

**(*в*)** Представление $M_{21}$, в котором *A* и *B* соединены с *C* и *D*

| Вершина | Степень | Последователи |
|---|---|---|
| *D* | 2 | C |

**(*г*)** Представление $M_{22}$, в котором *C* и *D* соединены с *C* и *D*

**Рис. 5.14.** Разреженное представление блоков матрицы

## 5.2.5. Другие эффективные подходы к итеративному вычислению PageRank

Алгоритм, описанный в разделе 5.2.3, не единственный. Мы обсудим и другие подходы, требующие меньше процессоров. У них есть общее с алгоритмом из раздела 5.2.3 полезное свойство, заключающееся в том, что матрица $M$ читается только один раз, хотя вектор $\mathbf{v}$ читается $k$ раз, где $k$ выбирается так, чтобы $1/k$-я часть векторов $\mathbf{v}$ и $\mathbf{v}'$ помещалась в оперативной памяти. Напомним, что для работы алгоритма из раздела 5.2.3 нужно было $k^2$ процессоров в предположении, что все задачи-распределители выполняются параллельно на разных процессорах.

Мы можем назначить все блоки в одной строке одному распределителю и таким образом уменьшить количество распределителей до $k$. Например, на рис. 5.12 блоки $M_{11}$, $M_{12}$, $M_{13}$ и $M_{14}$ были бы назначены одному распределителю. Если блоки

представлены, как на рис. 5.14, то мы сможем прочитать блоки из одной строки по одному за раз, так что для хранения матрицы не потребуется много оперативной памяти. Одновременно с чтением блока $M_{ij}$ мы должны прочитать полосу вектора $\mathbf{v}_j$. В результате каждый из $k$ распределителей читает весь вектор $\mathbf{v}$ и $1/k$-ю часть матрицы.

Таким образом, затраты на чтение $M$ и $\mathbf{v}$ такие же, как в алгоритме из раздела 5.2.3, но у этого подхода есть то преимущество, что каждый распределитель может скомбинировать все члены для части $v'_i$, за которую отвечает он и только он. Иными словами, редукторам остается только конкатенировать куски $\mathbf{v}'$, полученные от $k$ распределителей.

Мы можем обобщить эту идею на окружение, в котором MapReduce не используется. Пусть имеется единственный процессор, а $M$ и $\mathbf{v}$ хранятся на его локальном диске, причем $M$ – в рассмотренном выше разреженном представлении. Мы можем сначала смоделировать первый распределитель – тот, в котором используются блоки от $M_{11}$ до $M_{1k}$ и весь вектор $\mathbf{v}$ для вычисления $v'_1$. Затем моделируем второй распределитель, который читает блоки от $M_{21}$ до $M_{2k}$ и весь вектор $\mathbf{v}$ для вычисления $v'_2$. И так далее. Как и в предыдущих алгоритмах, нам придется прочитать $M$ один раз, а $\mathbf{v}$ – $k$ раз. При этом $k$ можно сделать настолько малым насколько возможно, чтобы удовлетворить ограничению: в оперативной памяти помещается $1/k$-я часть $\mathbf{v}$ и $1/k$-я часть $\mathbf{v}'$ вместе с наименьшей частью $M$, которую можно прочитать с диска (как правило, один дисковый блок).

## 5.2.6. Упражнения к разделу 5.2

**Упражнение 5.2.1.** Пусть требуется сохранить булеву матрицу $n \times n$ (содержащую только элементы 0 и 1). Можно было бы представить ее самими битами или списком позиций, в которых находятся единицы; каждая позиция описывается двумя целыми числами по $\lceil \log_2 n \rceil$ бит в каждом. Первое представление подходит для плотных матриц, второе – для разреженных. Насколько разреженной должна быть матрица (т. е. какова доля единиц), чтобы второе представление дало экономию?

**Упражнение 5.2.2.** Применяя метод, описанный в разделе 5.2.1, представьте матрицы переходов следующих графов:

(*а*) на рис. 5.4;
(*б*) на рис. 5.7.

**Упражнение 5.2.3.** Применяя метод, описанный в разделе 5.2.4, представьте матрицу переходов графа на рис. 5.2, предполагая, что сторона блока равна 2.

**Упражнение 5.2.4.** Рассмотрим граф веба в виде цепочки $n$ вершин, как на рис. 5.9. Применяя метод из раздела 5.2.4, выразите представление матрицы переходов этого графа в виде функции от $k$. Можете предполагать, что $k$ является делителем $n$.

# 5.3. Тематический PageRank

Существует несколько усовершенствований PageRank. В этом разделе мы изучим одно из них: назначение больших весов некоторым страницам в зависимости от их тематики. Для этого мы изменим поведение случайных пользователей, вынуждая их оставаться на странице, которая заведомо относится к выбранной теме. В следующем разделе мы увидим, что идею учета тематики можно применить для устранения эффекта нового вида спама, который называется «ссылочным» и разработан для обмана алгоритма PageRank.

## 5.3.1. Зачем нужен тематический PageRank

У разных людей разные интересы, но иногда разные интересы выражаются с помощью одного и того же поискового терма. Канонический пример – запрос по слову «jaguar», которое может относиться к животному, автомобилю, версии операционной системы MAC или даже к старой игровой консоли. Если поисковая система может установить, что пользователя интересуют автомобили, то она сумеет предложить ему более релевантные страницы.

В идеале у каждого пользователь должен быть его собственный вектор PageRank, который учитывает важность каждой страницы для этого пользователя. Но хранить векторы многомиллиардной длины для нескольких миллиардов пользователей невозможно, поэтому нужно что-то попроще. В тематическом алгоритме PageRank создается по одному вектору для каждой темы из небольшого набора, причем в этом векторе ранги страниц, относящихся к указанным темам, повышаются. Затем мы стараемся классифицировать пользователей по степени их интереса к выбранным темам. Конечно, при этом отчасти теряется точность, но зато мы храним для каждого пользователя короткий, а не гигантский вектор.

> **Пример 5.9.** Один полезный набор тем включает 16 категорий верхнего уровня (спорт, медицина и т. д.) из каталога Open Directory (DMOZ)[6]. Можно было бы создать 16 векторов, по одному для каждой темы. Если бы нам удалось определить, что пользователя интересует одна из этих тем – быть может, по содержимому страниц, которые он недавно просматривал, – то мы смогли бы использовать соответствующий этой теме вектор PageRank при ранжировании страниц.

## 5.3.2. Смещенное случайное блуждание

Пусть идентифицированы некоторые страницы, представляющие какую-то тему, например «спорт». Чтобы создать тематический PageRank по теме спорта, мы можем создавать случайных пользователей только на случайных страницах, посвященных спорту, а не на произвольных случайных страницах. В результате слу-

[6] Этот каталог, размещенный по адресу www.dmoz.org, представляет собой набор веб-страниц, классифицированных людьми.

чайный пользователь с высокой вероятностью окажется на одной из идентифицированных спортивных страниц или на страницах, достижимых с них по короткому пути. Интуиция подсказывает, что если с некоторой страницы ведут ссылки на спортивные страницы, то и сама эта страница посвящена спорту. И наоборот, со страницы, посвященной спорту, ссылки, скорее всего, ведут на спортивные страницы, хотя вероятность, что страница связана со спортом, уменьшается с ростом расстояния от спортивной страницы.

Математическое описание итераций, в результате которых вычисляется тематический PageRank, почти такое же, как для общего PageRank. Единственное отличие заключается в способе добавления новых пользователей. Пусть $S$ – множество целых чисел, содержащее пары строка-столбец для страниц, которые мы идентифицировали как относящиеся к определенной теме (назовем его *множеством телепортации*). Обозначим $\mathbf{e}_S$ вектор, в котором элементы, соответствующие $S$, равны 1, а все остальные – 0. Тогда *тематический PageRank для S* определяется как предел последовательности

$$\mathbf{v}' = \beta M\mathbf{v} + (1 - \beta)\,\mathbf{e}_S / |S|.$$

Здесь как обычно $M$ – матрица переходов для веба, а $|S|$ – размер множества $S$.

**Пример 5.10.** Еще раз рассмотрим первоначальный граф веба, который изображен на рис. 5.1 и повторен на рис. 5.15. Положим $\beta = 0.8$. Тогда матрица переходов для этого графа, умноженная на $\beta$, имеет вид:

$$\beta M = \begin{bmatrix} 0 & 2/5 & 4/5 & 0 \\ 4/15 & 0 & 0 & 2/5 \\ 4/15 & 0 & 0 & 2/5 \\ 4/15 & 2/5 & 0 & 0 \end{bmatrix}.$$

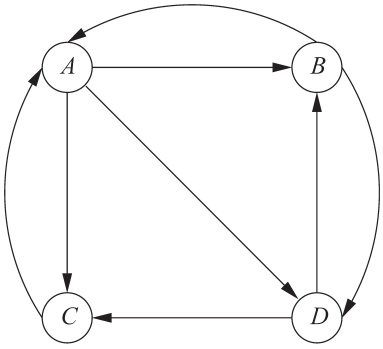



**Рис. 5.15.** Тот же пример графа веба, что и выше

Предположим, что наша тема представлена множеством телепортации $S = \{B, D\}$. Тогда второй и четвертый элемент вектора $(1 - \beta)\mathbf{e}_S/|S|$ равны 1/10, а остальные нулю. Действительно, $1 - \beta = 1/5$, размер множества $S$ равен 2, а в векторе $\mathbf{e}_S$ элементы, соответствующие $B$ и $D$, равны 1, а соот-

ветствующие $A$ и $C$ – нулю. Таким образом, уравнение, описывающее итеративный процесс, имеет вид:

$$v' = \begin{bmatrix} 0 & 2/5 & 4/5 & 0 \\ 4/15 & 0 & 0 & 2/5 \\ 4/15 & 0 & 4/5 & 2/5 \\ 4/15 & 2/5 & 0 & 0 \end{bmatrix} v + \begin{bmatrix} 0 \\ 1/10 \\ 0 \\ 1/10 \end{bmatrix}.$$

Ниже показаны результаты нескольких начальных итераций. Отметим, что пользователи начинают блуждание только со страниц, принадлежащих множеству телепортации. Хотя предел от начального распределения не зависит, при таком выборе процесс может сойтись быстрее.

$$\begin{bmatrix} 0/2 \\ 1/2 \\ 0/2 \\ 1/2 \end{bmatrix}, \begin{bmatrix} 2/10 \\ 3/10 \\ 2/10 \\ 3/10 \end{bmatrix}, \begin{bmatrix} 42/150 \\ 41/150 \\ 26/150 \\ 41/150 \end{bmatrix}, \begin{bmatrix} 62/250 \\ 71/250 \\ 46/250 \\ 71/250 \end{bmatrix}, \ldots, \begin{bmatrix} 54/210 \\ 59/210 \\ 38/210 \\ 59/210 \end{bmatrix}.$$

Отметим, что из-за концентрации пользователей в $B$ и $D$, ранг этих вершин оказывается выше, чем в примере 5.2, где наибольший PageRank был у вершины $A$.

## 5.3.3. Использование тематического PageRank

Чтобы включить тематический PageRank в поисковую систему, мы должны:

1. Решить, для каких тем создавать специализированные векторы PageRank.
2. Выбрать для каждой темы множество телепортации и использовать его для вычисления тематического вектора PageRank по этой теме.
3. Придумать, как определять тему или набор тем, наиболее релевантный данному поисковому запросу.
4. Использовать векторы PageRank для этих тем при упорядочении результатов запроса.

Один из способов выбора множества тем мы уже упомянули: рубрики верхнего уровня в каталоге Open Directory. Есть и другие подходы, но в любом случае понадобится участие человека для классификации хотя бы некоторых страниц.

Третья задача – самая трудная, и для ее решения было предложено несколько методов. Перечислим некоторые из них.

(*а*) Дать пользователю возможность выбрать тему из меню.

(*б*) Вывести темы из слов, встречающихся на тех веб-страницах, которые пользователь недавно искал, или в его недавних запросах. Как перейти от набора слов к теме, мы обсудим в разделе 5.3.4.

(*в*) Вывести темы из имеющейся информации о пользователе, например, о его закладках или интересах, указанные в его профиле в Facebook.

## 5.3.4. Вывод тем из слов

Вопрос о классификации документов по тематике изучается уже много десятков лет, и мы не будем здесь вдаваться в детали. Скажем лишь, что тема характеризуется словами, которые аномально часто встречаются в документах на эту тему. Например, слова «защитник» или «корь» нечасто встретишь в документах, посвященных вебу. Первое из них чаще среднего встречается в документах на спортивную тематику, а второе – на медицинскую.

Если подвергнуть исследованию весь веб или большую случайную выборку страниц, то можно будет получить базовые частоты слов. Затем мы можем взять большую выборку страниц, заведомо посвященных определенной теме, скажем, все страницы на тему спорта из каталога Open Directory. Вычислим частоты слов в этой выборке и найдем слова, которые появляются в спортивных страницах намного чаще, чем в общей выборке. При этом нужно избегать чрезвычайно редких слов, которые появляются в спортивной выборке с повышенной частотой. Возможно, это просто неправильно написанное слово, случайно встретившееся на одной или нескольких страницах, посвященных спорту. Поэтому слово считается характерным для данной темы, только если количество его вхождений превышает некоторый порог.

Выделив большой набор слов, которые гораздо чаще встречаются в спортивной выборке, чем в общей, и проделав то же самое для всех остальных отобранных тем, мы можем заняться классификацией других страниц по тематике. Опишем один простой подход. Пусть $S_1$, $S_2$, ..., $S_k$ – множества слов, характерных для каждой из отобранных тем, а $P$ – множество слов, встречающихся на странице $P$. Вычислим коэффициент Жаккара (см. раздел 3.1.1) для $P$ и каждого множества $S_i$. Сопоставим странице $P$ тему с максимальным коэффициентом Жаккара. Отметим, что все коэффициенты Жаккара могут оказаться очень маленькими, особенно если размеры множеств $S_i$ малы. Поэтому важно выбирать достаточно большие множества $S_i$, гарантирующие покрытие всех аспектов темы, представленной множеством.

Мы можем использовать этот метод или один из его многочисленных вариантов для классификации страниц, которые недавно были найдены по запросам данного пользователя. Можно было бы решить, что пользователя интересует тема, к которой относится большинство таких страниц. Или можно было бы смешать тематические векторы PageRank пропорционально числу страниц, отнесенных к каждой теме, и тем самым построить один вектор PageRank, отражающий различные интересы пользователей. Ту же самую процедуру можно было бы использовать и для страниц, помещенных пользователем в закладки, или даже объединить закладки с недавно просмотренными страницами.

## 5.3.5. Упражнения к разделу 5.3

**Упражнение 5.3.1.** Вычислите тематический PageRank для графа на рис. 5.15 в предположении, что множество телепортации состоит из следующих страниц:

(*а*) только $A$;
(*б*) $A$ и $C$.

# 5.4. Ссылочный спам

Когда стало очевидно, что из-за PageRank и других используемых Google приемов спам термов перестал давать эффект, спамеры начали изобретать методы, призванные обмануть алгоритм PageRank, так чтобы он придавал чрезмерную важность определенным страницам. Методы искусственного повышения PageRank страницы получили общее название «ссылочный спам». В этом разделе мы сначала поговорим о том, как спамеры создают ссылочный спам, а затем рассмотрим несколько способов противостояния этим методам, включая TrustRank и измерение спамной массы.

## 5.4.1. Архитектура спам-фермы

Спам-фермой называется набор страниц, назначение которых – увеличить PageRank одной или нескольких определенных страниц (рис. 5.16). С точки зрения спамера, веб делится на три части:

1. *Недоступные страницы*: страницы, на которые спамер повлиять не может. Это большая часть веба.
2. *Доступные страницы*: страницы, на которые спамер может повлиять, хотя напрямую их и не контролирует.
3. *Собственные страницы*: страницы, принадлежащие спамеру и контролируемые им.

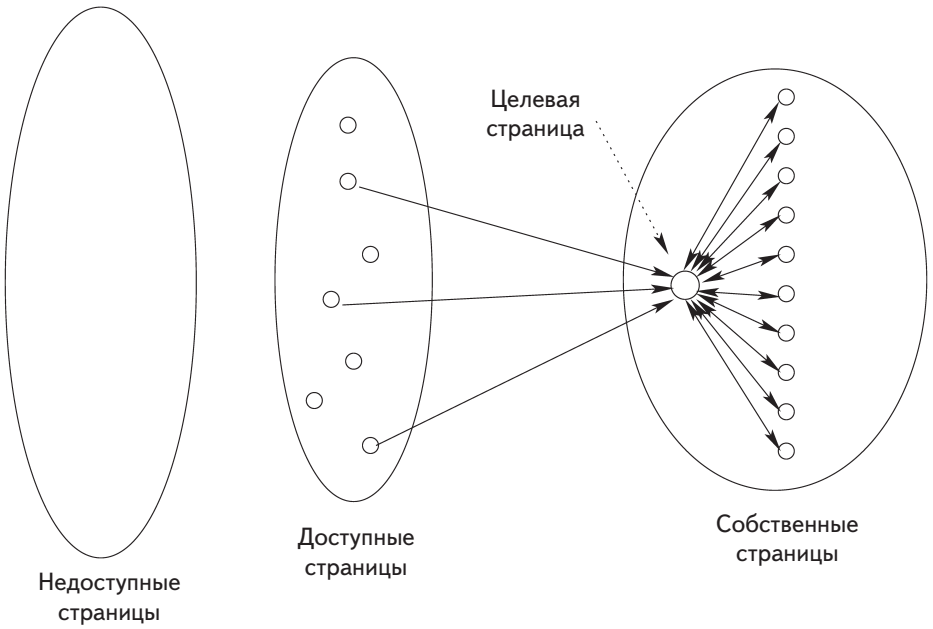



**Рис. 5.16.** Веб с точки зрения ссылочного спамера

Спам-ферма состоит из собственных страниц спамера, организованных, как показано на рисунке справа, а также некоторых ссылок с доступных страниц на спамерские. Без ссылок извне спам-ферма была бы бесполезна, потому что типичная поисковая система никогда не зашла бы на нее. По поводу доступных страниц – кажется удивительным, как можно повлиять на страницу, не владея ей. Но сегодня есть много сайтов, например блоги или газеты, которые предлагают посетителям оставлять комментарии. Чтобы организовать ссылки на собственные страницы, необходимые для повышения их ранга, спамер оставляет много комментариев такого вида: «Я согласен. Ознакомьтесь с моей статьей на сайте www.mySpamFarm.com».

В составе спам-фермы существует одна *целевая страница t*, ранг которой спамер пытается максимизировать. Кроме нее, есть *m обеспечивающих страниц* (их очень много), которые аккумулируют часть PageRank, равномерно распределенную между ними (часть $(1 - \beta)$ ранга, представляющая пользователей, телепортирующихся на случайную страницу). Заодно обеспечивающие страницы предотвращают, насколько это возможно, потерю ранга $t$ из-за телепортации на каждом раунде. Отметим, что с $t$ ведут ссылки на каждую обеспечивающую страницу, а с обеспечивающих страниц только на $t$.

## 5.4.2. Анализ спам-фермы

Предположим, что PageRank вычисляется с типичным параметром телепортации $\beta = 0.85$. Здесь $\beta$ – доля PageRank страницы, которая распределяется между ее последователями на следующем раунде. Пусть $n$ – общее число страниц в вебе и пусть некоторые из них образуют спам-ферму, показанную на рис. 5.16, с целевой страницей $t$ и $m$ обеспечивающими страницами. Обозначим $x$ часть PageRank, которую вносят доступные страницы. То есть $x$ – это сумма по всем доступным страницам $p$, ссылающимся на $t$, произведения PageRank $p$ на $\beta$, поделенная на число последователей $p$. Наконец, обозначим $y$ неизвестный PageRank страницы $t$. Нам нужно найти $y$.

Прежде всего, PageRank каждой обеспечивающей страницы равен

$$\beta y/m + (1 - \beta)/n.$$

Первый член в этой формуле представляет вклад $t$. PageRank $y$ страницы $t$ скорректирован с учетом телепортации, поэтому только часть $\beta y$ распределяется между последователями $t$, т. е. равномерно между $m$ обеспечивающими страницами. Второй член – это доля данной обеспечивающей страницы в части $(1 - \beta)$ PageRank, которая равномерно распределена между всеми страницами в вебе.

Теперь вычислим PageRank $y$ целевой страницы $t$. Он складывается из трех источников:

1. Вклад $x$ извне.
2. $\beta$, умноженное на PageRank каждой обеспечивающей страницы, т. е.

$$\beta(\beta y/m + (1 - \beta)/n).$$

3. $(1 - \beta)/n$, приходящаяся на $t$ доля части PageRank, равной $1 - \beta$. Эта величина пренебрежимо мала, поэтому отбросим ее, чтобы упростить анализ.

Складывая (1) и (2), получаем

$$y = x + \beta m\left(\frac{\beta y}{m} + \frac{1-\beta}{n}\right) = x + \beta^2 y + \beta(1-\beta)\frac{m}{n}.$$

Решаем это уравнение относительно $y$:

$$y = \frac{x}{1-\beta^2} + c\frac{m}{n},$$

где $c = \beta(1 - \beta)/(1 - \beta^2) = \beta/(1 + \beta)$.

> **Пример 5.11.** Если взять $\beta = 0.85$, то получим $1/(1 - \beta^2) = 3.6$ и $c = \beta/(1 + \beta) = 0.46$. Таким образом, описанная конструкция позволила увеличить вклад внешнего PageRank на 360 % и получить PageRank, равный 46 % от доли веба $m/n$, приходящейся на спам-ферму.

## 5.4.3. Борьба со ссылочным спамом

Теперь поисковые системы вынуждены распознавать и исключать ссылочный спам, как десять лет назад исключали спам термов. Существуют два подхода к решению этой проблемы. Первый – искать структуры, характерные для спам-ферм, когда одна страница ссылается на очень много страниц, каждая из которых ссылается обратно на нее. Поисковые системы, безусловно, ищут такие структуры и исключают соответствующие страницы из своего индекса. Это заставляет спамеров разрабатывать другие структуры с той же целью – увеличить PageRank одной или нескольких целевых страниц. Количество вариаций на тему рис. 5.16 бесконечно, поэтому война между спамерами и поисковыми системами продлится долго.

Но есть и другой подход к исключению ссылочного спама, не зависящий от поиска спам-ферм. Вместо этого поисковая система может модифицировать определение PageRank, так чтобы ранг спамных страниц автоматически понижался. Мы рассмотрим два варианта этой идеи.

1. *TrustRank*, вариант тематического PageRank, предназначенный для понижения оценки спамных страниц.
2. *Спамная масса* – вычисление, которое идентифицирует вероятные спамные страницы и дает поисковой системе возможность исключить их вовсе или сильно понизить PageRank.

## 5.4.4. TrustRank

TrustRank – это тематический PageRank, в котором под «темой» понимается множество доверенных (не спамных) страниц. В основе теории лежит предположение о том, что, хотя спамная страница может ссылаться на доверенную, маловероятно, что доверенная страница будет ссылаться на спамную. Граница проходит по сай-

там, содержащим блоги и другие возможности создавать спамные ссылки, даже если их основное содержимое достойно всяческого уважения, как в случае авторитетной газеты, разрешающей читателям оставлять комментарии.

Для реализации TrustRank мы должны разработать подходящее множество телепортации из доверенных страниц. Проверялось два подхода.

1. Поручить людям исследовать набор страниц и решить, какие из них заслуживают доверия. Например, мы можем отобрать для изучения страницы с максимальным PageRank, исходя из предположения, что ссылочный спам может поднять ранг страницы со дна до середины, но никогда не приведет ее в начало списка.
2. Выбрать домен с контролируемым членством, предполагая, что спамеру трудно внедрить свои страницы в такой домен. Например, можно выбрать домен .edu, потому что среди страниц университетов вряд ли скроется спам-ферма. Можно было бы выбрать также домены .mil или .gov. Но беда в том, что почти все сайты из этих доменов находятся в США. Чтобы получить хорошее распределение доверенных страницы, следует включить аналогичные сайты из других стран, например ac.il или edu.sg.

Вероятно, современные поисковые системы реализуют стратегию второго типа, так что PageRank в действительности является вариантом TrustRank.

## 5.4.5. Спамная масса

Идея спамной массы заключается в том, чтобы для каждой страницы измерить, какая часть ее PageRank привнесена спамом. Для этого вычислим обычный PageRank и TrustRank, основанный на множестве телепортации из доверенных страниц. Пусть страница $p$ имеет PageRank $r$ и TrustRank $t$. Тогда спамная масса $p$ равна $(r - t)/r$. Если спамная масса страницы $p$ – отрицательное или небольшое положительное число, то страница, скорее всего, не спамная. Если же спамная масса близка к 1, то это, вероятно, спам. Исключив страницы с большой спамной массой из индекса поисковой системы, мы избавимся от кучи ссылочного спама, не занимаясь поиском конкретных спам-ферм.

**Пример 5.12.** Рассмотрим PageRank и тематический PageRank, вычисленные для графа на рис. 5.1 в примерах 5.2 и 5.10 соответственно. В последнем случае множество телепортации состояло из вершин *B* и *D*, поэтому предположим, что это доверенные страницы. В таблице на рис. 5.17 сведены PageRank, TrustRank и спамная масса всех четырех вершин.

| Вершина | PageRank | TrustRank | Спамная масса |
|---|---|---|---|
| *A* | 3/9 | 54/210 | 0.229 |
| *B* | 2/9 | 59/210 | –0.264 |
| *C* | 2/9 | 38/210 | 0.186 |
| *D* | 2/9 | 59/210 | –0.264 |

**Рис. 5.17.** Вычисление спамной массы

В этом простом примере уверенно можно сделать только один вывод: вершины *B* и *D*, про которые мы заранее знали, что это не спам, имеют отрицательную спамную массу и потому спамом не являются. У двух других вершин, *A* и *C*, спамная масса положительна, поскольку их PageRank больше, чем TrustRank. Например, спамная масса *A* вычисляется путем деления разности 3/9 – 54/210 = 8/105 на PageRank 3/9, в результате чего получается 8/35, или примерно 0.229. Однако их спамная масса все же ближе к 0, чем к 1, так что это, вероятно, не спам.

### 5.4.6. Упражнения к разделу 5.4

**Упражнение 5.4.1.** В разделе 5.4.2 мы проанализировали спам-ферму, изображенную на рис. 5.16, в которой каждая обеспечивающая страница ссылается на целевую. Проведите подобный анализ для спам-фермы, в которой:

(*а*) каждая обеспечивающая страница ссылается не на целевую, а на себя;
(*б*) на обеспечивающих страницах вообще нет ссылок;
(*в*) каждая обеспечивающая страница ссылается и на целевую, и на себя.

**Упражнение 5.4.2.** Для графа веба на рис. 5.1 в предположении, что доверенной является только страница *B*:

(*а*) вычислите TrustRank каждой страницы;
(*б*) вычислите спамную массу каждой страницы.

**! Упражнение 5.4.3.** Допустим, два спамера договорились объединить свои спам-фермы. Как бы вы организовали ссылки на страницы, чтобы максимально увеличить PageRank целевых страниц каждой фермы? Есть ли выгода от объединения спам-ферм?

## 5.5. Хабы и авторитетные страницы

Идея «хабов и авторитетных страниц» была предложена вскоре после первой реализации PageRank. Алгоритм вычисления хабов и авторитетных страниц немного напоминает вычисление PageRank, поскольку сводится к итеративному вычислению неподвижной точки последовательности операций умножения матрицы на вектор. Но между ними есть и существенные различия, и обе идеи взаимно дополняют друг друга.

Алгоритм хабов и авторитетных страниц, иногда называемый HITS (hyperlink-induced topic search), первоначально задумывался не как этап предобработки, предшествующий обработке поисковых запросов – как PageRank, а как шаг, выполняемый в составе обработки запроса для ранжирования только ответов на этот запрос. Однако мы опишем его как метод анализа всего веба или той его части, которую обходит поисковый робот. Есть основания полагать, что нечто подобное действительно используется поисковой системой Ask.

## 5.5.1. Предположения, лежащие в основе HITS

Если в алгоритме PageRank понятие важности страниц одномерно, то в HITS рассматриваются два вида важности.

1. Одни страницы ценны, потому что содержат информацию по некоторой теме. Они называются *авторитетными*.
2. Другие страницы ценны, не потому что содержат информацию по какой-то теме, а потому что говорят, где найти такую информацию. Они называются *хабами*.

**Пример 5.13.** У типичного факультета университета есть веб-страница с перечнем читаемых курсов, с которой ведут ссылки на страницы с подробной информацией о каждом курсе: преподаватель, название, содержание курса и т. д. Если вы хотите что-то узнать о курсе, то заходите на его страницу; на странице факультета этих деталей нет. Страница курса является авторитетной страницей для данного курса. Но если вы хотите узнать, какие курсы предлагает факультет, то искать эту информацию на странице отдельного курса бесполезно; нужен список курсов. Это и есть страница-хаб для курсов.

В PageRank мы видели рекурсивное определение важности: страница важна, если на нее ссылаются важные страницы. А в HITS используется взаимно рекурсивное определение двух понятий: страница является хорошим хабом, если ссылается на хорошие авторитетные страницы, и страница является хорошей авторитетной страницей, если на нее ссылаются хорошие хабы.

## 5.5.2. Формализация хабов и авторитетных страниц

Для формализации интуитивных соображений припишем каждой веб-странице две оценки. Первая будет описывать *хабность* страницы – в какой мере она является хорошим хабом, вторая – *авторитетность*. В предположении, что страницы пронумерованы, мы будем представлять эти оценки векторами **h** и **a**: $i$-й элемент **h** содержит хабность, а $i$-й элемент **a** – авторитетность $i$-й страницы.

Важность распределяется между последователями страницы в соответствии с матрицей переходов для веба. А в алгоритме HITS, чтобы получить оценку хабности, обычно складывают авторитетности последователей, а чтобы получить оценку авторитетности, складывают хабности предшественников. Если бы этим все и заканчивалось, то значения хабности и авторитетности росли бы неограниченно. Поэтому обычно векторы **h** и **a** нормируются на величину наибольшего элемента. Можно вместо этого нормировать так, чтобы сумма элементов равнялась 1.

Чтобы формально описать итеративное вычисление **h** и **a**, воспользуемся *матрицей ссылок* в вебе $L$. Если имеется $n$ страниц, то $L$ – матрица размерности $n \times n$, в которой $L_{ij} = 1$, если существует ссылка со страницы $i$ на страницу $j$, и $L_{ij} = 0$ в про-

тивном случае. Нам понадобится также матрица $L^T$, полученная *транспонированием L*. Это означает, что $L^T_{ij} = 1$, если существует ссылка со страницы *j* на страницу *i*, иначе $L^T_{ij} = 0$. Отметим, что $L^T$ похожа на матрицу *M*, используемую в PageRank, но в тех позициях, где $L^T$ содержит 1, *M* содержит дробь – 1, деленная на количество исходящих ссылок со страницы, представленной данным столбцом.

**Пример 5.14.** Рассмотрим веб, изображенный на рис. 5.4 и воспроизведенный еще раз на рис. 5.18. Сделаем важное наблюдение: тупики и паучьи ловушки не мешают сходимости итеративного вычисления HITS к паре векторов. Следовательно, мы можем работать с рис. 5.18 напрямую; телепортация и изменение графа не нужны. Матрица ссылок *L* и результат ее транспонирования показаны на рис. 5.19.

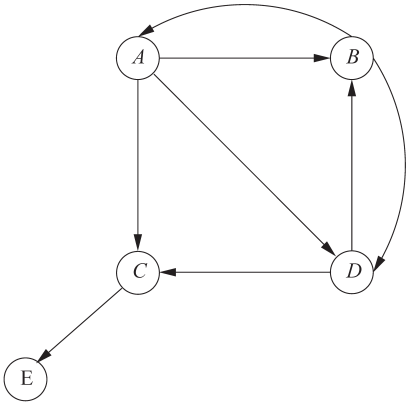



**Рис. 5.18.** Данные для примеров HITS

$$L = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad L^T = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

**Рис. 5.19.** Матрица ссылок для веб, изображенного на рис. 5.18, и транспонированная к ней

Тот факт, что хабность страницы пропорциональна сумме авторитетностей ее последователей, описывается выражением $\mathbf{h} = \lambda L\mathbf{a}$, где $\lambda$ – неизвестная константа, представляющая нормировочный коэффициент. Аналогично тот факт, что авторитетность страницы пропорциональна сумме хабностей ее предшественников, описывается выражением $\mathbf{a} = \mu L^T\mathbf{h}$, где $\mu$ – другой нормировочный коэффициент. Эти выражения позволяют вычислять хабность и авторитетность независимо, подставляя одно выражение в другое:

- $\mathbf{h} = \lambda\mu LL^T\mathbf{h}$.
- $\mathbf{a} = \lambda\mu L^TL\mathbf{a}$.

Но поскольку $LL^T$ и $L^TL$ разрежены не так сильно, как *L* и $L^T$, вычислять **h** и **a** обычно лучше путем взаимной рекурсии. Пусть начальное значение **h** – вектор, все элементы которого равны 1.

1. Вычислим $\mathbf{a} = L^T\mathbf{h}$ и нормируем результат, так чтобы наибольшая компонента равнялась 1.
2. Затем вычислим $\mathbf{h} = L\mathbf{a}$ и снова нормируем.

Теперь, имея новое значение **h**, мы можем повторить шаги (1) и (2) и продолжать так до тех пор, пока изменения обоих векторов между соседними итерациями не станут достаточно малы. Получившиеся значения считаем аппроксимацией предельных.

**Пример 5.15.** Выполним первые две итерации алгоритма HITS для графа веба, изображенного на рис. 5.18. На рис. 5.20 показан последовательность вычисленных векторов. Первый столбец – начальное значение **h**, в котором все элементы равны 1. Второй столбец – вычисленная оценка относительной авторитетности $L^{\mathrm{T}}\mathbf{h}$, где каждой странице сопоставляется сумма хабностей ей предшественников. Третий столбец – первая оценка **a**, вычисленная путем нормировки второго столбца; в данном случае мы разделили каждый элемент на 2 – наибольшее значение во втором столбце.

$$\begin{bmatrix}1\\1\\1\\1\\1\end{bmatrix} \quad \begin{bmatrix}1\\2\\2\\2\\1\end{bmatrix} \quad \begin{bmatrix}1/2\\1\\1\\1\\1/2\end{bmatrix} \quad \begin{bmatrix}3\\3/2\\1/2\\2\\0\end{bmatrix} \quad \begin{bmatrix}1\\1/2\\1/6\\2/3\\0\end{bmatrix}$$

$h \qquad L^Th \qquad a \qquad La \qquad h$

$$\begin{bmatrix}1/2\\5/3\\5/3\\3/2\\1/6\end{bmatrix} \quad \begin{bmatrix}3/10\\1\\1\\9/10\\1/10\end{bmatrix} \quad \begin{bmatrix}29/10\\6/5\\1/10\\2\\0\end{bmatrix} \quad \begin{bmatrix}1\\12/29\\1/29\\20/29\\0\end{bmatrix}$$

$L^Th \qquad a \qquad La \qquad h$

**Рис. 5.20.** Первые две итерации алгоритма HITS

В четвертом столбце показан вектор $L\mathbf{a}$. То есть мы вычисляем оценку хабности каждой страницы как сумму оценок авторитетностей ее последователей. Пятый столбец – результат нормировки четвертого. В данном случае мы делим на 3, потому что это наибольшее значение в четвертом столбце. В столбцах с шестого по девятый повторен тот же процесс, но уже начиная с лучшей оценки хабности, взятой из пятого столбца.

Предел, к которому сходится этот процесс, не очевиден, но его можно вычислить с помощью простой программы. Вот какие она дает результаты:

$$h = \begin{bmatrix}1\\0.3583\\0\\0.7165\\0\end{bmatrix} \qquad a = \begin{bmatrix}0.2087\\1\\1\\0.7913\\0\end{bmatrix}$$

Эти результаты вполне осмысленны. Заметим для начала, что хабность $E$, безусловно равна 0, потому что с этой страницы нет исходящих ссылок. Хабность $C$ зависит только от авторитетности $E$ и наоборот, поэтому неудивительно, что обе равны 0. $A$ – самый крупный хаб, потому что ведет на три авторитетные страницы, $B$, $C$ и $D$. В свою очередь, $B$ и $C$ – самые авторитетные страницы, потому что на них ссылаются два крупнейших хаба, $A$ и $D$.

Для графов размером с веб единственный способ найти решение уравнений хабности-авторитетности – последовательные приближения. Но в нашем крохотном примере их можно решить и непосредственно. Будем использовать уравнение $\mathbf{h} = \lambda\mu LL^{\mathrm{T}}\mathbf{h}$. Сначала вычислим $LL^{\mathrm{T}}$:

$$LL^T = \begin{bmatrix} 3 & 1 & 0 & 2 & 0 \\ 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Положим $\upsilon = 1/(\lambda\mu)$ и обозначим элементы вектора $\mathbf{h}$, соответствующим вершинам от $A$ до $E$, буквами от $a$ до $e$. Тогда уравнения для $\mathbf{h}$ можно записать в виде:

$$\upsilon a = 3a + b + 2d \qquad \upsilon b = a + 2b$$
$$\upsilon c = c \qquad \upsilon d = 2a + 2d$$
$$\upsilon e = 0$$

Из уравнения для $b$ получаем $b = a/(\upsilon - 2)$, а из уравнения для $d$ – $d = 2a/(\upsilon - 2)$. Подставляя эти выражения в уравнения для $a$, получаем $\upsilon a = a(3+5/(\upsilon-2))$. Сокращая $a$, получаем квадратное уравнение относительно $\upsilon$: $\upsilon^2 - 5\upsilon + 1 = 0$. Его положительный корень равен $\upsilon = (5 + \sqrt{21})/2 = 4.791$. Зная, что $\upsilon$ отлично от 0 и 1, мы из оставшихся уравнений находим, что $c = e = 0$.

Наконец, осознав, что $a$ – самый большой элемент $\mathbf{h}$, и полагая $a = 1$, мы получаем $b = 0.3583$ и $d = 0.7165$. Вместе с решениями $c = e = 0$ это дает предельные значения $\mathbf{h}$. Значение $\mathbf{a}$ можно вычислить, умножив $\mathbf{h}$ на $L^{\mathrm{T}}$ и разделив на нормировочный коэффициент 2.

### 5.5.3. Упражнения к разделу 5.5

**Упражнение 5.5.1.** Вычислите хабность и авторитетность каждой вершины графа на рис. 5.1.

**! Упражнение 5.5.2.** Пусть граф представляет собой цепочку $n$ вершин, как на рис. 5.9. Выразите векторы хабности и авторитетности в виде функции от $n$.

## 5.6. Резюме

- *Спам термов.* Ранние поисковые системы не могли вернуть релевантные результаты, потому что были уязвимы для спама термов – включения в веб-страницу слов, искажающих ее истинное содержимое.

- *Решение проблемы спама термов, предложенное Google.* Компания Google сумела побороть спам термов, реализовав две идеи. Первая – алгоритм PageRank определения относительной важности веб-страниц. Вторая – доверие не тому, что страница говорит о себе, а тому, что говорят о ней другие страницы в ссылках или рядом с ними.
- *PageRank.* Алгоритм PageRank сопоставляет каждой веб-странице вещественное число, называемое ее рангом PageRank. PageRank страницы – это мера ее важности или релевантности поисковому запросу. В простейшем виде PageRank находится как решение рекурсивного уравнения «страница важна, если важны ссылающиеся на нее страницы».
- *Матрица переходов для веба.* Мы представляем ссылки в вебе матрицей, в которой $i$-я строка и $i$-й столбец представляют $i$-ю страницу веба. Если со страницы $j$ на страницу $i$ ведет хотя бы одна ссылка, то элемент на пересечении строки $i$ и столбца $j$ равен $1/k$, где $k$ – число страниц, на которые ссылается страница $j$. Все остальные элементы матрицы переходов равны 0.
- *Вычисление PageRank для сильно связных графов веба.* Если граф веба сильно связный (т. е. из любой вершины можно достичь любой другой), то PageRank является главным собственным вектором матрицы переходов. Для вычисления PageRank мы можем начать с произвольного ненулевого вектора и повторно умножать его на матрицу перехода, получая улучшающиеся оценки[7]. Примерно после 50 итераций оценка окажется очень близка к пределу – истинному значению PageRank.
- *Модель случайного пользователя.* Вычисление PageRank можно интерпретировать как моделирование поведения многих случайных пользователей, каждый из которых начинает блуждание на случайной странице и на каждом шаге случайно выбирает одну из исходящих ссылок. Вероятность, с которой пользователь окажется на данной странице, и есть ее PageRank. Интуитивно понятно, что авторы страницы создают ссылки на страницы, которые считают полезными, так что случайные пользователи стремятся к полезной странице.
- *Тупики.* Тупиком называется веб-страница, на которой нет ни одной ссылки. Из-за наличия тупиков PageRank некоторых или всех страниц обращается в 0 при итеративном вычислении, и это может случиться даже для страниц, не являющихся тупиковыми. Мы можем исключить все тупики до начала вычисления PageRank, рекурсивно удалив вершины, из которых не исходит ни одно ребро. Отметим, что удаление вершины может привести к тому, что другая вершина, которая ссылалась только на удаленную, станет тупиком. Поэтому процесс по необходимости должен быть рекурсивным.
- *Паучьи ловушки.* Паучьей ловушкой называется множество вершин, которые могут ссылаться друг на друга, но не наружу. При итеративном вы-

[7] Строго говоря, чтобы этот метод работал, одной сильной связности недостаточно. Однако другие необходимые условия выполняются для любой крупной сильно связной компоненты веба, если только она не была построена искусственно.

числении PageRank наличие паучьей ловушки приводит к тому, что весь PageRank распределяется между входящими в нее узлами.

- *Схемы с телепортацией*. Для борьбы с паучьими ловушками (и с тупиками, если мы их не исключаем) вычисление PageRank обычно не сводится к последовательному умножению на матрицу переходов. Выбирается параметр β, как правило, равный примерно 0.85. Текущая оценка PageRank умножается на матрицу переходов и на β, а затем к оценке для каждой страницы прибавляется $(1 - \beta)/n$, где $n$ – общее число страниц
- *Телепортация и случайные пользователи*. Вычисление PageRank с применением параметра β можно интерпретировать следующим образом: каждый случайный пользователь с вероятностью $1 - \beta$ покидает веб, а вместо него на случайной странице появляется новый пользователь.
- *Эффективное представление матрицы переходов*. Поскольку матрица переходов очень сильно разрежена (почти все ее элементы равны 0), можно сэкономить время и память, если представлять ее списком ненулевых элементов. Однако у ненулевых элементов есть и еще одно свойство: в каждом столбце они одинаковы – значение любого ненулевого элемента равно единице, поделенной на количество ненулевых элементов в данном столбце. Поэтому лучше хранить матрицу по столбцам и представлять каждый столбец числом ненулевых элементов и списком номеров строк, в которых эти элементы находятся.
- *Умножение очень большой матрицы на вектор*. Для графов размером с веб сохранить всю оценку вектора PageRank в оперативной памяти одного компьютера вряд ли возможно. Но мы можем разбить вектор на $k$ частей, а матрицу переходов на $k^2$ квадратных блоков и назначить каждый блок отдельной машине. Части векторов рассылаются $k$ машинам, поэтому накладные расходы на репликацию вектора невелики.
- *Представление блоков матрицы переходов*. При разбиении матрицы переходов на квадратные блоки ее столбцы разбиваются на $k$ сегментов. Если в некотором сегменте нет ненулевых элементов, то для его представления делать ничего не нужно. Но если есть хотя бы один ненулевой элемент, то сегмент следует представить общим числом ненулевых элементов во всем столбце (чтобы знать, чему равен каждый элемент) и списком строк, содержащих ненулевые элементы.
- *Тематический PageRank*. Если мы знаем, что человек, отправивший запрос, интересуется конкретной темой, то имеет смысл повысить PageRank страницам, относящимся к этой теме. Для вычисления такой формы PageRank мы определяем множество страниц, заведомо посвященных данной теме, и используем его как «множество телепортации». Вычисление PageRank модифицируется таким образом, чтобы добавочное слагаемое прибавлялось только к страницам из множества телепортации, а не распределялось между всеми страницами в вебе.

- *Создание множеств телепортации*. Чтобы тематический PageRank мог работать, мы должны идентифицировать страницы, с большей вероятностью относящиеся к данной теме. Один из возможных подходов – начать со страниц, отнесенных к соответствующей рубрике в каталоге Open Directory (DMOZ). Другой подход – определить, какие слова заведомо ассоциированы с данной темой, и взять в качестве множества телепортации страницы, на которых эти слова встречаются аномально часто.
- *Ссылочный спам*. Чтобы обмануть алгоритм PageRank, бессовестные личности создали спам-фермы. Это наборы страниц, задача которых – придать как можно больший PageRank некоторой целевой странице.
- *Структура спам-фермы*. Типичная спам-ферма состоит из одной целевой и большого количества обеспечивающих страниц. Целевая страница ссылается на каждую из обеспечивающих, а обеспечивающие – только на целевую. Кроме того, необходимо создать хотя бы несколько ссылок извне фермы. Например, спамер может разместить ссылки на свою целевую страницу, оставляя комментарии в чужих блогах или дискуссионных группах.
- *TrustRank*. Один из способов свести на нет эффект ссылочного спама – вычислить вариант тематического PageRank, называемый TrustRank, для которого множество телепортации состоит из доверенных страниц. Например, это может быть набор страниц университетских сайтов. При этом добавочное слагаемое в вычислении PageRank не делится с многочисленными обеспечивающими страницам, так что их PageRank снижается.
- *Спамная масса*. Чтобы найти спам-фермы, мы можем вычислить для каждой страницы PageRank и TrustRank. Те страницы, для которых TrustRank много меньше PageRank, с большой вероятностью принадлежат спам-ферме.
- *Хабы и авторитетные страницы*. В алгоритме PageRank важность – одномерное понятие. А в алгоритме HITS производится попытка измерить два разных аспекта важности. Авторитетными называются страницы, содержащие ценную информацию, а хабами – страницы, которые сами по себе не содержат информации, зато ссылаются на страницы, где ее можно найти.
- *Рекурсивная формулировка алгоритма HITS*. Для вычисления оценок хабности и авторитетности страницы необходимо решить два рекурсивных уравнения: «хаб ссылается на много авторитетных страниц, а на авторитетную страницу ссылается много хабов». Их решение сводится к повторному умножению матрицы на вектор, как и при вычислении PageRank. Однако в отличие от PageRank существование тупиков или паучьих ловушек не мешает решению уравнений HITS, так что никакие схемы с добавочными слагаемыми не нужны.

# 5.7. Список литературы

Алгоритм PageRank впервые описан в работе [1]. Работа [2] – экспериментальное изучение структуры веба, выявившее существование тупиков и паучьих ловушек. Применение метода разбиения на полосы для итеративного вычисления PageRank заимствовано из [5].

Тематический PageRank описан в работе [6], TrustRank –в работе [4], а идея спамной массы взята из [3].

Идея алгоритма HITS (хабы и авторитетные страницы) впервые сформулирована в работе [7].

1. S. Brin, L. Page «Anatomy of a large-scale hypertextual web search engine», Proc. 7th Intl. World-Wide-Web Conference, pp. 107–117, 1998.
2. A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, J. Weiner «Graph structure in the web», Computer Networks 33:1–6, pp. 309–320, 2000.
3. Z. Gyöngi, P. Berkhin, H. Garcia-Molina, J. Pedersen «Link spam detection based on mass estimation», Proc. 32nd Intl. Conf. on Very Large Databases, pp. 439–450, 2006.
4. Z. Gyöngi, H. Garcia-Molina, J. Pedersen «Combating link spam with trustrank», Proc. 30th Intl. Conf. on Very Large Databases, pp. 576–587, 2004.
5. T. H. Haveliwala «Efficient computation of PageRank», Stanford Univ. Dept. of Computer Science technical report, Sept., 1999. Available as `http://infolab.stanford.edu/~taherh/papers/efficient-pr.pdf`
6. T. H. Haveliwala «Topic-sensitive PageRank», Proc. 11th Intl. World-Wide-Web Conference, pp. 517–526, 2002
7. J. M. Kleinberg «Authoritative sources in a hyperlinked environment», J. ACM 46:5, pp. 604–632, 1999.

# ГЛАВА 6.
# Частые предметные наборы

В этой главе мы обратимся к одному семейству методов, позволяющих охарактеризовать данные: обнаружению частых предметных наборов. Эту задачу часто рассматривают как поиск «ассоциативных правил», хотя последний предполагает более полную характеристику данных, выявление которых принципиально зависит от обнаружения частых предметных наборов.

Мы начнем с рассмотрения модели данных на основе «корзины покупок», которая по существу представляет собой связь многие-ко-многим между элементами двух видов: «предметами» и «корзинами», дополненную некоторыми предположениями о форме данных. Задача о частых предметных наборах заключается в нахождении множеств предметов, которые встречаются вместе во многих корзинах.

Эта задача отличается от поиска похожих множеств, рассматривавшегося в главе 3. Здесь нас интересует абсолютное число корзин, содержащих конкретный набор предметов, тогда как в главе 3 мы искали предметы, для которых содержащие их корзины сильно пересекаются, пусть даже абсолютное число таких корзин мало.

Это различие приводит к новому классу алгоритмов для нахождения частых предметных наборов. Мы начнем с алгоритма Apriori, который отсеивает большинство крупных наборов из числа кандидатов, поскольку сначала ищет мелкие наборы, а крупный набор не может быть частым, если таковыми не являются все его поднаборы. Затем мы рассмотрим различные усовершенствования базового алгоритма Apriori, акцентируя внимание на очень больших наборах данных, которые не помещаются в оперативную память.

Далее мы перейдем к приближенным алгоритмам, которые работают быстрее, но не гарантируют нахождения всех частых предметных наборов. В этот класс попадают также алгоритмы, которые допускают распараллеливание средствами MapReduce. Наконец, мы вкратце обсудим поиск частых предметных наборов в потоке данных.

## 6.1. Модель корзины покупок

Модель корзины покупок применяется для описания распространенной формы связи многие-ко-многим между объектами двух видов. С одной стороны, имеются

предметы, а, с другой, корзины, которые иногда называют «транзакциями». Каждая корзина содержит набор предметов, и обычно предполагается, что число предметов в корзине мало – гораздо меньше общего числа предметов. Обычно считается, что число корзин очень велико – больше, чем помещается в оперативной памяти. Предполагается, что данные хранятся в файле, где представлены в виде последовательности корзин. В терминах распределенной файловой системы, описанной в разделе 2.1, корзины – это объекты в файле, а каждая корзина имеет тип «множество предметов».

## 6.1.1. Определение частого предметного набора

На интуитивном уровне «частым» называется набор предметов, встречающийся во многих корзинах. Формально предположим, что задано число *s*, называемое *пороговой поддержкой*. Если *I* – набор предметов, то *поддержкой I* называется количество корзин, для которых *I* является поднабором. Набор *I* называется *частым*, если его поддержка не меньше *s*.

**Пример 6.1.** На рис. 6.1 приведены наборы слов. Каждый набор – это корзина, а слова – предметы. Чтобы получить эти наборы, мы произвели в Google поиск по фразе `cat dog` и взяли фрагменты их страниц с наибольшим рангом. Пусть вас не смущает, что некоторые слова встречаются в корзине дважды, несмотря на то, что корзина является множеством и, по идее, не должна содержать дубликатов. Также не обращайте внимания на регистр букв.

1. **{Cat, and, dog, bites}**
2. **{Yahoo, news, claims, a, cat, mated, with, a, dog, and, produced, viable, offspring}**
3. **{Cat, killer, likely, is, a, big, dog}**
4. **{Professional, free, advice, on, dog, training, puppy, training}**
5. **{Cat, and, kitten, training, and, behavior}**
6. **{Dog, &, Cat, provides, dog, training, in, Eugene, Oregon}**
7. **{«Dog, and, cat», is, a, slang, term, used, by, police, officers, for, a, male–female, relationship}**
8. **{Shop, for, your, show, dog, grooming, and, pet, supplies}**

**Рис. 6.1.** Восемь корзин, содержащих слова

Поскольку пустое множество является подмножеством любого множества, поддержка ∅ равна 8. Однако, как правило, пустое множество мы рассматривать не будем, т. к. оно ни о чем не говорит.

Из одноэлементных наборов наиболее частыми, очевидно, являются {cat} и {dog}. Слово «dog» встречается во всех корзинах, кроме (5), поэтому его поддержка равна 7, а слово «cat» во всех, кроме (4) и (8), так что имеет поддержку 6. Слово «and» тоже встречается часто – в корзинах (1), (2), (5), (7) и (8) – и, следовательно, имеет поддержку 5. Слова «a» и «training» встречаются в трех наборах, а слова «for» и «is» – в двух. Все остальные слова встречаются по одному разу.

Установим пороговую поддержку $s$ = 3. Тогда частых одноэлементных предметных наборов будет пять: {dog}, {cat}, {and}, {a}, {training}.

Теперь займемся двухэлементными наборами. Двухэлементный набор не может быть частым, если каждый его элемент по отдельности не является частым набором. Следовательно, существует только десять возможных двухэлементных наборов. На рис. 6.2 показано, какие корзины содержат эти двухэлементные наборы.

| | **training** | **a** | **and** | **cat** |
|---|---|---|---|---|
| **dog** | 4, 6 | 2, 3, 7 | 1, 2, 8 | 1, 2, 3, 6, 7 |
| **cat** | 5, 6 | 2, 3, 7 | 1, 2, 5 | |
| **and** | 5 | 2, 7 | | |
| **a** | нет | | | |

**Рис. 6.2.** Вхождения двухэлементных наборов

Например, мы видим, что набор {dog, training} встречается только в корзинах (4) и (6). Следовательно, его поддержка равна 2, т. е. он не частый. Для $s$ = 3 существует пять двухэлементных наборов:

{dog, a} {dog, and} {dog, cat}
{cat, a} {cat, and}

Каждый встречается по три раза, за исключением набора {dog, cat}, который встречается пять раз.

Посмотрим, существуют ли частые трехэлементные наборы. Трехэлементный набор может быть частым, только если каждый его двухэлементный поднабор является частым. Например, набор {dog, a, and} не может быть частым, потому что тогда частым должен был бы быть набор {a, and}, а это не так. Набор {dog, cat, and} мог бы быть частым, потому что все его двухэлементные поднаборы частые. Но увы, все три слова встречаются вместе только в корзинах (1) и (2), поэтому частым этот набор не является. Набор {dog, cat, a} тоже мог бы быть частым, поскольку все его двухэлементные поднаборы частые. Как легко видеть, все три слова встречаются в корзинах (2), (3) и (7), поэтому этот набор действительно является частым. Больше никаких троек-кандидатов нет, поскольку не существует других троек, для которых все двухэлементные поднаборы частые. Коль скоро нашелся только один частый набор из трех предметов, то частых наборов из четырех и более предметов быть не может.

## 6.1.2. Применения частых предметных наборов

Первым применением модели корзины покупок стал анализ настоящих корзин в реальных магазинах. Супермаркеты и сетевые магазины хранят содержимое каждой корзины покупок (физической тележки), прошедшей через кассу. «Предметами» здесь являются продаваемые товары, а «корзинами» – наборы предметов в одной тележке. Крупная торговая сеть может предлагать 100 000 различных товаров и собирать данные о миллионах корзин.

### Онлайновая и реальная торговля

В разделе 3.1.3 мы предположили, что розничные интернет-магазины могли бы использовать меры сходства для поиска пар предметов, для которых велика доля покупателей, приобретающих их одновременно, пусть даже в абсолютных цифрах таких покупателей не очень много. После этого магазин мог бы показать рекламу одного предмета из пары тем немногим покупателям, которые купили другой предмет. Такая тактика бессмысленна в реальных магазинах, потому что если товар покупает немного людей, то рекламировать его продажи было бы неэффективно. Поэтому методы, описанные в главе 3, для реальных магазинов не слишком полезны.

И наоборот, интернет-магазину неинтересны аналитические методы, обсуждаемые в этой главе, потому что они предназначены для поиска часто встречающихся предметных наборов. Если бы интернет-магазин ограничивался только частыми наборами, то упустил бы таящиеся в «длинном хвосте» возможности избирательной рекламы, адаптированной к конкретному покупателю.

Обнаружив частые предметные наборы, магазин может узнать, какие товары часто покупают вместе. Особенно важны наборы из двух и более товаров, которые встречаются вместе чаще, чем можно было бы ожидать, если бы товары приобретались независимо. Этот аспект проблемы мы обсудим в разделе 6.1.3, а пока просто займемся поиском частых предметных наборов. В ходе такого анализа мы обнаружим, что многие люди покупают вместе хлеб и масло, но это малоинтересно, потому что мы и так знаем, что эти товары были бы популярны и по отдельности. Можно обнаружить, что часто покупают вместе сосиски и горчицу. Это не сюрприз для любителя сосисок, но открывает перед супермаркетом возможность нетривиального маркетинга. Он может провести акцию по продаже сосисок и поднять цену на горчицу. Придя в магазин за дешевыми сосисками, человек вспомнит о горчице и купит ее тоже. При этом он либо не заметит, что цена горчицы поднялась, либо сочтет, что не имеет смысла идти куда-то еще за горчицей подешевле.

Хорошо известен пример такого рода с «подгузниками и пивом». Трудно заранее предположить, что эти товары как-то связаны, но анализ данных в одной торговой сети показал, что покупающие подгузники аномально часто покупают и пиво. Была высказана гипотеза, что раз вы покупаете подгузники, то дома, наверное, есть маленький ребенок, а тогда вы вряд ли пойдете в бар и, следовательно, скорее купите пиво домой. Тот же маркетинговый ход, который мы описали для сосисок и горчицы, можно было бы применить и в данном случае.

Однако приложения частых предметных наборов не ограничиваются корзинами покупок. Эта модель применима и ко многим другим видам добычи данных. Приведем несколько примеров.

1. *Взаимосвязанные концепции*. Пусть предметами являются слова, а корзинами – документы (например, веб-страницы, блоги или сообщения в Твиттере). Корзина-документ содержит предметы-слова, присутствующие в документе. Если поискать наборы слов, присутствующие совместно во многих документах, то обнаружится, что в них превалируют наиболее употребительные слова (стоп-слова). Мы видели это в примере 6.1, где нашей целью были фрагменты, посвященные собакам и кошкам, но оказалось, что в частых предметных наборах чаще всего встречаются стоп-слова «and» и «a». Но если игнорировать общеупотребительные слова, то можно надеяться, что среди частых пар мы найдем пары слов, представляющие общую концепцию. Например, можно ожидать, что пара {Brad, Angelina} будет встречаться с аномальной частотой[1].
2. *Обнаружение плагиата*. Пусть предметами будут документы, а корзинами –предложения. Предмет-документ находится в корзине-предложении, если предложение встречается в документе. На первый взгляд, тут все поставлено с ног на голову, однако это именно то, что нам нужно. Не забывайте, что между предметами и корзинами может существовать произвольная связь многие-ко-многим. То есть предлог «в» не обязательно должен иметь обычный смысл – «часть чего-то». В данном приложении мы ищем пары предметов, которые встречаются совместно в нескольких корзинах. Найденная пара означает, что имеются два документа, в которых несколько предложений совпадают. На практике даже одно или два совпадающих предложения могут быть признаками плагиата.
3. *Биомаркеры*. Рассмотрим предметы двух типов: биомаркеры, например гены или белки крови, и болезни. Корзиной будем считать набор данных о пациенте: геном и химический анализ крови, а также историю болезни. Тогда частый предметный набор, состоящий из одной болезни и одного или нескольких биомаркеров, может рассматриваться как тест для выявления болезни.

## 6.1.3. Ассоциативные правила

Хотя тема настоящей главы – поиск частых предметных наборов в данных, эту информацию часто представляют в виде коллекции так называемых ассоциативных правил вида «если–то». Ассоциативное правило записывается в виде $I \to j$, где $I$ – набор предметов, а $j$ – предмет. Импликация означает, что если все предметы из $I$ встречаются в некоторой корзине, то «вероятно» $j$ также встретится в ней.

Формализуем понятие «вероятно», определив *достоверность* правила $I \to j$ как отношение поддержки $I \cup \{j\}$ к поддержке $I$. То есть достоверность правила – это доля тех корзин, содержащих все предметы из $I$, в которых встречается также $j$.

[1] Намек на то, что Анджелина Джоли и Брэд Питт грызутся, как кошка с собакой. – *Прим. перев.*

**Пример 6.2.** Рассмотрим корзины на рис. 6.1. Достоверность правила {*cat*, *dog*} → *and* равна 3/5. Слова «cat» и «dog» встречаются в пяти корзинах: (1), (2), (3), (6) и (7). Из них слово «and» содержат только корзины (1), (2), (7) т. е. 3/5.

Другой пример: достоверность правила {*cat*} → *kitten* равна 1/6. Слово «cat» встречается в шести корзинах: (1), (2), (3), (5), (6) и (7). Из них только (5) содержит слово «kitten».

Достоверность сама по себе может быть полезной, если поддержка левой части правила достаточно велика. Например, нам необязательно знать, что люди, покупающие сосиски, необычно часто покупают также горчицу, если мы знаем, что многие покупают сосиски и многие покупают сосиски и горчицу. Мы и так можем проделать трюк с распродажей сосисок, описанный в разделе 6.1.2. Однако часто ценность ассоциативного правила возрастает, если оно отражает действительно существующую связь, когда предмет или предметы в левой части каким-то образом влияют на предмет в правой части.

Поэтому мы определяем *интересность* ассоциативного правила $I \rightarrow j$ как разность между его достоверностью и долей корзин, содержащих $j$. Иными словами, если $I$ не оказывает никакого влияния на $j$, то следует ожидать, что доля корзин, содержащих $j$, среди корзин, которые содержат $I$, будет в точности такой же, как доля среди всех вообще корзин. Интересность такого правила равна 0. Однако правило интересно – и неформально, и строго – если его интересность высока, т. е. присутствие $I$ в корзине каким-то образом влечет и присутствие $j$, или резко отрицательна, т. е. присутствие $I$ несовместимо с присутствием $j$.

**Пример 6.3.** История с пивом и подгузниками в действительности означает, что интересность ассоциативного правила {*подгузники*} → *пиво* высока. То есть доля покупателей пива среди покупателей подгузников, существенно выше, чем доля покупателей пива среди всех покупателей. Пример правила с отрицательной интересностью: {*кока*} → *пепси*. То есть люди, покупающие кока-колу, скорее всего, не покупают еще и пепси, хотя доля всех покупателей пепси велика; обычно люди предпочитают один из этих напитков, но не оба сразу. Точно так же можно ожидать, что интересность правила {*пепси*} → *кока* отрицательна.

Выполним расчеты на примере рис. 6.1. Достоверность правила {*dog*} → *cat* равна 5/7, поскольку слово «dog» встречается в семи корзинах, из которых пять содержат и слово «cat». Однако «cat» встречается в шести из восьми корзин, поэтому следует ожидать, что 75 % из семи корзин, содержащих «dog», содержат и «cat». Следовательно, интересность правила равна 5/7 – 3/4 = –0.036, почти 0. У правила {*cat*} → *kitten* интересность равна 1/6 – 1/8 = 0.042, поскольку в одной из шести корзин со словом «cat» есть и «kitten», тогда как «kitten» встречается всего в одной из восьми корзин. Это значение, хоть и положительно, но все равно близко к 0, т. е. данное ассоциативное правило не очень интересно.

### 6.1.4. Поиск ассоциативных правил с высокой достоверностью

Выявление полезных ассоциативных правил не намного сложнее поиска частых предметных наборов. В этой главе мы вернемся к частым предметным наборам, но пока предположим, что умеем находить наборы с поддержкой не меньше пороговой величины $s$. Если мы ищем ассоциативные правила $I \rightarrow j$, применимые к достаточно большой доле корзин, то поддержка $I$ должна быть достаточно высока. На практике, например в реальных магазинах, «достаточно высока» означает порядка 1 % корзин. Кроме того, мы хотим, чтобы достоверность правила тоже была приличной, скажем 50 %, иначе практической пользы от него ждать не приходится. Таким образом, у набора $I \cup \{j\}$ тоже будет высокая поддержка.

Допустим, что мы нашли все предметные наборы с поддержкой не ниже пороговой и что для каждого из них точно вычислили поддержку. Мы можем найти среди них все ассоциативные правила, обладающие одновременно высокой поддержкой и высокой достоверностью. То есть, если $J$ – частый набор $n$ предметов, то существует только $n$ возможных ассоциативных правил, связанных с этим набором, а именно правила вида $J - \{j\} \rightarrow j$ для каждого $j$ из $J$. Если $J$ – частый набор, то $J - \{j\}$ должен быть по меньшей мере таким же частым. Следовательно, это тоже частый набор, и мы уже вычислили поддержку $J$ и $J - \{j\}$. Их отношение и есть достоверность правила $J - \{j\} \rightarrow j$.

Следует предположить, что частых предметных наборов не слишком много и, следовательно, потенциальных ассоциативных правил с высокой поддержкой и достоверностью тоже будет не так много. Причина в том, что каждое выявленное правило требует каких-то действий. Если дать директору магазина миллион ассоциативных правил с поддержкой и достоверностью выше установленных нами порогов, то он не сможет даже прочитать их, что уж говорить о действиях. Аналогично, если предложить миллион потенциальных биомаркеров, то мы не сможем поставить эксперименты, необходимые для их проверки. Поэтому стремятся подобрать пороги так, чтобы частых предметных наборов было сравнительно немного. Ниже мы покажем, что это предположение ведет к важным следствиям в плане эффективности алгоритмов нахождения частых предметных наборов.

### 6.1.5. Упражнения к разделу 6.1

**Упражнение 6.1.1.** Пусть имеется 100 предметов и 100 корзин, пронумерованных от 1 до 100. Предмет $i$ находится в корзине $b$ тогда и только тогда, когда $b$ нацело делится на $i$. Следовательно, предмет 1 находится во всех корзинах, предмет 2 – в пятидесяти четных корзинах и т. д. Корзина 12 содержит предметы {1, 2, 3, 4, 6, 12}, поскольку это все делители 12. Ответьте на следующие вопросы:

(*а*) Если пороговая поддержка равна 5, то какие предметы будут частыми?

! (*б*) Если пороговая поддержка равна 5, то какие пары предметов будут частыми?

! (*в*) Чему равна сумма размеров всех корзин?

**! Упражнение 6.1.2.** Какая корзина наибольшая в примере из упражнения 6.1.1?

**Упражнение 6.1.3.** Пусть имеется 100 предметов и 100 корзин, пронумерованных от 1 до 100. Предмет $i$ находится в корзине $b$ тогда и только тогда, когда $i$ нацело делится на $b$. Так, корзина 12 состоит из предметов {12, 24, 36, 48, 60, 72, 84, 96}. Повторите упражнение 6.1.1 при таких условиях.

**! Упражнение 6.1.4.** Этот вопрос касается данных, из которых нельзя извлечь интересной информации о частых предметных наборах, потому что не существует коррелированных наборов предметов. Предположим, что предметы пронумерованы от 1 до 10 и что предмет $i$ включается в корзину с вероятностью $1/i$, причем все решения принимаются независимо. Иными словами, каждая корзина содержит предмет 1, половина корзин содержит предмет 2, треть – предмет 3 и т. д. Предположим, что число корзин настолько велико, что их совокупность ведет себя в соответствии с законами статистики. Пусть пороговая поддержка равна 1 % от числа корзин. Найдите частые предметные наборы.

**Упражнение 6.1.5.** Какова достоверность следующих ассоциативных правил для данных из упражнения 6.1.1?

(*а*) $\{5, 7\} \rightarrow 2$
(*б*) $\{2, 3, 4\} \rightarrow 5$

**Упражнение 6.1.6.** Какова достоверность следующих ассоциативных правил для данных из упражнения 6.1.3?

(*а*) $\{24, 60\} \rightarrow 8$
(*б*) $\{2, 3, 4\} \rightarrow 5$

**!! Упражнение 6.1.7.** Опишите все ассоциативные правила с достоверностью 100 %, если данные о корзинах взяты из

(*а*) упражнения 6.1.1
(*б*) упражнения 6.1.3

**! Упражнение 6.1.8.** Докажите, что в данных из упражнения 6.1.4 нет интересных ассоциативных правил, т. е. интересность любого правила равна 0.

## 6.2. Корзины покупок и алгоритм Apriori

Мы начинаем разговор о том, как находить частые предметные наборы или выведенную из них информацию, например ассоциативные правила с высокой поддержкой и достоверностью. Здесь будет рассмотрено первое усовершенствование очевидных алгоритмов, получившее название Apriori, на основе которого было разработано много вариантов. В следующих двух разделах мы обсудим улучшения. Но прежде чем приступать к самому алгоритму Apriori, сформулируем предположения о хранении и манипулировании данными при поиске частых предметных наборов.

### 6.2.1. Представление данных о корзинах покупок

Мы уже упоминали, что, по предположению, данные о корзинах хранятся в файле – корзина за корзиной. Возможно, данные находятся в распределенной файловой системе, как в разделе 2.1, и корзины представляют собой объекты. А, быть может, это обычный файл, в котором корзины и их содержимое представлены с помощью символьных кодов.

**Пример 6.4.** Файл мог бы начинаться так:

```
{23,456,1001}{3,18,92,145}{...
```

Здесь корзина начинается символом { и заканчивается символом }. Предметы в корзине представлены целыми числами через запятую. То есть первая корзина содержит предметы 23, 456 и 1001, а вторая – предметы 3, 18, 92 и 145.

Иногда одна машина обрабатывает файл целиком. А иногда используется технология MapReduce или другой подобный инструмент для распределения работы между многими процессорами, и тогда каждый процессор получает только часть файла. Как выясняется, объединить результаты параллельной работы процессоров для получения всей коллекции предметных наборов с поддержкой не ниже пороговой трудно, так что этот вопрос мы рассмотрим только в разделе 6.4.4.

Мы также предполагаем, что размер файла корзин настолько велик, что он не помещается в оперативную память. Следовательно, стоимость алгоритма определяется, прежде всего, временем чтения корзин с диска. После того как блок корзин загружен в оперативную память, мы можем сгенерировать все его подмножества размера $k$. Поскольку, по предположению, средний размер корзины мал, генерация всех пар в оперативной памяти должна занять значительно меньше времени, чем чтение корзины с диска. Например, если в корзине 20 предметов, то количество пар равно $\binom{20}{2} = 190$, и их легко сгенерировать с помощью двух вложенных циклов.

По мере увеличения размера генерируемых подмножеств растет и время; для генерации всех наборов размера $k$ для корзины, содержащей $n$ предметов, требуется время $n^k/k!$. В конечном итоге это время окажется больше времени чтения данных с диска. Однако:

1. Зачастую нам нужны только небольшие частые предметные наборы, поэтому $k$ не бывает больше 2 или 3.
2. А если нам все-таки необходимы предметные размеры большого размера $k$, обычно есть возможность исключить многие предметы из каждой корзины, поскольку они не могут входить в частый предметный набор. Поэтому с ростом $k$ уменьшается $n$.

В общем, мы хотим сказать, что объем работы по анализу всех корзин обычно можно предполагать пропорциональным размеру файла. И значит, мы можем измерять время работы алгоритма поиска частых предметных наборов количеством операций чтения дисковых блоков из файла данных.

Отметим еще, что все обсуждаемые алгоритмы обладают тем свойством, что читают файл корзин последовательно. Следовательно, их можно охарактеризовать числом проходов по файлу, а время работы будет пропорционально произведению числа проходов на размер файла. Поскольку контролировать размер данных мы не можем, существенным является только число проходов, и именно эту сторону алгоритма поиска частых предметных наборов мы будем принимать во внимание при измерении времени его работы.

## 6.2.2. Использование оперативной памяти для подсчета предметных наборов

Однако необходимо остановиться еще на одном касающемся данных вопросе. Во всех алгоритмах поиска частых предметных наборов при проходе по данным необходимо вести много счетчиков. Например, требуется подсчитывать, сколько раз каждая пара встречается в корзинах. Если памяти недостаточно для хранения всех счетчиков, то простое прибавление 1 к какому-нибудь из них может привести к подгрузке страницы с диска. В таком случае алгоритм начинает пробуксовывать и работает на много порядков медленнее, чем когда все счетчики хранятся в памяти. Вывод – нельзя ничего подсчитывать, если нет уверенности, что для этого хватит оперативной памяти. Следовательно, у каждого алгоритма есть ограничение на количество обрабатываемых предметов.

> **Пример 6.5.** Пусть некоторый алгоритм должен подсчитывать все пары предметов, а всего предметов $n$. Тогда необходимо хранить в памяти $\binom{n}{2}$, или приблизительно $n^2/2$ целых чисел. Если целое занимает 4 байта, то потребуется $2n^2$ байтов. Если объем оперативной памяти составляет 2 ГБ, или $2^{31}$ байтов, то должно быть $n \leq 2^{15}$, или приблизительно $n < 33\,000$.

Хранить $\binom{n}{2}$ счетчиков, так чтобы можно было легко найти счетчик для пары $\{i, j\}$ – нетривиальная задача. Прежде всего, мы не делали никаких предположений о способе представления предметов. Это могут быть, к примеру, строки типа «хлеб». С точки зрения памяти, эффективнее представлять предметы последовательными целыми числами от 1 до $n$, где $n$ – количество различных предметов. Если предметы изначально представлены не так, то понадобится хэш-таблица для преобразования предметов на этапе чтения из файла. То есть при чтении каждого предмета мы вычисляем его хэш-код и смотрим, есть ли он уже в хэш-таблице. Если да, то берем из нее соответствующее предмету целое число. Если нет, назначаем ему следующее по порядку число (текущее значение количества различных прочитанных предметов) и записываем его вместе с предметом в хэш-таблицу.

### Метод треугольной матрицы

Даже после представления предметов целыми числами остается проблема: пару $\{i, j\}$ нужно учитывать только в одном месте. Например, можно было бы упорядочить элементы пары, считая, что $i < j$, и использовать только элемент $a[i, j]$ дву-

мерного массива $a$. При такой стратегии половина массива пропадает зря. Более эффективен одномерный *треугольный массив*. В элементе $a[k]$ хранится счетчик для пары $\{i, j\}$, $1 \le i < j \le n$, где

$$k = (i - 1)(n - i/2) + j - i.$$

При такой организации пары хранятся в лексикографическом порядке, т. е. сначала $\{1, 2\}$, $\{1, 3\}$, ... , $\{1, n\}$, затем $\{2, 3\}$, $\{2, 4\}$, ..., $\{2, n\}$ и так далее до $\{n - 2, n - 1\}$, $\{n - 2, n\}$ и, наконец, $\{n - 1, n\}$.

## Метод троек

Существует еще один подход к хранению счетчиков, который может оказаться более подходящим в зависимости от того, какая доля потенциально возможных пар предметов реально встречается хотя бы в одной корзине. Мы можем хранить счетчик в виде тройки $[i, j, c]$, означающей, что для пары $\{i, j\}$, где $i < j$, счетчик равен $c$. Используется некая структура, например хэш-таблица, в которой $i$ и $j$ образуют ключ, позволяющая узнать, существует ли тройка для данных $i$ и $j$, и, если да, то быстро найти ее. Будем называть такой способ хранения счетчиков *методом троек*.

В отличие от треугольной матрицы, в методе троек не нужно ничего хранить, если счетчик равен 0. С другой стороны, в этом методе требуется хранить не одно, а три числа для каждой пары, встречающейся в какой-то корзине. Кроме того, надо учесть накладные расходы на организацию хэш-таблицы или иной структуры данных для быстрого поиска. В итоге треугольная матрица оказывается лучше, если по меньшей мере 1/3 из общего числа $\binom{n}{2}$ потенциально возможных пар встречается в корзинах. Если же эта доля значительно меньше 1/3, то стоит подумать о методе троек.

**Пример 6.6.** Пусть имеется 100 000 предметов и 10 000 000 корзин по 10 предметов в каждой. Тогда в методе треугольной матрицы потребуется хранить $\binom{1000000}{2} \approx 5 \times 10^9$ целых счетчиков[2]. С другой стороны, общее число пар во всех корзинах равно $10^7 \times \binom{10}{2} = 4.5 \times 10^8$. Даже в худшем случае, когда каждая пара встречается ровно один раз, может быть не более $4.5 \times 10^8$ пар с ненулевыми счетчиками. Если бы мы использовали для хранения счетчиков метод троек, то количество хранимых целых чисел превышало бы эту величину в три раза, т. е. составило бы всего $1.35 \times 10^9$. Следовательно, в этом случае метод троек потребляет заведомо меньше памяти, чем треугольная матрица.

Но даже если бы корзин было в десять или в сто раз больше, следовало бы ожидать весьма неравномерного распределения предметов, так что метод троек все равно оказался бы эффективнее. То есть у некоторых пар значение счетчика было бы очень велико, а количество различных пар, встречающихся хотя бы в одной корзине, оказалось бы гораздо меньше теоретического максимума.

[2] Здесь и далее в этой главы мы пользуемся приближением $\binom{n}{2} = n^2/2$ для больших $n$.

## 6.2.3. Монотонность предметных наборов

Своей эффективностью обсуждаемые в этой главе алгоритмы в немалой степени обязаны так называемому свойству *монотонности* предметных наборов:

- Если набор предметов $I$ частый, то таковым является и любой его поднабор.

Причина проста. Пусть $J \subseteq I$. Тогда любая корзина, содержащая все элементы $I$, конечно же, содержит и все элементы $J$. Следовательно, счетчик для $J$ никак не меньше счетчика для $I$ и, если счетчик для $I$ не меньше $s$, то это верно и в отношении счетчика для $J$. Поскольку $J$ может содержаться в некоторых корзинах, где нет каких-то предметов из $I - J$, вполне возможно, что счетчик для $J$ строго больше счетчика для $I$.

Свойство монотонности не только лежит в основе алгоритма Apriori, но позволяет более компактно представить информацию о частых предметных наборах. Если задана пороговая поддержка $s$, то будем называть предметный набор *максимальным*, если никакой содержащий его набор не является частым. Если выписать только максимальные предметные наборы, то мы будем точно знать, что все поднаборы максимального набора частые, и ни один набор, не являющийся частью какого-то максимального набора, частым быть не может.

**Пример 6.7.** Снова рассмотрим данные из примера 6.1 с пороговой поддержкой $s = 3$. Мы видели, что имеется пять одноэлементных частых наборов: слова «cat», «dog», «a», «and» и «training». Каждый из них, кроме «training», содержится в каком-то частом двухэлементном наборе, поэтому единственным максимальным одноэлементным частым набором является {training}. Существует также пять частых двухэлементных наборов с $s = 3$:

| | | |
|---|---|---|
| {dog, a} | {dog, and} | {dog, cat} |
| {cat, a} | {cat, and} | |

Мы нашли также один трехэлементный частый набор {dog, cat, a}, а частых наборов с большим числом элементов нет. Таким образом, эта тройка – максимальный набор, но все три частых двухэлементных набора, которые она содержит, максимальными не являются. Из двухэлементных наборов частыми будут только {dog, and} и {cat, and}. Отметим, что, зная частые двухэлементные наборы, мы можем сделать вывод, что одноэлементные наборы типа {dog} частые.

## 6.2.4. Доминирование подсчета пар

Как вы, наверное, заметили, до сих пор мы говорили, в основном, о подсчете пар. И это понятно: на практике оперативная память используется, прежде всего, для определения частых пар. Количество предметов, конечно, может быть очень большим, но, как правило, все же не настолько, чтобы мы не могли подсчитать в памяти все одноэлементные наборы.

А как насчет более крупных наборов – троек, четверок и т. д.? Напомним, что анализ частых предметных наборов имеет смысл, только если наборов получается сравнительно немного, иначе мы просто не сможем просмотреть их, не говоря уже о том, чтобы оценить значимость. Поэтому на практике пороговую поддержку устанавливают достаточно высокой, так чтобы частые наборы были редкостью. Из свойства монотонности следует, что если имеется частая тройка, то частыми будут и все три входящих в нее пары. И, разумеется, могут существовать частые пары, не входящие ни в одну частую тройку. Поэтому мы найдем больше частых пар, чем частых троек, больше частых троек, чем частых четверок, и т. д.

Этого аргумента было бы недостаточно, если бы оказалось невозможно избежать подсчета всех троек, потому что общее число троек гораздо больше числа пар. Задача алгоритма Apriori и родственных ему – избежать подсчета многих троек и более крупных наборов, и, как мы увидим, они с ней эффективно справляются. Поэтому в дальнейшем мы сосредоточимся на алгоритмах подсчета частых пар.

## 6.2.5. Алгоритм Apriori

Пока сконцентрируемся на поиске одних лишь частых пар. Если оперативной памяти хватает для подсчета всех пар с помощью одного из методов, упомянутых в разделе 6.2.2 (треугольная матрица или метод троек), то все сводится к однократному чтению файла корзин. Для каждой корзины мы в двух вложенных циклах генерируем все пары. При генерации каждой пары прибавляем 1 к ее счетчику. В конце просматриваем все пары и отбираем те, для которых счетчики не меньше пороговой поддержки $s$. Это и будут частые пары.

Однако это простое решение не годится, если пар предметов слишком много и оперативной памяти не хватает. Алгоритм Apriori устроен так, чтобы сократить количество подсчитываемых пар ценой двух проходов по файлу вместо одного.

### Первый проход Apriori

На первом проходе создаются две таблицы. Первая таблица при необходимости преобразует названия предметов в целые числа от 1 до $n$, как описано в разделе 6.2.2. Вторая таблица – массив счетчиков; в $i$-м элементе запоминается, сколько раз встречался предмет с номером $i$. В начальный момент счетчики для всех предметов равны 0.

По мере чтения файла мы достаем каждый предмет из корзины и преобразуем его название в целое число. Затем используем это число как индекс массива счетчиков и прибавляем 1 к элементу массива с таким индексом.

### Между проходами Apriori

После первого прохода по счетчикам предметов определяем частые одноэлементные наборы. Может показаться странным, что многие одноэлементные наборы частыми не являются. Но вспомните, что порог $s$ выбран настолько высоким, чтобы частых предметных наборов было не слишком много; обычно выбирают $s$ на уровне 1 % от числа корзин. Подумайте, как вы ходите в супермаркет. Конечно,

какие-то продукты вы покупаете чаще, чем в 1 % случаев: молоко, хлеб, газировку и т. п. Можно даже поверить, что 1 % посетителей покупают подгузники, хотя вы, возможно, не из их числа. Однако на полках полно товаров, которые интересуют менее 1 % покупателей, например сливочная заправка для салата Цезарь.

На втором проходе Apriori мы присваиваем новые номера от 1 до $m$ только частым предметам. Получается массив, проиндексированный от 1 до $n$, в котором $i$-й элемент равен либо 0, если предмет $i$ не частый, либо уникальному целому числу в диапазоне от 1 до $m$, если частый. Будем называть его таблицей *частых предметов*.

### Второй проход Apriori

На втором проходе мы подсчитываем пары, состоящие из двух частых предметов. Напомним, что пара не может быть частой, если таковым не является хотя бы один из ее элементов. Поэтому ни одной частой пары мы не пропустим. Для второго прохода требуется $2m^2$, а не $2n^2$ байтов, если для подсчета применяется треугольная матрица. Отметим, что перенумерация только частых предметов необходима, чтобы можно было использовать матрицу правильного размера. Весь набор структур данных в оперативной памяти на первом и втором проходе показан на рис. 6.3.

Отметим также, что эффект исключения нечастых предметов мультипликативен: если только половина предметов частые, то для счетчиков потребуется в четыре раза меньше памяти. Аналогично, если применяется метод троек, то считать необходимо только те пары двух частых элементов, которые встречаются по крайней мере в одной корзине.

Второй проход устроен следующим образом.

1. Для каждой корзины по таблице частых предметов смотрим, какие из находящихся в ней предметов частые.
2. Во вложенном цикле генерируем все пары частых предметов из этой корзины.
3. Для каждой такой пары прибавляем 1 к ее счетчику в структуре данных для хранения счетчиков.

Наконец, в конце второго прохода мы просматриваем структуру, где хранятся счетчики, и находим частые пары.

## 6.2.6. Применение Apriori для поиска всех частых предметных наборов

Та же техника, что применена для поиска частых пар без подсчета всех пар, позволяет искать и более крупные частые предметные наборы, не подсчитывая все без исключения. В алгоритме Apriori выполняется один проход для каждого размера набора $k$. Если частых предметных наборов некоторого размера не найдено, то из свойства монотонности следует, что и более крупных наборов не будет, а, значит, можно останавливаться.


**Рис. 6.3.** Организация оперативной памяти на двух проходах алгоритма Apriori

Переход от размера $k$ к размеру $k + 1$ можно описать следующим образом. Для каждого размера $k$ существует два множества предметных наборов:

1. $C_k$ – множество *наборов-кандидатов* размера $k$, т. е. наборов, которые нужно подсчитывать, чтобы понять, являются ли они частыми.
2. $L_k$ – множество действительных частых предметных наборов размера $k$.

Как происходит переход от одного множества предметных наборов и размера к следующему, показано на рис. 6.4.

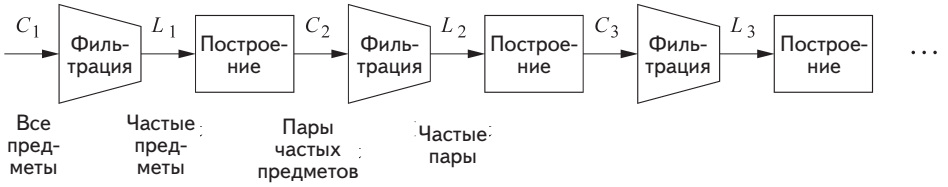



**Рис. 6.4.** В алгоритме Apriori чередуется построение множества наборов-кандидатов и его фильтрация для поиска частых наборов

Мы начинаем с множества $C_1$, содержащего все одноэлементные предметные наборы, т. е. сами предметы. Это означает, что до просмотра данных любой предмет в принципе может быть частым. На первом шаге фильтрации мы вычисляем счетчики всех предметов и оставляем в множестве частых предметов $L_1$ только те, для которых счетчик не меньше пороговой поддержки $s$.

В множество $C_2$ пар-кандидатов входят пары предметов из $L_1$, т. е. пары частых предметов. Отметим, что $C_2$ явно не строится. Мы просто используем определе-

ние $C_2$ и для проверки членства в нем проверяем, являются ли оба предмета элементами $L_1$. На втором проходе алгоритма Apriori мы вычисляем счетчики всех пар-кандидатов и определяем, какие из них встречаются не менее $s$ раз. Эти пары образуют множество частых пар $L_2$.

Продолжать эту последовательность можно до бесконечности. Множество $C_3$ кандидатов-троек строится (неявно) как множество троек, в которых любые два предмета образуют пару, имеющуюся в $L_2$. Из предположения о редкости частых предметных наборов, выдвинутого в разделе 6.2.4, следует, что частых пар будет немного, поэтому их можно сохранить в таблице в оперативной памяти. Точно так же, немного будет троек-кандидатов, поэтому все можно пересчитать с помощью обобщения метода троек. Обобщение заключается в том, что если для подсчета пар мы использовали тройки, то для подсчета троек будем использовать четверки, состоящие из номеров трех предметов и ассоциированного с этой тройкой счетчика. Аналогично можно подсчитать наборы размера $k$, применяя кортежи с $k + 1$ компонентами, из которых последняя – счетчик, а первые $k$ – номера предметов в отсортированном порядке.

Для нахождения $L_3$ мы выполняем третий проход по файлу корзин. В каждой корзине нужно просматривать только элементы, вошедшие в $L_1$. Из них мы образуем все пары и смотрим, имеется пара в $L_2$ или нет. Никакой предмет из корзины, не встречающийся хотя бы в двух парах, образованных предметами из той же корзины, не может входить ни в одну содержащуюся в ней частую тройку. Таким образом, мы ограничили поиск троек теми, которые одновременно содержатся в корзине и являются кандидатами, вошедшими в $C_3$. Счетчик любой такой тройки увеличивается на 1.

> **Пример 6.8.** Пусть корзина состоит из предметов от 1 до 10. Из них предметы с 1 по 5 оказались частыми. И было обнаружено, что частыми являются следующие пары: {1, 2}, {2, 3}, {3, 4}, {4, 5}. Сначала мы исключаем нечастые предметы, оставляя только предметы с 1 по 5. Однако каждый из предметов 1 и 5 встречается только в одной только в одной частой паре и потому не может входить в частую тройку, содержащуюся в этой корзине. Следовательно, вопрос об увеличении счетчика на 1 может стоять только для троек, образованных предметами из набора {2, 3, 4}. Естественно, такая тройка только одна. Однако в $C_3$ мы ее не найдем, потому что двойка {2, 4}, очевидно, нечастая.

Построение коллекций более крупных частых предметных наборов и кандидатов продолжается тем же порядком, пока на каком-то проходе не окажется, что частых наборов больше нет. В этом случае алгоритм останавливается. Итак:

1. Определяем $C_k$ как множество таких предметных наборов размера $k$, что любые $k - 1$ предметов образуют набор, входящий в $L_{k-1}$.
2. Находим $L_k$, выполняя проход по файлу корзин и подсчитывая только те предметные наборы размера $k$, которые входят в $C_k$. Те наборы, для которых счетчик окажется не меньше $s$, войдут в $L_k$.

## 6.2.7. Упражнения к разделу 6.2

**Упражнение 6.2.1.** Пусть количество предметов $n$ равно 20. Если воспользоваться треугольной матрицей для подсчета пар, то счетчик какой пары будет храниться в элементе $a[100]$?

**! Упражнение 6.2.2.** В описании метода треугольной матрицы из раздела 6.2.2 формула для $k$ включает деление произвольного целого числа на 2. Однако $k$ должно быть целым. Докажите, что $k$, вычисляемое по этой формуле, на самом деле всегда целое.

**! Упражнение 6.2.3.** Пусть в наборе данных о корзинах покупок имеется $I$ предметов, распределенных по $B$ корзинам. Предположим, что каждая корзина содержит ровно $K$ предметов. Выразите в виде функции от $I$, $B$ и $K$:

(*а*) объем памяти, необходимой для хранения всех пар предметов в методе треугольной матрицы, в предположении, что один элемент массива занимает 4 байта;

(*б*) наибольшее возможное число пар с ненулевым счетчиком;

(*в*) условия, при которых метод троек гарантированно потребляет меньше памяти, чем треугольная матрица.

**!! Упражнение 6.2.4.** Как бы вы стали подсчитывать все предметные наборы размера 3 путем обобщения метода треугольной матрицы? То есть требуется организовать одномерный массив, в котором имеется ровно один элемент для каждого набора из трех предметов.

**! Упражнение 6.2.5.** Пусть пороговая поддержка равна 5. Найдите максимальные частые предметные наборы для данных из:

(*а*) упражнения 6.1.1;

(*б*) упражнения 6.1.3.

**Упражнение 6.2.6.** Примените алгоритм Apriori с пороговой поддержкой 5 к данным из

(*а*) упражнения 6.1.1;

(*б*) упражнения 6.1.3.

**! Упражнение 6.2.7.** Допустим, имеются корзины, удовлетворяющие следующим условиям:

1. Пороговая поддержка равна 10 000.
2. Существует миллион предметов, представленных числами 0, 1, ..., 999999.
3. Существует $N$ частых предметов, т. е. таких, которые встречаются не менее 10 000 раз.
4. Существует один миллион пар, встречающихся не менее 10 000 раз.
5. Существует два миллиона пар, встречающихся ровно один раз. Из этих пар $M$ содержат два частых предмета, а еще $M$ содержат по крайней мере один нечастый предмет.
6. Больше никакие пары не встречаются.
7. Целое число представляется 4 байтами.

Предположим, что мы выполняем алгоритм Apriori и на втором проходе можем выбирать, как хранить счетчики пар-кандидатов: в треугольной матрице или в хэш-таблице вида предмет-предмет-счетчик. В первом случае пренебрегаем памятью, необходимой для преобразования исходных номеров предметов в номера частых предметов, а во втором – накладными расходами на организацию хэш-таблицы. Выразите в виде функции от $N$ и $M$ минимальное число байтов оперативной памяти, необходимое для выполнения алгоритма Apriori при таких данных.

# 6.3. Обработка больших наборов данных в оперативной памяти

Алгоритм Apriori всех устраивает, если на шаге с наибольшими требованиями к памяти – обычно это подсчет пар-кандидатов в множестве $C_2$ – памяти достаточно для работы без пробуксовки (многократного перемещения данных между диском и оперативной памятью). Было предложено несколько алгоритмов для уменьшения размера множества кандидатов $C_2$. Здесь мы рассмотрим алгоритм PCY, в котором используется тот факт, что на первом проходе Apriori обычно имеется много памяти, не нужной для подсчета одиночных предметов. Затем мы обсудим многоэтапный алгоритм, в котором используется идея PCY и добавляются проходы для дальнейшего уменьшения размера $C_2$.

## 6.3.1. Алгоритм Парка-Чена-Ю (PCY)

В этом алгоритме, называемом PCY по первым буквам фамилий авторов (Park, Chen, Yu) находит применение тот факт, что на первом проходе значительная часть оперативной памяти часто не используется. Если есть миллион предметов и несколько гигабайтов оперативной памяти, то нам понадобится не более 10 % памяти для двух таблиц, показанных на рис. 6.3: таблицы преобразования названий предметов в небольшие целые числа и массива для хранения счетчиков этих чисел. В алгоритме PCY в этой памяти размещается массив целых чисел, обобщающий идею фильтра Блума (см. раздел 4.3). Эта идея схематически изображена на рис. 6.5.

Представьте этот массив как хэш-таблицу, в ячейках которой хранятся целые числа, а не множества ключей (как в обычной хэш-таблице) или биты (как в фильтре Блума). Пары предметов хэшируются в ячейки таблицы. Просматривая корзину на первом проходе, мы не только прибавляем 1 к счетчикам предметов в ней, но и генерируем во вложенном цикле все пары. Каждая пара хэшируется и к значению в соответствующей ей ячейки прибавляется 1. Заметим, что сама пара в ячейку не кладется, она лишь влияет на единственное находящееся в ячейке целое число.

В конце первого прохода в каждой ячейке находится счетчик, равный сумме счетчиков всех пар, хэшированных в эту ячейку. Если счетчик в некоторой ячейке не меньше пороговой поддержки, будем называть такую ячейку *частой*. На основе имеющейся информации мы ничего не можем сказать о парах, хэшированных в ча-

стую ячейку, теоретически все они могут быть частыми. Но если счетчик в ячейке меньше $s$ (*нечастая ячейка*), то мы точно знаем, что никакая из хэшированных в эту ячейку пар не может быть частой, даже если состоит из двух частых элементов. И этим наблюдением можно воспользоваться на втором проходе. Определим множество пар-кандидатов $C_2$ как множество таких пар $\{i, j\}$, что:

1. $i$ и $j$ – частые предметы.
2. $\{i, j\}$ хэшируется в частую ячейку.

Именно второе условие отличает PCY от Apriori.

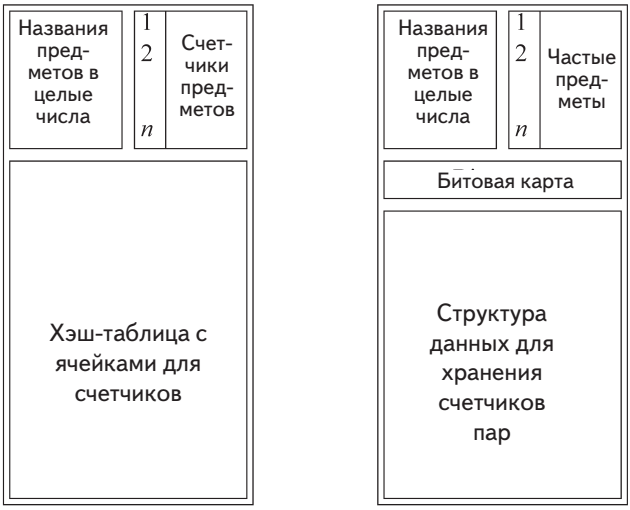



**Рис. 6.5.** Организация оперативной памяти на первых двух проходах алгоритма PCY

**Пример 6.9.** В зависимости от характера данных и объема доступной памяти использование хэш-таблицы на первом проходе может оказаться выгодным или невыгодным. В худшем случае все ячейки будут частыми, и алгоритм PCY на втором проходе будет подсчитывать в точности те же пары, что Apriori. Но иногда можно ожидать, что в большинстве своем ячейки будут нечастыми, и тогда PCY позволяет ослабить требования к памяти на втором проходе.

Пусть имеется гигабайт оперативной памяти, в котором можно разместить хэш-таблицу на первом проходе. Предположим также, что файл содержит миллиард корзин по десять предметов в каждой. Ячейка – это целое число, обычно размером 4 байта, поэтому у нас хватит места для четверти миллиарда ячеек. Число пар во всех корзинах равно $10^9 \times \binom{10}{2}$, или $4.5 \times 10^{10}$, оно же равно сумме счетчиков в ячейках. Таким образом, средняя величина счетчика равна $4.5 \times 10^{10}/2.5 \times 10^8 = 180$. Если пороговая поддержка $s$ не больше 180, то мы можем ожидать небольшого количества нечастых ячеек.

Если же $s$ гораздо больше, скажем, порядка 1000, то огромное большинство ячеек будут нечастыми. Максимально возможное число частых ячеек равно $4.5 \times 10^{10}/1000$, или 45 миллионов из 250 миллионов.

Между проходами PCY хэш-таблица сводится к *битовой карте*, в которой каждой ячейке соответствует один бит. Этот бит равен 1, если ячейка частая, и 0, если нет. Следовательно, целые числа длиной 32 бита заменяются одиночными битами, и битовая карта, показанная на рис. 6.5, занимает всего 1/32 памяти, которая понадобилась бы для хранения счетчиков в противном случае. Но если большинство ячеек нечастые, то можно ожидать, что количество пар, подсчитываемых на втором проходе, окажется гораздо меньше общего числа пар частых предметов. Следовательно, PCY иногда может без пробуксовки обработать на втором проходе наборы данных такого размера, для которого Apriori не хватило бы памяти.

Существует еще одна тонкость, касающаяся второго прохода PCY, которая тоже влияет на объем потребной памяти. На втором проходе Apriori мы при желании могли использовать метод треугольной матрицы, поскольку частые предметы можно было перенумеровать от 1 до некоторого $m$. Но в PCY это не получится. Дело в том, что пары частых предметов, от подсчета которых PCY нас избавляет, встречаются в треугольной матрице в случайных местах; это пары, которые на первом проходе были хэшированы в нечастую ячейку. Неизвестен способ сжатия матрицы, который позволил бы не тратить зря место на не подсчитываемые пары.

Таким образом, в PCY мы вынуждены использовать метод троек. Это ограничение, возможно, не играет особой роли, если доля пар частых предметов, действительно встречающихся в корзинах невелика; тогда нам в любом случае выгоднее использовать в Apriori тройки. Но если большинство пар частых предметов встречаются вместе хотя бы в одной ячейке, то в PCY мы вынуждены использовать метод троек, тогда как в Apriori предпочли бы воспользоваться треугольной матрицей. Следовательно, если PCY позволяет избежать подсчета менее 2/3 пар частых предметов, то мы не получим выигрыша по сравнению с Apriori. Хотя выявление частых пар в PCY происходит существенно иначе, чем в APriori, на более поздних стадиях, когда мы ищем частые тройки и более крупные наборы, отличий от Apriori практически нет. Это справедливо и для других улучшений Apriori, рассматриваемых в этом разделе. Поэтому, начиная с этого места, мы будем описывать только построение частых пар.

## 6.3.2. Многоэтапный алгоритм

*Многоэтапный алгоритм* (Multistage Algorithm) улучшает PCY за счет использования нескольких последовательных хэш-таблиц, чтобы еще уменьшить количество пар-кандидатов. В обмен для поиска частых пар выполняется еще два прохода. Схема многоэтапного алгоритма показана на рис. 6.6.

Первый проход многоэтапного алгоритма такой же, как первый проход PCY. После него находятся и сворачиваются в битовую карту частые ячейки – снова как в PCY. Но на втором проходе пары-кандидаты не подсчитываются. Вместо

этого имеющаяся оперативная память используется для размещения еще одной хэш-таблицы, с другой хэш-функцией. Поскольку битовая карта, полученная из первой хэш-таблицы, занимает 1/32 часть располагаемой памяти, то во второй хэш-таблице будет почти столько же ячеек, сколько в первой.

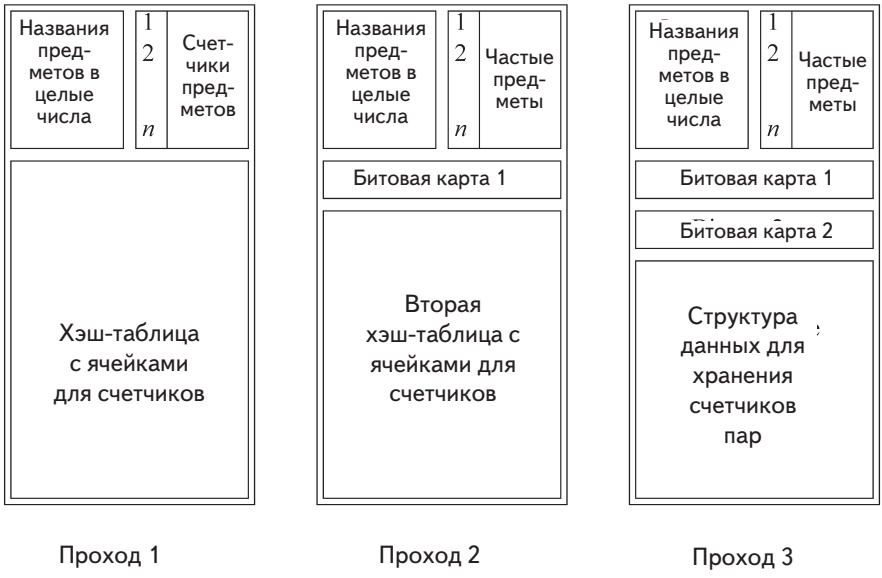



**Рис. 6.6.** В многоэтапном алгоритме используются дополнительные хэш-таблицы для уменьшения числа пар-кандидатов

На втором проходе мы снова просматриваем файл корзин. Подсчитывать предметы еще раз нет нужды, поскольку эти счетчики уже сохранены. Но мы должны сохранить информацию о том, какие предметы частые, поскольку она понадобится на втором и третьем проходах. Во время второго прохода мы хэшируем некоторые пары предметов в ячейки второй хэш-таблицы. Пара $\{i, j\}$ хэшируется только тогда, когда удовлетворяет обоим критериям подсчета на втором проходе алгоритма PCY, т. е. оба предмета $i$ и $j$ частые, а сама пара была хэширована в частую ячейку на первом проходе. В результате сумма счетчиков во второй хэш-таблице должна быть значительно меньше суммы на первом проходе. И потому, хотя количество ячеек во второй хэш-таблице составляет лишь 31/32 от количества ячеек первой, мы ожидаем, что во второй таблице будет намного меньше частых ячеек, чем в первой.

После второго прохода вторая хэш-таблица также сворачивается в битовую карту, которая сохраняется в оперативной памяти. Обе битовые карты занимают чуть меньше 1/16 части имеющейся памяти, так что остается еще достаточно места для подсчета пар-кандидатов на третьем проходе. Пара $\{i, j\}$ входит в $C_2$ тогда и только тогда, когда:

1. $i$ и $j$ – частые предметы.
2. Пара $\{i, j\}$ была хэширована в частую ячейку первой хэш-таблицы.
3. Пара $\{i, j\}$ была хэширована в частую ячейку второй хэш-таблицы.

Третье условие отличает многоэтапный алгоритм от PCY.

Понятно, что между первым и последним проходом в многоэтапном алгоритме можно вставить еще сколько угодно проходов. Лимитирующим является тот факт, что после каждого прохода необходимо сохранять битовые карты всех предыдущих проходов. В конце концов не останется оперативной памяти для счетчиков. Но сколько бы проходов ни выполнять, действительно частые пары всегда хэшируются в частые ячейки, поэтому избежать их подсчета невозможно.

**Тонкая ошибка в многоэтапном алгоритме**

Иногда, реализуя этот алгоритм, пытаются исключить второе требование к паре-кандидату $\{i, j\}$ – что она должна хэшироваться в частую ячейку на первом проходе. И аргументируют это следующим – некорректным – образом: если бы пара не была хэширована в частую ячейку на первом проходе, то она вообще не хэшировалась бы на втором проходе и, стало быть, не вносила бы вклад в свою ячейку на втором проходе. Да, действительно, пара не подсчитывается на втором проходе, но это не означает, что она не была бы хэширована в частую ячейку, если бы хэшировалась. Поэтому вполне возможно, что $\{i, j\}$ состоит из двух частых предметов и хэшируется в частую ячейку на втором проходе, хотя и не была хэширована в частую ячейку на первом проходе. Таким образом, на проходе многоэтапного алгоритма, где вычисляются счетчики, должны быть соблюдены все три условия.

## 6.3.3. Многохэшевый алгоритм

Иногда мы можем получить почти все преимущества дополнительных проходов многоэтапного алгоритма за один проход. Этот вариант PCY называется многохэшевым алгоритмом (Multihash Algorithm). Вместо использования двух разных хэш-таблиц на двух последовательных проходах, мы используем две хэш-функции и две отдельные хэш-таблицы в оперативной памяти на одном проходе, как показано на рис. 6.7.

Опасность использования двух хэш-таблиц на одном проходе заключается в том, в каждой хэш-таблице вдвое меньше ячеек, чем в одной большой таблице в алгоритме PCY. При условии, что средняя величина счетчика в ячейке в алгоритме PCY много меньше пороговой поддержки, мы можем работать с двумя хэш-таблицами половинного размера и при этом по-прежнему ожидать, что большинство ячеек в обеих таблицах будут нечастыми. Следовательно, в такой ситуации вполне можно выбрать многохэшевый алгоритм.

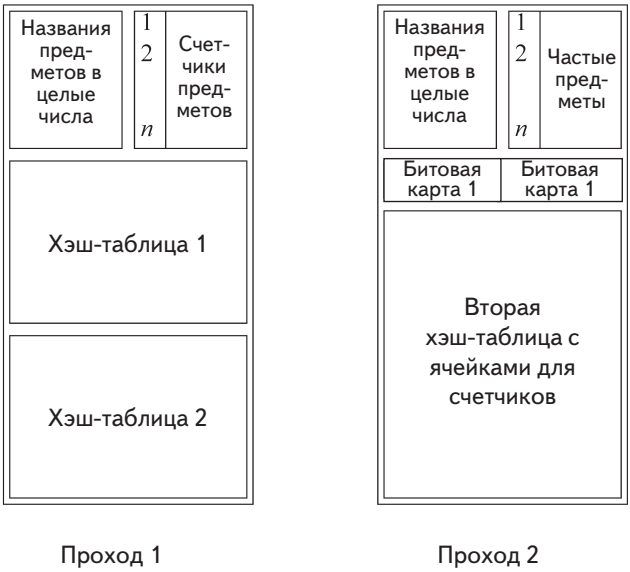



**Рис. 6.7.** В многохэшевом алгоритме на одном проходе используется несколько хэш-таблиц

**Пример 6.10.** Предположим, что выполняется алгоритм PCY, и средний счетчик в ячейке равен $s/10$, где $s$ – пороговая поддержка. Тогда, если бы мы использовали многохэшевый алгоритм с двумя хэш-таблицами половинного размера, то средний счетчик был бы равен $s/5$. В результате не более 1/5 ячеек в каждой таблице были бы частыми, а случайно выбранная нечастая пара с вероятностью не более $(1/5)^2 = 0.04$ оказалась бы в частой ячейке в обеих хэш-таблицах.

Аналогичное рассуждение показывает, что нечастая пара может оказаться в частой ячейке одной хэш-таблицы в алгоритме PCY с вероятностью не более 1/10. Таким образом, можно ожидать, что количество нечастых пар в PCY, будет в 2.5 раза больше, чем в многохэшевом варианте. Поэтому многохэшевый алгоритм, по-видимому, будет потреблять меньше памяти на втором проходе.

Но верхние границы не дают полной картины. Количество частых ячеек в каждом алгоритме может оказаться намного меньше максимального, поскольку наличие нескольких очень частых пар смещает распределение счетчиков в ячейках. Однако этот анализ позволяет предположить, что для некоторых данных и порогов поддержки есть возможность получить выигрыш от одновременного хранения нескольких таблиц с разными хэш-функциями в оперативной памяти.

На втором проходе многохэшевого алгоритма каждая хэш-таблица, как обычно, сворачивается в битовую карту. Отметим, что две битовые карты для двух хэш-функций на рис. 6.7 занимают ровно столько места, сколько одна битовая карта

заняла бы на втором проходе PCY. Условия, при которых пара $\{i, j\}$ попадает в $C_2$ и потому должна подсчитываться на втором проходе, такие же, как на третьем проходе многоэтапного алгоритма: оба предмета $i$ и $j$ должны быть частыми и пара должна хэшироваться в частую ячейку в обеих хэш-таблицах.

Напомним, что многоэтапный алгоритм не ограничен двумя хэш-таблицами. И точно так же на первом проходе многохэшевого алгоритма мы можем разделить имеющуюся оперативную память между произвольным числом хэш-таблиц. Опасность в том, что если взять слишком много таблиц, то средняя величина счетчика в ячейке превысит пороговую поддержку. Тогда в каждой хэш-таблице окажется очень мало нечастых ячеек. И хотя пара подсчитывается, только если она хэшируется в частую ячейку в каждой хэш-таблице, может оказаться, что при добавлении новой таблицы вероятность стать кандидатом для нечастой пары возрастет, а не уменьшится.

## 6.3.4. Упражнения к разделу 6.3

**Упражнение 6.3.1.** Ниже показаны 12 корзин. Каждая содержит три из шести предметов, пронумерованных от 1 до 6.

$\{1, 2, 3\}$ $\{2, 3, 4\}$ $\{3, 4, 5\}$ $\{4, 5, 6\}$
$\{1, 3, 5\}$ $\{2, 4, 6\}$ $\{1, 3, 4\}$ $\{2, 4, 5\}$
$\{3, 5, 6\}$ $\{1, 2, 4\}$ $\{2, 3, 5\}$ $\{3, 4, 6\}$

Пусть пороговая поддержка равна 4. На первом проходе алгоритма PCY используется хэш-таблица с 11 ячейками, и пара $\{i, j\}$ хэшируется в ячейку $i \times j$ по модулю 11.

(*а*) Любым методом вычислите поддержку каждого предмета и каждой пары предметов.
(*б*) Для каждой пары укажите, в какую ячейку она хэшируется.
(*в*) Какие ячейки являются частыми?
(*г*) Какие пары подсчитываются на втором проходе алгоритма PCY?

**Упражнение 6.3.2.** Предположим, что для данных из упражнения 6.3.1 выполняется многоэтапный алгоритм. Первый проход такой же, как в предыдущем упражнении, а на втором проходе мы хэшируем пары в девять ячеек с помощью хэш-функции $\{i, j\}$ в $(i + j) \bmod 9$. Какие счетчики окажутся в каждой ячейке после второго прохода? Уменьшает ли второй проход количество пар-кандидатов? Заметим, что все предметы частые, поэтому пара не будет хэшироваться на втором проходе, только если на первом она хэширована в нечастую ячейку.

**Упражнение 6.3.3.** Предположим, что для данных из упражнения 6.3.1 выполняется многохэшевый алгоритм. Будем использовать две хэш-таблицы по пять ячеек в каждой. Первая хэш-функция отображает пару $\{i, j\}$ в $(2i + 3j + 4) \bmod 5$, вторая – в $(i + 4j) \bmod 5$. Поскольку эти хэш-функции не симметричны относительно $i$ и $j$, то перед их вычислением упорядочиваем предметы, так чтобы $i < j$. Найдите значения счетчиков в каждой из 10 ячеек. Насколько велика должна

быть пороговая поддержка, чтобы многохэшевый алгоритм исключил больше пар, чем алгоритм PCY с одной хэш-таблицей и функцией из упражнения 6.3.1?

! **Упражнение 6.3.4.** Предположим, что мы выполняем алгоритм PCY для поиска частых пар при следующих условиях на корзины:

1. Пороговая поддержка равна 10 000.
2. Существует миллион предметов, представленных числами 0, 1, ..., 999999.
3. Существует 250 000 частых предметов, встречающихся не менее 10 000 раз.
4. Существует миллион пар, встречающихся не менее 10 000 раз.
5. Существует $P$ пар, встречающихся ровно по одному разу и образованных двумя частыми предметами.
6. Никаких других пар нет.
7. Целое число занимает 4 байта.
8. При хэшировании пары распределяются по ячейкам случайно, но равномерно, насколько это возможно, т. е. можно предполагать, что во всех ячейках окажется в точности поровну пар, встречающихся по одному разу.

Предположим, что объем оперативной памяти – $S$ байтов. Для успешного выполнения алгоритма PCY количество ячеек должно быть достаточно велико, так чтобы большинство ячеек были нечастыми. Кроме того, на втором проходе должно быть достаточно места для подсчета всех пар-кандидатов. Выразите в виде функции от $S$ максимальное значение $P$, при котором алгоритм PCY можно успешно применить к описанным данным.

! **Упражнение 6.3.5**. В предположениях упражнения 6.3.4 ответьте, ослабит ли многохэшевый алгоритм требования к объему оперативной памяти на втором проходе. Выразите в виде функции от $S$ и $P$ оптимальное число хэш-таблиц на первом проходе.

! **Упражнение 6.3.6.** Предположим, что для поиска частых пар используется 3-проходный многохэшевый алгоритм при следующих условиях на корзины:

1. Пороговая поддержка равна 10 000.
2. Существует миллион предметов, представленных числами 0, 1, ..., 999999. Все предметы частые, т. е. встречаются не менее 10 000 раз.
3. Существует миллион пар, встречающихся не менее 10 000 раз.
4. Существует $P$ пар, встречающихся ровно по одному разу
5. Никаких других пар нет.
6. Целое число занимает 4 байта.
7. При хэшировании пары распределяются по ячейкам случайно, но равномерно, насколько это возможно, т. е. можно предполагать, что во всех ячейках окажется в точности поровну пар, встречающихся по одному разу.
8. Хэш-функции, используемые на первых двух проходах, независимы.

Предположим, что объем оперативной памяти равен $S$ байтов. Выразите в виде функции от $S$ и $P$ ожидаемое число пар-кандидатов на третьем проходе многохэшевого алгоритма.

# 6.4. Алгоритм с ограниченным числом проходов

В алгоритмах поиска частых предметных наборов, рассмотренных до сих пор, использовалось по одному проходу для каждого размера набора. Если оперативной памяти не хватает для хранения счетчиков частых предметных наборов одного размера, то не видно способа избежать $k$ проходов для точного вычисления всех частых наборов. Однако есть много приложений, в которых необязательно знать все частые предметные наборы. Например, при поиске товаров, которые покупают вместе, администрация супермаркета вряд ли будет устраивать распродажи для каждого найденного частого предметного набора, так что вполне достаточно найти не все, а только большинство наборов.

В этом разделе мы исследуем некоторые алгоритмы, предложенные для поиска всех или большинства частых предметных наборов за два прохода. Начнем с очевидного решения – использовать не весь набор данных, а только выборку из него. В алгоритме SON используется два прохода, он находит точный ответ и легко реализуется с помощью технологии MapReduce или другой технологии распараллеливания. Наконец, в алгоритме Тойвонена, который тоже дает точный ответ, используется в среднем два прохода, но в редких случаях он может не завершаться за любое наперед указанное время.

## 6.4.1. Простой рандомизированный алгоритм

Вместо того чтобы использовать файл корзин целиком, мы могли бы взять случайную выборку и сделать вид, что это весь набор данных. Пороговую поддержку можно скорректировать, так чтобы она отражала уменьшенное число корзин. Например, если пороговая поддержка для полного набора данных равна $s$, и мы включаем в выборку 1 % корзин, то следует проанализировать предметные наборы, встречающиеся по меньшей мере в $s/100$ корзинах.

Самый безопасный способ сформировать выборку – читать весь набор данных и выбирать корзину с фиксированной вероятностью $p$. Предположим, что в файле представлено $m$ корзин. По завершении чтения мы будем иметь выборку размером приблизительно $pm$. Однако если есть основания полагать, что корзины в файле уже расположены в случайном порядке, то читать весь файл необязательно, а можно взять первые $pm$ корзин. Или, если файл находится в распределенной файловой системе, можно включить в выборку случайные блоки.

Сформировав выборку корзин, мы сохраним их в части оперативной памяти. Другая часть используется для выполнения одного из рассмотренных выше алгоритмов: Apriori, PCY, многоэтапного ли многохэшевого. Разница в том, что на всех проходах для каждого размера предметного набора алгоритм применяется к выборке, хранящейся в памяти, так что читать ее с диска не нужно. Когда частые предметные наборы будут найдены, их можно записать на диск. Это, а также начальное чтение файла для получения выборки – единственные операции ввода-вывода в алгоритме.

**Почему бы не выбрать корзины из начала файла?**

Формировать выборку из одного куска большого файла рискованно, потому что данные могут быть распределены неравномерно. Предположим, к примеру, что в файле записано содержимое настоящих корзин покупателей в универмаге, отсортированных по дате продажи. Взяв только первые корзины, мы получим старые данные. Так, в корзинах может не оказаться iPod'ов, хотя впоследствии этот товар стал очень популярен.

Другой пример: рассмотрим файл с данными о медицинских анализах, произведенных в разных больницах. Если каждая порция относится к одной больнице, то при выборке случайных порций мы получим данные только из небольшого подмножества больниц. Если разные больницы делают разные анализы или пользуются разными методиками, то данные могут оказаться сильно смещенными.

Разумеется, алгоритм завершится неудачно, если для выбранного нами метода не хватит памяти, оставшейся после размещения выборки. Если необходимо больше памяти, то можно читать выборку с диска на каждом проходе. Поскольку выборка гораздо меньше полного набора данных, мы все равно сэкономим на дисковом вводе-выводе по сравнению с ранее описанными алгоритмами.

## 6.4.2. Предотвращение ошибок в алгоритмах формирования выборки

Говоря о простом алгоритме из раздела 6.4.1, следует помнить об одной проблеме: нельзя предполагать, что он вернет все частые предметные наборы, присутствующие в полном файле, или что всякий возвращенный им набор является частым, если рассматривать его относительно полного файла. Предметный набор, который является частым во всем файле, но не в выборке, – ложноотрицательный результат, а набор, являющийся частым в выборке, но не во всем файле – ложноположительный.

Если выборка достаточно велика, то серьезных ошибок, скорее всего не будет. То есть предметный набор, поддержка которого много больше пороговой, почти наверняка будет найден и в случайной выборке, а предметный набор с поддержкой много меньше пороговой, вряд ли окажется частым в выборке. Однако набор, поддержка которого очень близка к пороговой, имеет одинаковые шансы оказаться или не оказаться частым в выборке.

Ложноположительные результаты можно устранить, пройдя по всему файлу и подсчитав все предметные наборы, которые были сочтены частыми при анализе выборки. Оставить нужно только наборы, которые являются частыми и в выборке, и в целом. Отметим, что ложноотрицательные результаты так обнаружить невозможно, потому что они не являются частыми предметными наборами в выборке.

Чтобы решить эту задачу за один проход, мы должны подсчитать частые предметные наборы всех размеров сразу, оставаясь в оперативной памяти. Если нам удалось выполнить простой алгоритм в памяти, то высоки шансы, что и подсчитать все частые предметные наборы мы сможем, потому что:

(*а*) скорее всего, среди частых наборов больше всего одноэлементных и двухэлементных, а нам уже приходилось их подсчитывать, и мы сумели сделать это за один проход;

(*б*) теперь в нашем распоряжении вся оперативная память, потому что хранить в ней выборку не нужно.

Мы не можем полностью устранить ложноотрицательные результаты, но можем уменьшить их количество, если объем оперативной памяти позволяет. Мы предложили установить для выборки пороговую поддержку *ps*, где *s* – исходная пороговая поддержка, а *p* – доля всего набора данных, включенная в выборку. Однако можно было бы взять немного меньшее значение, например 0.9*ps*. Раз порог меньше, то придется подсчитывать больше предметных наборов каждого размера, поэтому требования к объему памяти повышаются. С другой стороны, если памяти достаточно, то среди наборов, имеющих поддержку не ниже 0.9*ps* в выборке, окажутся почти все наборы с поддержкой не ниже *s* во всем файле. Если затем выполнить полный проход для исключения наборов данных, оказавшихся частыми в выборке, но не являющихся таковыми в целом, то не останется ни одного ложноположительного результата и, если повезет, ни одного или очень мало ложноотрицательных.

## 6.4.3. Алгоритм SON

В следующем варианте нам удастся избежать как ложноположительных, так и ложноотрицательных результатов, но ценой двух полных проходов. Этот алгоритм называется SON по первым буквам фамилий авторов (Savasere, Omiecinski, Navathe). Идея в том, чтобы разбить входной файл на порции (это может быть как «порция» в смысле распределенной файловой системы, так и просто часть файла). Рассмотрим каждую порцию как выборку и выполним для нее алгоритм из раздела 6.4.1. В качестве порога возьмем *ps*, где *p* – отношение размера выборки к размеру всего файла, а *s* – пороговая поддержка. Сохраним на диске все частые предметные наборы, найденные для каждой порции.

Обработав таким образом все порции, вычислим объединение всех предметных наборов, оказавшихся частыми в одной или нескольких выборках. Это будут *наборы-кандидаты*. Отметим, что если предметный набор не является частым ни в одной выборке, то его поддержка меньше *ps* в каждой порции. Поскольку число порций равно $1/p$, мы можем сделать вывод, что полная поддержка этого набора меньше $(1/p)ps = s$. Следовательно, любой предметный набор, являющийся частым во всем файле, обязан быть частым хотя бы в одной выборке, поэтому есть гарантия, что все истинные частые предметные наборы попали в число кандидатов, т. е. ложноотрицательных результатов не будет.

Мы выполнили один полный проход по данным, когда читали и обрабатывали каждую порцию. А на втором проходе мы подсчитываем все наборы-кандидаты и выбираем в качестве частых те из них, для которых поддержка не меньше *s*.

## 6.4.4. Алгоритм SON и MapReduce

Алгоритм SON хорошо адаптируется к параллельным вычислениям. Все порции можно обрабатывать параллельно, а частые предметные наборы из разных порций объединять для построения множества кандидатов. Затем кандидатов можно распределить между несколькими процессорами, поручить каждому процессору подсчет поддержки каждого кандидата относительно подмножества корзин и наконец, просуммировав получившиеся значения, получить полную поддержку наборов-кандидатов относительно всех корзин. Этот процесс необязательно реализовывать именно с помощью MapReduce, однако существует естественный способ выразить оба прохода в виде операций MapReduce. Ниже описывается весь каскад MapReduce-MapReduce.

- **Первая функция Map.** Взять назначенное подмножество корзин и найти частые предметные наборы в этом подмножестве, применяя алгоритм из раздела 6.4.1. Как было предложено, уменьшить пороговую поддержку с *s* до *ps*, где *p* – доля входного файла, назначенная одному распределителю. На выходе получаем множество пар (*F*, 1), где ключом *F* является частый предметный набор из выборки, а значение несущественно и всегда равно 1.
- **Первая функция Reduce.** Каждому редуктору назначается множество ключей, являющихся предметными наборами. Значение, входящее в пару, игнорируется, и редуктор просто порождает те ключи, которые встречаются не менее одного раза. Таким образом, выходом первой функции Reduce являются наборы-кандидаты.
- **Вторая функция Map.** На втором этапе распределители получают все выходы редукторов первого этапа (наборы кандидатов) и порцию входного файла данных. Каждый распределитель подсчитывает, сколько раз каждый набор-кандидат встречается в корзинах из назначенной ему порции данных. На выходе получается множество пар ключ-значение (*C*, *v*), где *C* – набор-кандидат, а *v* – его поддержка в корзинах, полученных данным распределителем.
- **Вторая функция Reduce.** Редукторы получают предметные наборы в качестве ключей и суммируют ассоциированные с ними значения. Результатом является полная поддержка каждого предметного набора, переданного редуктору для обработки. Те наборы, для которых сумма значений не меньше *s*, являются частыми относительно всего файла данных, поэтому редуктор выводит их вместе со счетчиками. Предметные наборы с полной поддержкой меньше *s*, не попадают в выход редуктора[3].

[3] Строго говоря, функция Reduce должна порождать значения для каждого ключа. Она может порождать 1 для предметных наборов, признанных частыми, и 0 для остальных.

### 6.4.5. Алгоритм Тойвонена

В этом алгоритме случайность используется не просто для формирования выборки, как в разделе 6.4.1. При наличии достаточного объема памяти алгоритм Тойвонена совершает один проход по небольшой выборке и полный проход по всем данным. В нем не бывает ни ложноположительных, ни ложноотрицательных результатов, но есть небольшая вероятность, что он вообще не даст никакого ответа. В таком случае его следует повторять, пока не будет получен ответ. Однако среднее число проходов до получения всех частых предметных наборов и никаких других – небольшая константа.

В начале алгоритм Тойвонена формирует небольшую выборку из входного файла и находит в ней частые предметные наборы-кандидаты. Делается это так же, как описано в разделе 6.4.1, однако порог должен быть установлен несколько меньше значения, пропорционального размеру выборки. То есть, если пороговая поддержка для всего набора данных равна $s$, а доля размера выборки – $p$, то для поиска частых предметных наборов в выборке следует установить порог где-то $0.9ps$ или $0.8ps$. Чем меньше порог, тем больше требуется оперативной памяти для вычисления всех частых предметных наборов относительно выборки, но тем больше вероятность избежать ситуации, когда алгоритм не дает ответа.

Построив множество частых предметных наборов для выборки, мы затем строим *негативную кайму* (negative border). Это множество предметных наборов, не являющихся частыми в выборке, но таких, что все их *непосредственные поднаборы* (построенные удалением ровно одного предмета) частые.

> **Пример 6.11.** Предположим, что имеются предметы {*A*,*B*,*C*,*D*,*E*}, и мы нашли в выборке такие частые предметные наборы: {*A*}, {*B*}, {*C*}, {*D*}, {*B*,*C*}, {*C*,*D*}. Отметим, что пустой набор ∅ тоже частый при условии, что число корзин не меньше пороговой поддержки, хотя в описанных выше алгоритмах мы этот очевидный факт опускали. Сразу видно, что набор {*E*} принадлежит негативной кайме, потому что не является частым в выборке, но его непосредственный поднабор ∅ частый.
>
> Наборы {*A*,*B*}, {*A*,*C*}, {*A*,*D*} и {*B*,*D*} тоже принадлежат негативной кайме. Ни один из них частым не является, и у каждого есть по два непосредственных поднабора, являющихся частыми. Например, у {*A*,*B*} два непосредственных поднабора {*A*} и {*B*}. Из остальных шести двухэлементных наборов ни один не принадлежит негативной кайме: наборы {*B*,*C*} и {*C*,*D*} – потому что частые, а остальные четыре содержат *E* и еще какой-то предмет и не могут быть частыми, потому что непосредственный поднабор {*E*} не частый.
>
> Ни один набор из трех и более предметов негативной кайме не принадлежит. Например, {*B*,*C*,*D*} не принадлежит, потому что его непосредственный поднабор {*B*,*D*} не частый. Таким образом, негативная кайма состоит из пяти наборов: {*E*}, {*A*,*B*}, {*A*,*C*}, {*A*,*D*} и {*B*,*D*}.

Для завершения алгоритма Тойвонена мы совершаем еще один проход по файлу и подсчитываем все предметные наборы, которые либо оказались частыми в выборке, либо принадлежат негативной кайме. Возможны два случая.

1. Ни один набор в негативной кайме не является частым во всем файле. В таком случае правильное множество частых предметных наборов в точности совпадает с предметными наборами в выборке, которые являются частыми и во всем файле.
2. Некоторые наборы в негативной кайме являются частыми во всем файле. Тогда нет уверенности, что не существует еще больших наборов – внутри негативной каймы или в множестве частых предметных наборов для выборки – которые являются также частыми в целом. Следовательно, в этот момент мы не можем дать ответ и должны повторить алгоритм с другой случайной выборкой.

## 6.4.6. Почему алгоритм Тойвонена работает

Понятно, что алгоритм Тойвонена никогда не дает ложноположительных результатов, поскольку считает частыми только наборы, которые были подсчитаны и оказались частыми во всем файле. Чтобы доказать, что он не дает и ложноотрицательных результатов, мы должны убедиться, что в случае, когда ни один набор внутри негативной каймы не является частым в целом, не может существовать ни одного предмета набора, который:

1. Является частым в целом, но
2. Не входит ни в негативную кайму, ни в множество частых предметных наборов для выборки.

Предположим противное. Тогда существует набор $S$, являющийся частым в целом, но не принадлежащий негативной кайме и не являющийся частым в выборке. Кроме того, в этом раунде алгоритм Тойвонена дал ответ, в котором $S$, очевидно, не входит в множество частых предметных наборов. В силу свойства монотонности все поднаборы $S$ также являются частыми в целом. Пусть $T$ – поднабор $S$ наименьшего размера среди всех тех поднаборов $S$, которые не являются частыми в выборке.

Мы утверждаем, что $T$ обязан принадлежать негативной кайме. Понятно, что $T$ удовлетворяет одному из условий принадлежности к ней: он не является частым в выборке. Но он удовлетворяет и второму условию: каждый его непосредственный поднабор является частым в выборке. Ибо если бы какой-то непосредственный поднабор не был частым в выборке, то существовал бы поднабор $S$, меньший $T$ и не являющийся частым в выборке, а это противоречит выбору $T$ как поднабору наименьшего размера среди поднаборов $S$, не являющихся частыми в выборке.

Итак, мы видим, что $T$ принадлежит негативной кайме и является частым во всем файле. Следовательно, этот раунд алгоритма не дал ответа.

## 6.4.7. Упражнения к разделу 6.4

**Упражнение 6.4.1.** Пусть имеется 8 предметов $A$,$B$, ..., $H$ и следующие максимальные частые предметные наборы: {$A$,$B$}, {$B$,$C$}, {$A$,$C$}, {$A$,$D$}, {$E$}, {$F$}. Найдите негативную кайму.

**Упражнение 6.4.2.** Примените алгоритм Тойвонена к данным из упражнения 6.3.1 с пороговой поддержкой 4. В качестве выборки возьмите первую строку корзин: {1, 2, 3}, {2, 3, 4}, {3, 4, 5}, {4, 5, 6}, т. е. треть файла. Пропорционально уменьшенный порог в этом случае будет равен 1.

(*а*) Какие предметные наборы являются частыми в выборке?

(*б*) Чему равна негативная кайма?

(*в*) Каков результат прохода по всему набору данных? Есть ли в негативной кайме предметные наборы, являющиеся частыми в целом?

**!! Упражнение 6.4.3.** Пусть предмет $i$ встречается ровно $s$ раз в файле, содержащем $n$ корзин, где $s$ – пороговая поддержка. Если взять выборку из $n/100$ корзин и уменьшить порог для выборки до $s/100$, то какова вероятность, что $i$ окажется частым? Можете предполагать, что $s$ и $n$ делятся на 100.

# 6.5. Подсчет частых предметных наборов в потоке

Предположим, что вместо файла имеется поток корзин, в котором требуется найти частые предметные наборы. Напомним, что разница между потоком и файлом состоит в том, что элементы потока доступны только в момент поступления, а темп поступления обычно настолько велик, что мы не можем сохранить весь поток, так чтобы к нему можно было обращаться с запросами. Кроме того, типичный поток эволюционирует со временем, поэтому предметные наборы, бывшие частыми в сегодняшнем потоке, могут перестать быть таковыми завтра.

Очевидное различие между потоками и файлами с точки зрения частых предметных наборов заключается в том, что у потока нет конца, поэтому рано или поздно поддержка набора превысит любой заданный порог при условии, что набор не перестает встречаться в потоке. Следовательно, в случае потоков мы должны считать пороговой поддержкой $s$ долю корзин, в которых должен встречаться набор, чтобы считаться частым. Но и при таком уточнении остается несколько вариантов, касающихся части потока, по которой измеряется эта доля.

В этом разделе мы обсудим несколько способов поиска частых предметных наборов в потоке. Сначала поговорим о том, как применить технику формирования выборки из предыдущего раздела. Затем рассмотрим модель затухающего окна из раздела 4.7 и обобщим метод, описанный в разделе 4.7.3, на поиск «популярных» предметов.

## 6.5.1. Методы выборки из потока

Далее мы будем предполагать, что элементами потока являются корзины с предметами. Пожалуй, самым простым подходом к вычислению текущей оценки частых предметных наборов в потоке является сохранение некоторого количества корзин в файле. Потом мы выполняем один из описанных в этой главе алгоритмов и, пока он работает, игнорируем вновь поступающие элементы или сохраняем их в

другом файле для анализа. Когда алгоритм закончится, у нас будет оценка частых предметных наборов в потоке. После этого есть несколько вариантов действий.

1. Мы можем использовать найденное множество в приложении, но сразу начать следующую итерацию выбранного алгоритма поиска частых предметных наборов. При этом можно либо:
   (*а*) воспользоваться файлом, сохраненным, пока работала предыдущая итерация, и одновременно начать запись в файл для следующей итерации;
   (*б*) начать запись в новый файл корзин сейчас и приступить к выполнению алгоритма, когда наберется достаточное количество корзин.
2. Мы можем продолжить подсчет числа вхождений каждого найденного частого предметного набора и общего количества встретившихся в потоке корзин с момент начала подсчета. Если какой-то предметный набор встречается в доле корзин, существенно меньшей порога *s*, то его можно исключить из множества частых наборов. При вычислении доли важно учитывать все корзины из первоначального файла, на основе которого были найдены частые предметные наборы. В противном случае есть риск наткнуться на короткий период времени, в течение которого истинно частый набор встречается недостаточно часто, и выкинуть его. Следует также предусмотреть какой-то способ добавления новых частых наборов в существующее множество. Упомянем две возможности.
   (*а*) Периодически сохранять новый участок потока в файле и прогонять очередную итерацию выбранного алгоритма поиска частых предметных наборов. Новое множество частых наборов формируется из результата этой итерации и тех элементов предыдущего множества, которые сохранились после исключения наборов, признанных нечастыми.
   (*б*) Добавлять в текущее множество случайные предметные наборы и в течение некоторого времени подсчитывать долю их вхождений, пока не составится надежное представление о том, частые они или нет. Можно выбирать новые наборы не совсем случайно, а ориентируясь на предметы, которые встречаются во многих наборах, уже признанных частыми. Например, выбирать новые наборы из негативной каймы (см. раздел 6.4.5) текущего множества частых наборов.

## 6.5.2. Частые предметные наборы в затухающих окнах

Напомним (см. раздел 4.7), что затухающее окно потока образуется путем выбора небольшой константы $c$ и назначения $i$-му элементу, считая назад от последнего поступившего, веса $(1 - c)^i$, или приближенно $e^{-ci}$. В разделе 4.7.3 был фактически описан метод вычисления частых предметов при условии, что пороговая поддерж-

ка определена несколько иначе. Точнее, мы рассматривали для каждого предмета поток, в котором 1 означает, что предмет встретился в некоторой позиции потока, а 0 – что не встретился. Мы определили «оценку» предмета как сумму весов элементов, равных 1. Мы были вынуждены запоминать все предметы с оценкой не ниже 1/2. Мы не можем использовать пороговую оценку больше 1, потому что не инициализируем счетчик предмета, пока тот не встретится в потоке, а при первом его появлении оценка равна всего 1 (поскольку $1 = (1 - c)^0$ – вес текущего предмета).

Чтобы адаптировать этот метод к потоку корзин, необходимо внести два изменения. Первое простое. Элементами потока являются корзины, а не отдельные предметы, поэтому в одном элементе может быть много предметов. Каждый из них следует считать «текущим» предметом и прибавлять 1 к их оценке после умножения всех текущих оценок на $1 - c$, как описано в разделе 4.7.3. Если у каких-то предметов в корзине еще нет текущей оценки, инициализировать ее, положив равной 1.

Второе изменение посложнее. Мы хотим найти все частые предметные наборы, а не только одноэлементные. Если бы мы инициализировали счетчик для каждого набора, увидев его впервые, то счетчиков оказалось бы слишком много. Например, для корзины с 20 предметами существует свыше миллиона поднаборов, и все их пришлось бы инициализировать. И это только для одной корзины. С другой стороны, как уже отмечалось, если задать пороговую оценку для инициализации набора выше 1, то мы не смогли бы инициализировать ни один набор, и метод оказался бы никуда не годен.

Решить эту проблему можно, если начинать оценивание некоторых предметных наборов при первом появлении экземпляра, но проявить осторожность в части того, для каких наборов это делать. Можно позаимствовать идею из алгоритма Apriori и начинать оценивание набора $I$, только если все его непосредственные поднаборы уже оцениваются. Из этого ограничения следует, что если $I$ – действительно частый набор, то в конце концов мы начнем его подсчитывать, но никогда не будем заниматься набором, если он не является хотя бы кандидатом в смысле алгоритма Apriori.

**Пример 6.12.** Предположим, что $I$ – частый предметный набор, но встречается в потоке периодически, один раз на каждые $2/c$ корзин. Тогда его оценка и оценка всех его поднаборов никогда не опустится ниже $e^{-1/2}$, а это число больше 1/2. Следовательно, если для некоторого поднабора $I$ оценка создана, то он будет оцениваться вечно. При первом появлении $I$ будут созданы только оценки для его одноэлементных поднаборов. Однако при следующем появлении начнут оцениваться все его двухэлементные поднаборы, потому что каждый их непосредственный поднабор уже оценивается. Аналогично, когда $I$ появится в $k$-ый раз, его поднаборы размера $k - 1$ уже оцениваются, поэтому мы инициализируем оценки для всех поднаборов размера $k$. Рано или поздно мы достигнем размера $|I|$, и в этот момент начнем оценивать сам набор $I$.

## 6.5.3. Гибридные методы

Подход, предложенный в разделе 6.5.2, имеет ряд достоинств. Он требует ограниченного объема работы при поступлении каждого нового элемента и всегда дает актуальную картину частых наборов в затухающем окне. Его основной недостаток заключается в том, что приходится помнить оценки для всех предметных наборов с оценкой не ниже 1/2. Ограничить число оцениваемых наборов можно, увеличив значение параметра $c$. Но чем больше $c$, тем меньше затухающее окно. Следовательно, мы будем вынуждены реагировать на локальные флуктуации частоты, вместо того чтобы накапливать информацию за длительный период.

Идеи из разделов 6.5.1 и 6.5.2 можно объединить. Например, можно было бы выполнить стандартный алгоритм поиска частых предметных наборов для выборки из потока с обычной пороговой поддержкой. Наборы, признанные этим алгоритмом частыми, будут рассматриваться, как будто все они поступили в текущий момент. Таким образом, они получат оценку, равную фиксированной доле своего счетчика.

Точнее, пусть $b$ – количество корзин в первоначальной выборке корзин, $c$ – коэффициент затухания, а $s$ – минимальная оценка, при которой предметный набор в затухающем окне считается частым. Тогда пороговая поддержка для начального прогона алгоритма поиска частых наборов равна $bcs$. Если набор $I$ имеет в выборке поддержку $t$, то ему назначается первоначальная оценка $t/(bc)$.

> **Пример 6.13.** Пусть $c = 10^{-6}$ и минимальная оценка в затухающем окне принята равной 10. Предположим, что мы берем из потока выборку объемом $10^8$ корзин. Тогда при анализе этой выборки будет применяться пороговая поддержка $10^8 \times 10^{-6} \times 10 = 1000$.
>
> Рассмотрим предметный набор $I$ с поддержкой 2000 в выборке. Тогда его начальная оценка будет равна $2000/(10^8 \times 10^{-6}) = 20$. После шага инициализации при получении каждой новой корзины из потока текущая оценка будет умножаться на $1 - c = 0.999999$. Если $I$ встречается в текущей корзине, то к его оценке прибавляется 1. Если оценка набора $I$ опускается ниже 10, то он больше не считается частым и исключается из множества частых наборов.

К сожалению, мы не знаем разумного способа инициализировать оценку новых предметных наборов. Если для набора $I$ нет оценки, и мы хотим, чтобы минимальная оценка была равна 10, то никаким образом оценка одиночной корзины не может скачком подняться от 0 до значения, большего 1. Наилучшая стратегия добавления новых наборов – повторно произвести поиск частых предметных наборов в выборке из потока и добавить в множество уже оцениваемых наборов такие, которые в нем пока отсутствуют, но имеют оценку выше пороговой.

## 6.5.4. Упражнения к разделу 6.5

**!! Упражнение 6.5.1.** Предположим, что подсчитываются частые предметные наборы в затухающем окне с коэффициентом затухания $c$. Предположим также,

что с вероятностью $p$ данный элемент потока (корзина) содержит оба предмета $i$ и $j$. Кроме того, с вероятностью $p$ корзина содержит $i$, но не содержит $j$, и с такой же вероятностью $p$ содержит $j$, но не содержит $i$. Выразите в виде функции от $c$ и $p$ долю времени, в течение которого у пары $\{i, j\}$ будет существовать оценка.

# 6.6. Резюме

- *Модель корзины покупок.* В этой модели данных рассматриваются две сущности: предметы и корзины, между которыми существует связь многие-ко-многим. Типичная корзина связана с небольшим числом предметов, тогда как каждый предмет может находиться во многих корзинах.
- *Частые предметные наборы.* Поддержкой набора предметов называется количество корзин, в которых все эти предметы встречаются одновременно. Предметные наборы с поддержкой не ниже пороговой называются частыми.
- *Ассоциативные правила.* Так называются импликации, означающие, что если корзина содержит некоторый набор предметов $I$, то с большой вероятностью она содержит также некий предмет $j$. Вероятность того, что $j$ присутствует в корзине, содержащей $I$, называется достоверностью правила. Интересностью правила называется отклонение достоверности от доли всех корзин, содержащих $j$.
- *Узкое место подсчета пар.* Чтобы найти частые предметные наборы, мы должны просмотреть все корзины и подсчитать, сколько раз встречаются наборы определенного размера. В типичном случае, когда ставится цель найти небольшое количество наборов, встречающихся чаще всего, память в основном расходуется на хранение счетчиков пар предметов. Поэтому любой метод поиска частых предметных наборов стремится оптимизировать объем памяти, необходимой для подсчета пар.
- *Треугольные матрицы.* Можно было бы использовать для подсчета пар двумерный массив, но при этом впустую растрачивается половина занятой им памяти, поскольку бессмысленно подсчитывать одну и ту же пару $\{i, j\}$ в обоих элементах $i$-$j$ и $j$-$i$. Если лексикографически упорядочить пары $(i, j)$, так чтобы было $i < j$, то можно хранить счетчики в одномерном массиве без потери памяти и при этом обеспечить к ним эффективный доступ.
- *Хранение счетчиков пары в виде троек.* Если в корзинах реально встречается меньше 1/3 потенциально возможных пар, то с точки зрения расхода памяти эффективнее хранить счетчики пар в виде троек $(i, j, c)$, где $c$ – счетчик пары $\{i, j\}$ и $i < j$. Индексная структура типа хэш-таблицы позволит эффективно находить счетчик для пары $(i, j)$.
- *Монотонность частых предметных наборов.* У предметных наборов есть важное свойство: если набор частый, то таковы же и все его поднаборы. Мы

пользуемся контрапозицией этого свойства – если набор не частый, то все его наднаборы тоже нечастые – чтобы не подсчитывать некоторые предметные наборы.

- *Алгоритм Apriori для пар*. Найти все частые пары можно за два прохода по корзинам. На первом проходе мы подсчитываем сами предметы и определяем, какие из них частые. На втором проходе подсчитываются только пары частых предметов. Свойство монотонности позволяет игнорировать все остальные пары.
- *Поиск более крупных частых предметных наборов*. Алгоритм Apriori и многие другие позволяют находить наборы из трех и более предметов. Для этого необходим отдельный проход для каждого размера. При поиске частых наборов размера $k$ в силу свойства монотонности достаточно просматривать только такие, для которых все поднаборы размера $k - 1$ уже признаны частыми.
- *Алгоритм PCY*. Это усовершенствование алгоритма Apriori, достигаемое путем создания на первом проходе хэш-таблицы в той части оперативной памяти, которая не используется для подсчета предметов. Пары предметов хэшируются, и ячейки хэш-таблицы используются как целочисленные счетчики количества отображенных на эту ячейку пар. Затем, на втором проходе необходимо подсчитывать только те пары частых предметов, которые были хэшированы в частую ячейку (со счетчиком, не меньшим пороговой поддержки) на первом проходе.
- *Многоэтапный алгоритм*. Мы можем включить дополнительные проходы между первым и вторым проходами алгоритма PCY и хэшировать пары в другие независимые хэш-таблицы. На каждом промежуточном проходе нужно хэшировать только пары частых предметов, которые были хэшированы в частые ячейки на всех предыдущих проходах.
- *Многохэшевый алгоритм*. Первый проход алгоритма PCY можно модифицировать, разделив доступную оперативную память на несколько хэш-таблиц. На втором проходе нужно подсчитывать только те пары частых предметов, которые были хэшированы в частые ячейки во всех хэш-таблицах.
- *Рандомизированные алгоритмы*. Вместо того чтобы просматривать все данные, мы можем сформировать случайную выборку корзин, настолько малую, чтобы в оперативной памяти помещалась и сама выборка, и счетчики наборов. Пороговую поддержку следует пропорционально уменьшить. Затем можно найти частые наборы в выборке и надеяться, что они дают хорошее представление данных в целом. Хотя для этого метода требуется не более одного прохода по всему файлу данных, он может давать как ложноположительные (предметные наборы, частые в выборке, но не в целом), так и ложноотрицательные (предметные наборы, частые в целом, но не в выборке) результаты.
- *Алгоритм SON*. Простой рандомизированный алгоритм можно улучшить, если разбить весь файл корзин на участки настолько малые, что все частые

предметные наборы, встречающиеся в одном участке, можно найти, оставаясь в оперативной памяти. Кандидатами считаются наборы, оказавшиеся частыми хотя бы в одном участке. На втором проходе мы подсчитываем всех кандидатов и находим точное множество частых предметных наборов. Этот алгоритм особенно хорошо приспособлен для распараллеливания средствами MapReduce.

- *Алгоритм Тойвонена*. Этот алгоритм начинается с поиска частых предметных наборов в выборке, причем пороговая поддержка снижена, так что шансы пропустить набор, являющийся частым в целом, невелики. Затем просматривается весь файл корзин и подсчитываются наборы, частые не только в выборке, но и в негативной кайме, т. е. такие, которые сами частыми не являются, но все их непосредственные поднаборы частые. Если ни один набор в негативной кайме не является частым в целом, то ответ точный. Если же в негативной кайме найден хотя бы один частый набор, то весь процесс придется повторить на другой выборке.
- *Частые предметные наборы в потоках*. Если использовать затухающее окно с коэффициентом $c$, то можно начинать подсчет предмета, впервые встретив его в какой-то корзине. Подсчет предметного набора начинается, если мы видим, что он встречается в текущей корзине и все его непосредственные поднаборы уже подсчитываются. Поскольку окно затухающее, мы умножаем все счетчики на $1 - c$ и исключаем те, для которых получился результат меньше $1/2$.

# 6.7. Список литературы

Модель корзины покупок, а также ассоциативные правила и алгоритм Apriori описаны в работах [1] и [2].

Алгоритм PCY взят из [4]. Описание многоэтапного и многохэшевого алгоритмов можно найти в [3].

Алгоритм SON заимствован из [5], а алгоритм Тойвонена впервые описан в [6].

1. R. Agrawal, T. Imielinski, A. Swami «Mining associations between sets of items in massive databases», Proc. ACM SIGMOD Intl. Conf. on Management of Data, pp. 207–216, 1993.
2. R. Agrawal, R. Srikant «Fast algorithms for mining association rules», Intl. Conf. on Very Large Databases, pp. 487–499, 1994.
3. M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, J.D. Ullman «Computing iceberg queries efficiently», Intl. Conf. on Very Large Databases, pp. 299-310, 1998.
4. J. S. Park, M.-S. Chen, P.S. Yu «An effective hash-based algorithm for mining association rules», Proc. ACM SIGMOD Intl. Conf. on Management of Data, pp. 175–186, 1995.

5. A. Savasere, E. Omiecinski, S.B. Navathe «An efficient algorithm for mining association rules in large databases», Intl. Conf. on Very Large Databases, pp. 432–444, 1995.
6. H. Toivonen «Sampling large databases for association rules», Intl. Conf. on Very Large Databases, pp. 134–145, 1996.

# ГЛАВА 7. Кластеризация

Кластеризация – это процесс исследования множества «точек» и их группировки в «кластеры» согласно некоторой метрике. Идея в том, что точки, попавшие в один кластер, находятся недалеко друг от друга, а точки из разных кластеров разделены большим расстоянием. На рис. 1.1 было показано, как могут выглядеть кластеры. На нем три кластера расположились вокруг трех разных перекрестков, а еще два смешались между собой, потому что разделены недостаточно точно.

В этой главе мы будем рассматривать методы выявления кластеров в данных. Особенно нас будут интересовать ситуации, когда набор данных очень велик либо находится в пространстве высокой размерности или вообще неевклидовом. Поэтому мы обсудим несколько алгоритмов, в которых предполагается, что данные не помещаются в оперативной памяти. Но начнем с простого: общих подходов к кластеризации и методов работы с кластерами в неевклидовом пространстве.

## 7.1. Введение в методы кластеризации

Начнем с понятий пространства и метрики. Мы определим два основных подхода к кластеризации: иерархический и отнесения точек. Затем обсудим «проклятие размерности», которое затрудняет кластеризацию в многомерных пространствах, но, как мы увидим, при правильном использовании позволяет сделать некоторые упрощения.

### 7.1.1. Точки, пространства и расстояния

Набор данных, подходящий для кластеризации, – это множество *точек*, т. е. объектов, принадлежащих некоторому *пространству*. В самом общем случае пространство – это просто универсальное множество точек. Но более типичен случай евклидова пространства (см. раздел 3.5.2), обладающего рядом важных свойств, полезных для кластеризации. В частности, точками евклидова пространства являются векторы вещественных чисел. Размерность вектора равна числу измерений пространства. Элементы вектора обычно называют *координатами* представленной им точки.

Любое пространство, в котором возможна кластеризация, обладает метрикой, определяющей расстояние между любыми двумя точками. С метриками мы познакомились в разделе 3.5. В евклидовых пространствах обычно применяется евклидова метрика (квадратный корень из суммы квадратов разностей координат точек по каждому измерению), но мы упоминали и некоторые другие, в т. ч. манхэттенское расстояние (сумму абсолютных величин разностей координат по каждому измерению) и $L_\infty$-расстояние (максимум абсолютных величин разностей координат по каждому измерению).

**Пример 7.1.** Классические применения кластеризации связаны с пространствами низкой размерности. На рис. 7.1 показаны результаты измерений роста и веса собак разных пород. Не зная заранее, какой породы собака, мы можем, посмотрев на рисунок, увидеть, что каждой из трех пород соответствует свой кластер. Если данных немного, то любой алгоритм кластеризации правильно выявит кластеры, да и простого нанесения точек на график и его визуального изучения бывает достаточно.

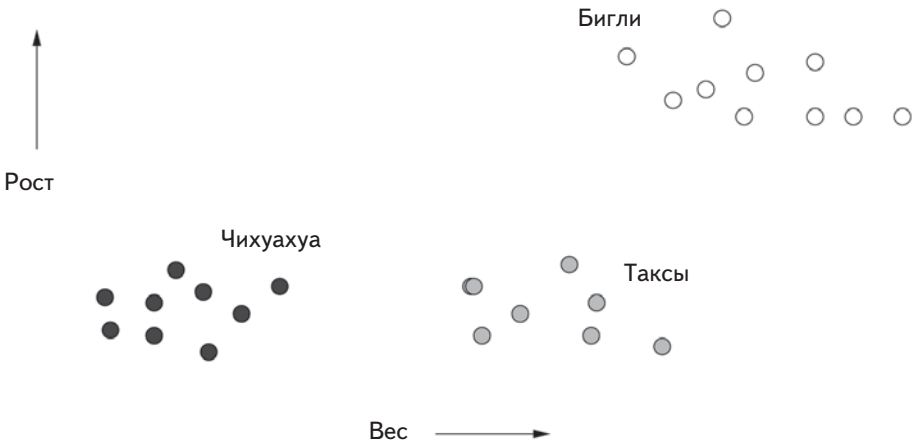



**Рис. 7.1.** Рост и вес собак трех пород

Но современные задачи кластеризации не так просты. В них встречаются евклидовы пространства очень высокой размерности, а то и вовсе неевклидовы. Например, трудной проблемой является кластеризация документов по тематике на основе частого вхождения в них необычных слов. Или проблема кластеризации киноманов по типам фильмов, которые им нравятся.

В разделе 3.5 мы рассматривали метрики для неевклидовых пространств, в т. ч. расстояние Жаккара, косинусное расстояние, расстояние Хэмминга и редакционное расстояние. Напомним, какие требования предъявляются к функции, претендующей называться расстоянием между двумя точками:

1. Расстояние неотрицательно и равно 0, только если точки совпадают.
2. Расстояние симметрично: расстояние от $x$ до $y$ равно расстоянию от $y$ до $x$.

3. Расстояние удовлетворяет неравенству треугольника: расстояние от $x$ до $z$ не больше суммы расстояний от $x$ до $y$ и от $y$ до $z$.

## 7.1.2. Стратегии кластеризации

Мы можем разделить (кластеризовать!) алгоритмы кластеризации на две группы с принципиально разными стратегиями.

1. *Иерархические*, или *агломеративные* алгоритмы сначала помещают каждую точку в отдельный кластер. Затем «близкие» кластеры объединяются с использованием одного из многих определений «близости». Процесс объединения прекращается, когда дальнейшее объединение приводит к кластерам, которыми по каким-то причинам нежелательны. Например, мы можем остановиться, получив заранее заданное число кластеров, или использовать какую-то меру компактности кластеров и отказаться объединять два кластера, если в результате получится кластер, точки которого разбросаны по слишком большой области.
2. Другой класс алгоритмов основан на *отнесении точек*. Точки рассматриваются в определенном порядке, и каждая относится к наиболее подходящему ей кластеру. Обычно этому процессу предшествует короткий этап, на котором оцениваются начальные кластеры. В зависимости от алгоритма допускается как объединение, так и разбиение кластеров. Кроме того, допускаются изолированные *выбросы* (точки, далеко отстоящие от любого из уже имеющихся кластеров).

Алгоритмы кластеризации можно классифицировать по следующим признакам.

(*а*) Предполагается ли, что пространство евклидово, или алгоритм может работать в произвольном метрическом пространстве. Мы увидим, что ключевое отличие евклидова пространства – возможность использовать в качестве обобщенного представления множества точек его *центроид* – среднюю точку. В неевклидовом пространстве не существует понятия центроида, поэтому приходится придумывать другие способы обобщенного представления кластера.

(*б*) Предполагает ли алгоритм, что данные помещаются в оперативную память, или данные должны находиться, в основном, во внешней памяти. Для работы с большими наборами данными алгоритмы должны пускаться на ухищрения, например, потому что невозможно проанализировать каждую пару точек. Кроме того, необходимо строить обобщенное представление кластера в оперативной памяти, потому что одновременно хранить в памяти все его точки не получается.

## 7.1.3. Проклятие размерности

Многомерные евклидовы пространства обладают рядом интуитивно неочевидных свойств, которые иногда называют «проклятием размерности». Эти аномалии ха-

рактерны и для неевклидовых пространств. Одно из проявлений «проклятия» – тот факт, что при большом числе измерений почти все пары точек находятся на одинаковом расстоянии друг от друга. Другое проявление – почти любые два вектора почти ортогональны. Рассмотрим эти явления по очереди.

## Распределение расстояний в многомерном пространстве

Рассмотрим $d$-мерное евклидово пространство. Выберем случайным образом $n$ точек в единичном кубе, т. е. такие точки $[x_1, x_2, ..., x_d]$, что все координаты $x_i$ принадлежат диапазону от 0 до 1. Если $d = 1$, то случайные точки располагаются на отрезке прямой длины 1. Мы ожидаем, что некоторые точки окажутся очень близко друг к другу, например, соседние точки внутри отрезка. Мы также ожидаем, что расстояния между некоторыми точками будут велики, например, между точками у противоположных концов отрезка. Среднее расстояние между точками равно $1/3$[1].

Пусть $d$ очень велико. Евклидово расстояние между случайными точками $[x_1, x_2, ..., x_d]$ и $[y_1, y_2, ..., y_d]$ равно

$$\sqrt{\sum_{i=1}^{d}(x_1 - y_1)^2}$$

Здесь $x_i$ и $y_i$ – случайные переменные, равномерно распределенные в диапазоне от 0 до 1. Поскольку $d$ велико, можно ожидать, что для некоторого $i$ величина $|x_i - y_i|$ будет близка к 1. Таким образом, расстояние между почти любыми двумя случайными точками ограничено снизу 1. На самом деле, можно усилить оценку нижней границы расстояния для всех пар точек, кроме пренебрежимо малой доли. Однако максимальное расстояние между любыми двумя точками равно $\sqrt{d}$, и можно доказать, что почти для всех пар точек расстояние не приближается к этой верхней границе. В действительности расстояние между почти любой парой точек близко к среднему.

Но если не существует близких пар точек, то трудно вообще построить какой-нибудь кластер. Нет никаких доводов в пользу включения в группу именно этой, а не какой-то другой пары точек. Конечно, данные не всегда случайны и даже в пространствах очень высокой размерности могут существовать полезные кластеры. Но из рассуждения о случайных данных следует, что будет чрезвычайно трудно найти такие кластеры среди столь большого числа пар с примерно одинаковым расстоянием.

## Углы между векторами

Снова рассмотрим три случайные точки $A$, $B$, $C$ в $d$-мерном пространстве, где $d$ велико. Мы не предполагаем, что точки находятся в единичном кубе. Что такое угол $ABC$? Будем считать, что $A$ имеет координаты $[x_1, x_2, ..., x_d]$, $C$ – координаты

[1] Это можно доказать, вычислив двойной интеграл, но мы опустим математику, потому что не она является темой обсуждения.

$[y_1, y_2, ..., y_d]$, а $B$ – начало координат. Напомним (см. раздел 3.5.4), что косинус угла $ABC$ равен скалярному произведению векторов $A$ и $C$, поделенному на произведение их длин:

$$\frac{\sum_{i=1}^{d} x_1 y_1}{\sqrt{\sum_{i=1}^{d} x_1^2}\sqrt{\sum_{i=1}^{d} y_1^2}}$$

С ростом $d$ знаменатель линейно возрастает, но числитель является суммой случайных величин, которые с равной вероятностью могут быть как отрицательными, так и положительными. Поэтому математическое ожидание числителя равно 0 и с ростом $d$ его стандартное отклонение растет только как $\sqrt{d}$. Следовательно, для больших $d$ косинус угла между любыми двумя векторами почти наверняка близок к 0, т. е. угол близок к 90 градусам.

Важным следствием из ортогональности случайных векторов является тот факт, что если имеются три случайные точки $A$, $B$, $C$ и известно, что расстояние от $A$ до $B$ равно $d_1$, а расстояние от $B$ до $C$ равно $d_2$, то можно предполагать, что расстояние от $A$ до $C$ приближенно равно $\sqrt{d_1^2 + d_2^2}$. Это неверно, даже приближенно, если число измерений мало. В предельном случае, когда $d = 1$, расстояние от $A$ до $C$ равно $d_1 + d_2$, если $A$ и $C$ лежат по разные стороны от $B$, и $|d_1 - d_2|$ – если по одну сторону.

### 7.1.4. Упражнения к разделу 7.1

**! Упражнение 7.1.1.** Докажите, что математическое ожидание расстояния между двумя случайными независимыми и равномерно распределенными точками отрезка прямой длиной 1 равно 1/3.

**!! Упражнение 7.1.2.** Каково математическое ожидание евклидова расстояния между двумя случайными независимыми и равномерно распределенными точками единичного квадрата?

**! Упражнение 7.1.3.** Пусть имеется $d$-мерное евклидово пространство. Рассмотрим векторы, элементы которых могут принимать только значения +1 и –1. Заметим, что длина каждого вектора равна $\sqrt{d}$, поэтому произведение их длин (знаменатель в формуле косинуса угла между ними) равно $d$. Если выбирать все элементы независимо и каждый может с одинаковой вероятностью быть равен +1 или –1, то каково распределение значений числителя в этой формуле (т. е. суммы попарных произведений соответственных элементов векторов)? Что можно сказать о математическом ожидании косинуса угла между векторами при росте $d$?

## 7.2. Иерархическая кластеризация

Начнем с рассмотрения иерархической кластеризации в евклидовом пространстве. Описанный ниже алгоритм годится только для относительно небольших

наборов данных, но даже в этом случае аккуратная реализация может повысить его эффективность. Если пространство неевклидово, то иерархическая кластеризация наталкивается на дополнительные проблемы. Поэтому мы рассмотрим «кластроиды» и способ представления кластера, когда невозможно определить его центроид, или среднюю точку.

## 7.2.1. Иерархическая кластеризация в евклидовом пространстве

Любой алгоритм иерархической кластеризации работает следующим образом. В начале помещаем каждую точку в отдельный кластер. Затем строим более крупные кластеры, объединяя два меньших. При этом нужно заранее решить:

1. Как представлять кластеры?
2. Как выбирать два кластера для объединения?
3. Когда прекращать объединение кластеров?

Имея ответы на эти вопросы, мы можем описать алгоритм:

```
WHILE не пришло время остановиться DO
    выбрать для объединения два наилучших кластера;
    объединить эти кластеры в один;
END;
```

Для начала предположим, что пространство евклидово. Это позволит представить кластер центроидом, или средней точкой. Отметим, что в кластере из одной точки эта точка и является центроидом, поэтому инициализация кластеров не представляет труда. Выберем такое правило объединения: определим расстояние между кластерами как расстояние между их центроидами и на каждом шаге будем выбирать кластеры с наименьшим расстоянием. Есть и другие определения межкластерного расстояния, да и наилучшую пару кластеров можно выбирать не только на основе расстояния. Некоторые варианты мы обсудим в разделе 7.2.3.

**Пример 7.2.** Рассмотрим, как работает иерархическая кластеризация на примере рис. 7.2. Здесь точки расположены в двумерном евклидовом пространстве, и каждая точка обозначена своими координатами ($x$, $y$). В начальный момент каждая точка – отдельный кластер, в котором сама она является центроидом. Среди всех пар точек минимальные расстояния у следующих: (10,5) и (11,4) (11,4) и (12,3). Расстояние между точками в каждой паре равно $\sqrt{2}$. Выберем произвольную пару, например (11,4) и (12,3), и объединим ее в один кластер. На рис. 7.3 показан результат и в том числе центроид нового кластера – точка (11.5, 3.5).

Можно было бы подумать, что на следующем шаге точка (10,5) объединяется с новым кластером, поскольку она близка к точке (11,4). Но наше правило требует сравнивать только центроиды кластеров, а расстояние от точки (10,5) до центроида нового кластера равно $1.5\sqrt{2}$, что немного больше 2. Таким образом, теперь двумя ближайшими кластерами являются точки (4,8) и (4,10). Объединяем их в новый кластер с центроидом в точке (4,9).

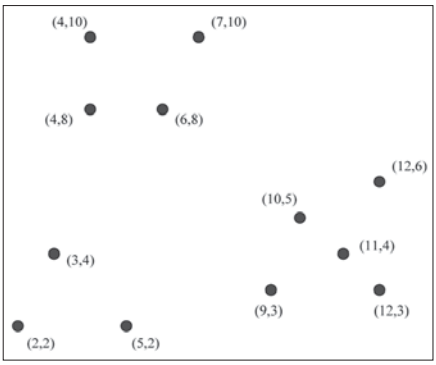



**Рис. 7.2.** Двенадцать точек, подлежащих иерархической кластеризации

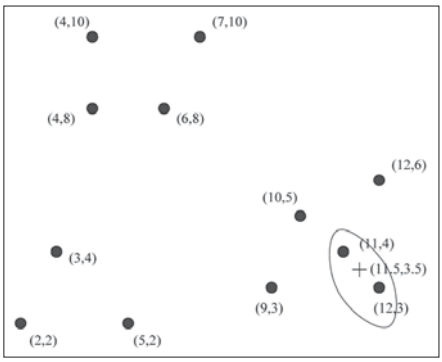



**Рис. 7.3.** Первые две точки объединяются в кластер

Теперь двумя ближайшими центроидами являются точки (10,5) и (11.5,3.5), поэтому объединяем эти два кластера. В результате получается кластер из трех точек (10,5), (11,4), (12,3). Центроид этого кластера – точка (11,4), которая случайно входит в кластер. Новое состояние показано на рис. 7.4.

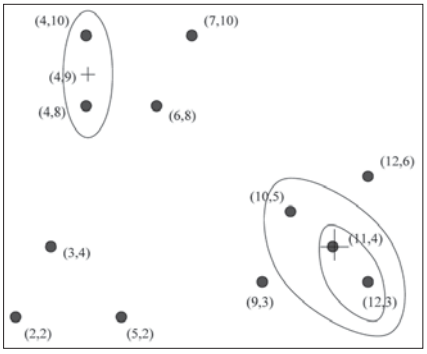



**Рис. 7.4.** Результат кластеризации после еще двух шагов

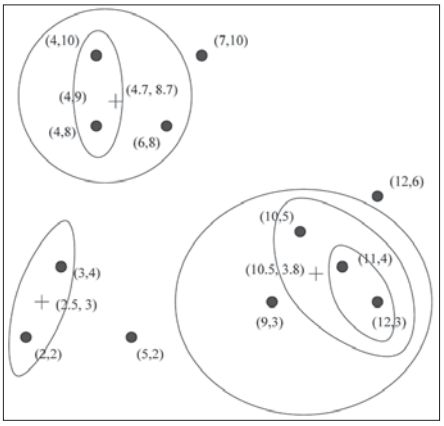



**Рис. 7.5.** Результат иерархической кластеризации после еще трех шагов

Теперь у нас есть несколько пар центроидов на расстоянии $\sqrt{5}$ друг от друга, и это минимальное расстояние. На рис. 7.5 показано, что получается при одном из способов их объединения:

1. Точка (6,8) объединена с кластером из двух элементов с центроидом в точке (4,9).
2. Точка (2,2) объединена с (3,4).
3. Точка (9,3) объединена с кластером из трех элементов с центроидом в точке (11,4).

Мы можем и дальше объединять кластеры. Далее обсудим правила остановки.

Есть несколько подходов к принятию решения об остановке процесса кластеризации.

1. Возможно, нам сообщили или мы сами предполагаем, сколько кластеров может быть в данных. Например, если известно, что данные о породах собак собирались для чихуахуа, такс и биглей, то останавливаться надо, когда останется всего три кластера.
2. Можно прекратить процесс, если в какой-то момент объединение уже существующих кластеров дает неприемлемый результат. Различные критерии приемлемости мы обсудим в разделе 7.2.3, но, например, можно было бы потребовать, чтобы среднее расстояние между центроидом и точками кластера не превышало некоторого порога. Такой подход имеет смысл, только когда есть основания полагать, что ни один кластер не занимает слишком большую область.
3. Можно продолжать кластеризацию, пока не останется всего один кластер. Однако бессмысленно возвращать единственный кластер, содержащий все точки. Вместо этого мы возвращаем дерево, показывающее, как объединялись точки. Такая форма ответа полезна в некоторых приложениях, например, когда точками являются геномы разных видов, а расстояние отражает различие геномов[2]. В этом случае дерево описывает эволюцию видов, т. е. вероятный порядок происхождения двух видов от общего предка.

**Пример 7.3.** Дерево, получающееся после доведения до конца кластеризации данных на рис. 7.2, показано на рис. 7.6.

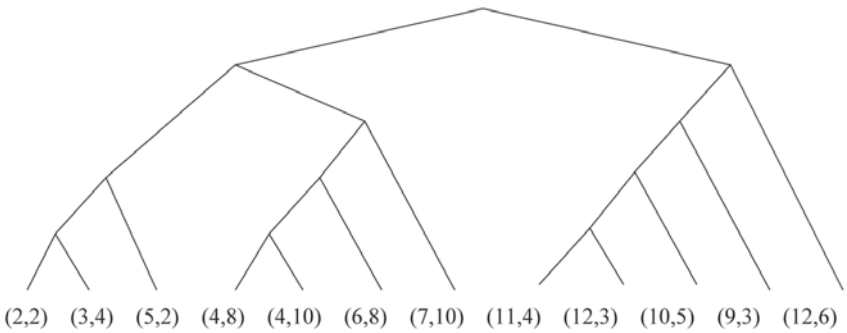



**Рис. 7.6.** Дерево, отражающее порядок группировки точек на рис. 7.2

## 7.2.2. Эффективность иерархической кластеризации

Базовый алгоритм иерархической кластеризации не очень эффективен. На каждом шаге мы должны вычислять расстояния между каждой парой кластеров, что-

[2] Это пространство, конечно, неевклидово, но принцип иерархической кластеризации распространяется – с некоторыми модификациями – и на неевклидовы пространства.

бы выбрать наилучшую для объединения пару. Начальный шаг требует времени $O(n^2)$, а последующие – времени, пропорционального $(n-1)^2$, $(n-2)^2$, ... . Сумма квадратов чисел от 1 до $n$ пропорциональна $O(n^3)$, т. е. это кубический алгоритм. Поэтому он годится только для относительно небольшого числа точек.

Однако существует и более эффективная реализация, о которой следует знать.

1. Начинаем по необходимости с вычисления расстояний между всеми парами точек. Этот шаг требует времени $O(n^2)$.
2. Помещаем пары и расстояния между ними в очередь с приоритетами, чтобы можно было находить наименьшее расстояние за один шаг. Эта операция также требует времени $O(n^2)$.
3. Решив, что нужно объединить кластеры $C$ и $D$, мы удаляем из очереди все элементы, относящиеся к этим кластерам; для этого требуется время $O(n \log n)$, поскольку всего необходимо сделать не более $2n$ удалений, а удаление из очереди с приоритетами можно выполнить за время $O(\log n)$.
4. Затем вычисляем расстояния между новым кластером и всеми остальными. На это требуется время $O(n \log n)$, поскольку нужно будет вставить в очередь не более $n$ элементов, а вставка в очередь с приоритетами также выполняется за время $O(\log n)$.

Поскольку последние два шага выполняются самое большее $n$ раз, а первые два только по разу, то общее время работы алгоритма составляет $O(n^2 \log n)$. Это лучше, чем $O(n^3)$, но все же налагает сильные ограничения на величину $n$, при которой кластеризация становится практически реализуемой.

## 7.2.3. Альтернативные правила управления иерархической кластеризацией

Мы рассмотрели одно правило выбора наилучших для объединения кластеров: найти пару кластеров с минимальным расстоянием между центроидами. Но есть и другие варианты.

1. Считать, что расстоянием между кластерами является минимум расстояний между любыми двумя точками, по одной из каждого кластера. Например, на рис. 7.3 мы на следующем шаге объединили бы точку (10,5) с кластером из двух точек, потому что для точки (10,5) указанное расстояние равно $\sqrt{2}$, а все остальные еще не включенные в кластеры точки находятся дальше. Отметим, что в примере 7.2 мы в конечном итоге пришли к такому же решению, но лишь после объединения другой пары точек. В общем случае это правило может привести совсем к другой кластеризации по сравнению с правилом расстояния между центроидами.
2. Считать, что расстоянием между кластерами является среднее расстояние между всеми парами точек, по одной из каждого кластера.
3. *Радиусом* кластера называется максимум из расстояний между всеми его точками и центроидом. Объединяем два кластера, для которых новый кластер будет иметь наименьший радиус. Небольшая модификация – объ-

единять кластеры, для которых результат характеризуется наименьшим средним расстоянием между точками и центроидом. Еще одна модификация – использовать сумму квадратов расстояний между точками и центроидом. В некоторых алгоритмах именно эти величины называются «радиусом».

4. *Диаметром* кластера называется максимум из расстояний между всеми парами его точек. Отметим, что между радиусом и диаметром кластера нет прямой связи, как в случае окружности, но, как правило, они пропорциональны. Возможны такие же модификации, как для радиуса.

**Пример 7.4.** Рассмотрим кластер, состоящий из пяти точек, изображенных на рис. 7.2 справа. Его центроид находится в точке (10.8, 4.2). Две точки – (9,3) и (12,6) – находятся на одинаковом максимальном расстоянии от центроида: $\sqrt{4.68} = 2.16$. Следовательно, радиус кластера равен 2.16. Для определения диаметра находим две точки кластера на максимальном удалении друг от друга. Это снова (9,3) и (12,6). Расстояние между ними равно $\sqrt{18} = 4.24$, это и есть диаметр. Отметим, что диаметр превосходит радиус не ровно в два раза, но близко к тому. Причина в том, что центроид находится на прямой, соединяющей точки (9,3) и (12,6).

Можно также по-другому задавать критерий остановки объединения. Мы уже упоминали критерий «остановиться, когда останется $k$ кластеров» для заранее выбранного $k$. Вот другие варианты.

1. Остановиться, если диаметр кластера, получающегося после наилучшего объединения, превышает порог. Можно вместо диаметра использовать радиус или любой из вышеупомянутых вариантов его определения.
2. Остановиться, если *плотность* кластера, получающегося после наилучшего объединения, оказалась ниже порога. Плотность можно определять поразному. Грубо говоря, это количество точек кластера в единице его объема. Эту величину можно оценить, как частное от деления числа точек на некоторую степень диаметра или радиуса кластера. В качестве степени можно взять размерность пространства. Иногда выбирают 1 или 2 независимо от размерности.
3. Остановиться, когда есть причины полагать, что следующая пара кластеров, подлежащая объединению, приведет к появлению плохого кластера. Например, мы можем отслеживать средний диаметр всех имеющихся кластеров. Пока объединяемые точки действительно принадлежат кластеру, это среднее будет расти постепенно. Но если мы объединим два кластера, которые объединять не следовало, то средний диаметр вырастет резко.

**Пример 7.5.** Вернемся к рис. 7.2. Здесь есть три естественных кластера. В примере 7.4 мы вычислили диаметр наибольшего – состоящего из пяти точек справа; он равен 4.24. Диаметр кластера из трех точек в левом нижнем углу равен 3, расстоянию между точками (2,2) и (5,2). Диаметр кластера

из четырех точек в левом верхнем углу равен $\sqrt{13} = 3.61$. Средний диаметр, 3.62, был достигнут после девяти объединений, так что рост, очевидно, медленный: примерно на 0.4 при каждом объединении.

Если бы мы захотели объединить два из этих естественных кластеров, то наилучший результат получился бы при объединении кластеров слева. Диаметр нового кластера оказался бы равен $\sqrt{89} = 9.43$; это расстояние между точками (2,2) и (7,10). Теперь средний диаметр стал равен $(9.43 + 4.24)/2 = 6.84$. Это скачок на величину, чуть меньшую достигнутой на всех девяти предыдущих шагах. Отсюда следует, что последнее объединение было нежелательным, его следует отменить и остановиться.

## 7.2.4. Иерархическая кластеризация в неевклидовых пространствах

Если пространство неевклидово, то нам нужна какая-то метрика, вычисляемая по точкам, например: расстояние Жаккара, косинусное или редакционное расстояние. То есть мы не можем положить в основу расстояния «местоположение» точек. В алгоритме из раздела 7.2.1 требовалось вычислять расстояния между точками, и способ вычисления расстояний у нас предположительно есть. Проблема возникает из-за представления кластера, потому что мы не можем заменить множество точек их центроидом.

**Пример 7.6.** Эта проблема характерна для всех рассмотренных выше неевклидовых пространств, но для конкретики предположим, что используется редакционное расстояние, и мы решили объединить строки `abcd` и `aecdb`. Редакционное расстояние между ними равно 3, и объединить их в кластер ничто не мешает. Однако не существует строки, представляющей их среднее, или вообще такой, которая естественно лежала бы между ними. Можно было бы взять одну из промежуточных строк, возникавших в процессе преобразования одной строки в другую с помощью операций вставки и удаления, например `aebcd`, но таких вариантов много. Кроме того, когда кластер образуется из более чем двух строк, понятие «промежуточная» вообще теряет смысл.

Раз мы не можем комбинировать точки, входящие в кластер, когда пространство неевклидово, остается единственный вариант: выбрать для представления всего кластера одну из его точек. В идеале эта точка должна быть близка ко всем точкам кластера, т. е. в каком-то смысле быть «центральной». Репрезентативная точка кластера называется *кластроидом*. Выбирать кластроид можно разными способами, все они призваны так или иначе минимизировать расстояния между кластроидом и остальными точками кластера. Типичные решения подразумевают минимизацию следующих величин:

1. Сумма расстояний до других точек кластера.
2. Максимум из расстояний до всех остальных точек кластера.
3. Сумма квадратов расстояний до других точек кластера.

**Пример 7.7.** Допустим, что используется редакционное расстояние, и кластер состоит из четырех точек: `abcd`, `aecdb`, `abecb`, `ecdab`. Расстояния между ними показаны в следующей таблице

| | **ecdab** | **abecb** | **aecdb** |
|---|---|---|---|
| **abcd** | 5 | 3 | 3 |
| **aecdb** | 2 | 2 | |
| **abecb** | 4 | | |

Если применить три критерия центроида к каждой из четырех точек кластера, то получим:

| **Точка** | **Сумма** | **Максимум** | **Сумма квадратов** |
|---|---|---|---|
| `abcd` | 11 | 5 | 43 |
| `aecdb` | 7 | 3 | 17 |
| `abecb` | 9 | 4 | 29 |
| `ecdab` | 11 | 5 | 45 |

Отсюда видно, что какой бы критерий ни взять, кластроидом окажется `aecdb`. Но в общем случае кластроид зависит от критерия.

Способы измерения расстояния между кластерами, упомянутые в разделе 7.2.3, можно применить и в неевклидовом случае, если воспользоваться кластроидом вместо центроида. Например, мы можем объединять два кластера с самыми близкими кластроидами. Можно было бы также использовать среднее или минимальное расстояние между парами точек, взятых из разных кластеров.

Другой возможный критерий связан с измерением плотности кластера, основанной на радиусе или диаметре. Оба понятия имеют смысл и в неевклидовом пространстве. Диаметр – по-прежнему, максимальное расстояние между точками кластера, а радиус можно определить в терминах кластроида, а не центроида. Более того, вполне можно было изначально использовать для вычисления радиуса ту же процедуру, что применялась для выбора кластроида. Например, если в качестве кластроида взять точку с наименьшей суммой квадратов расстояний до других точек, то радиус можно определить как эту сумму квадратов (или корень из нее).

Наконец, в разделе 7.2.3 обсуждались также критерии остановки кластеризации. Ни в одном из них понятие центроида явно не используется нигде, кроме определения радиуса, но мы уже заметили, что это понятие осмысленно и в неевклидовых пространствах. Поэтому при переходе от евклидова к неевклидову случаю существенных изменений в выборе критерия остановки нет.

## 7.2.5. Упражнения к разделу 7.2

**Упражнение 7.2.1.** Выполните иерархическую кластеризацию одномерного множества точек 1, 4, 9, 16, 25, 36, 49, 64, 81 в предположении, что кластер пред-

ставлен своим центроидом (средней точкой) и на каждом шаге объединяются кластеры с наименьшим расстоянием между центроидами.

**Упражнение 7.2.2.** Как изменится кластеризация в примере 7.2, если в качестве расстояния между кластерами использовать:

(*а*) минимум расстояний между парами точек по одной из каждого кластера;
(*б*) среднее всех расстояний между парами точек по одной из каждого кластера.

**Упражнение 7.2.3.** Выполните кластеризацию в примере 7.2, если для объединения выбираются два кластера таких, что результирующий кластер имеет:

(*а*) наименьший радиус;
(*б*) наименьший диаметр.

**Упражнение 7.2.4.** Вычислите плотность каждого из трех кластеров, изображенных на рис. 7.2, если «плотность» определяется как число точек, поделенное на:

(*а*) квадрат радиуса;
(*б*) диаметр (не в квадрате).

Каковы будут значения плотности, определенной в (*а*) и (*б*), кластеров, получающихся при объединении любых двух из этих трех кластеров? Свидетельствует ли различие плотности о том, что кластеры нужно или, наоборот, не нужно объединять?

**Упражнение 7.2.5.** Кластроиды можно выбирать, даже если пространство неевклидово. Вычислите кластроиды естественных кластеров на рис. 7.2 в предположении, что кластроид определен как точка, для которой достигает минимума сумма расстояний до всех остальных точек.

**! Упражнение 7.2.6.** Рассмотрим пространство строк с редакционным расстоянием в качестве метрики. Приведите пример такого множества строк, что если в качестве кластроида выбрать точку, минимизирующую сумму расстояний до всех остальных точек, то кластроидом будет одна строка, а если кластроидом считать точку, минимизирующую максимум расстояний до прочих точек, то другая.

# 7.3. Алгоритм k средних

В этом разделе мы начнем изучать алгоритмы отнесения точек. Самым известным семейством алгоритмов такого рода являются алгоритмы *k средних* (*k*-means). В них предполагается, что пространство евклидово и что число кластеров *k* известно заранее. Однако *k* можно вывести методом проб и ошибок. Познакомившись с семейством алгоритмов *k* средних, мы сосредоточимся на конкретном алгоритме BFR, названном по первым буквам фамилий авторов. Он позволяет производить вычисления, когда данные не помещаются в оперативной памяти.

## 7.3.1. Основы алгоритма k средних

Структура алгоритма *k* средних приведена на рис. 7.7. Существуют разные способы выбора начальных *k* точек, представляющих кластеры, мы обсудим их в

разделе 7.3.2. В основе алгоритма лежит цикл for, в котором каждая точка, кроме первоначально выбранных $k$ точек, относится к ближайшему кластеру, где под «ближайшим» понимается кластер с ближайшим центроидом. Отметим, что после отнесения к кластеру новых точек его центроид может сместиться. Но поскольку мы, скорее всего, помещаем в кластер близкие к нему точки, то смещение центроида не должно быть большим.

```
Выбрать начальные k точек, которые, скорее всего, принадлежат
    разным кластерам;
Сделать эти точки центроидами кластеров;
FOR каждая из оставшихся точек p DO
    Найти ближайший к p центроид;
    Включить p в кластер с этим центроидом;
    Пересчитать центроид кластера, в который включена p;
END;
```

**Рис. 7.7.** Структура алгоритма *k* средних

В конце можно выполнить необязательный шаг: зафиксировать центроиды кластеров и заново отнести каждую точку, включая и $k$ начальных к одному из $k$ кластеров. Обычно точка $p$ будет отнесена к тому же кластеру, в который попала на первом проходе. Но бывают случаи, когда центроид кластера, в который $p$ была помещена первоначально, сместился после этого далеко от $p$, так что на втором проходе $p$ попадает в другой кластер. На самом деле, даже некоторые из начальных $k$ точек могут оказаться в других кластерах. Но поскольку такие случаи необычны, мы не будем заострять на них внимание.

## 7.3.2. Инициализация кластеров в алгоритме k средних

Мы хотим выбирать точки, которые с большой вероятностью принадлежат разным кластерам. Существует два подхода.

1. Выбрать точки, отстоящие друг от друга как можно дальше.
2. Провести кластеризацию некоторой выборки данных, возможно иерархическую, чтобы получилось $k$ кластеров. Выбрать по одной точке из каждого кластера, скажем, взять ближайшую к центроиду.

Второй подход вполне бесхитростен. А у первого есть несколько вариаций. Хорошим считается такой способ:

```
Выбрать первую точку случайно;
WHILE точек меньше k DO
    Добавить точку, для которой минимум расстояний до каждой из
    выбранных точек достигает максимума;
END;
```

**Пример 7.8.** Рассмотрим двенадцать точек на рис. 7.2, воспроизведенном ниже на рис. 7.8. В худшем случае мы выберем начальную точку близко к

центру, скажем (6,8). Следующей будет выбрана самая далекая от нее точка, (12,3).

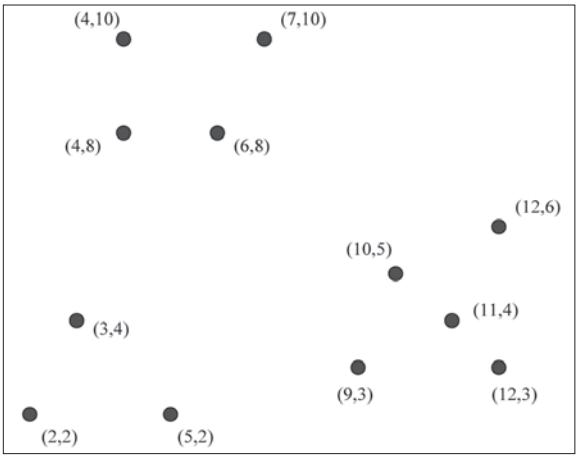



**Рис. 7.8.** Повторение рис. 7.2

Из десяти оставшихся точек максимум из минимума расстояний до (6,8) и (12,3) достигается для точки (2,2). Она отстоит на расстояние $\sqrt{52} = 7.21$ от точки (6,8) и на расстояние $\sqrt{101} = 10.05$ до точки (12,3), поэтому ее «оценка» равна 7.21. Легко проверить, что все прочие точки отстоят на расстояние, меньшее 7.21, хотя бы от одной из точек (6,8) или (12,3). Таким образом, в качестве трех начальных точек мы выбираем (6,8), (12,3) и (2,2). Отметим, что все они принадлежат разным кластерам.

Начав с другой точки, скажем (10,5), мы получили бы другое начальное множество из трех точек. В этом случае начальными были бы точки (10,5), (2,2) и (4,10). Но все равно они принадлежат разным кластерам.

## 7.3.3. Выбор правильного значения k

Мы не всегда заранее знаем правильное значение $k$ в алгоритме $k$ средних. Но, измерив качество кластеризации для разных значений $k$, обычно можно угадать подходящее значение. Напомним, что в разделе 7.2.3 и особенно в примере 7.5 мы заметили, что мера неадекватности кластера, например средний радиус или диаметр, растет медленно, пока количество построенных кластеров остается не меньше истинного их числа. Но как только мы пытаемся уменьшить число кластеров ниже этой величины, мера неадекватности скачкообразно возрастает. Это наблюдение иллюстрируется на рис. 7.9.

Если мы не знаем правильное значение $k$, то можем найти приемлемое приближение, выполнив несколько операций кластеризации, количество которых логарифмически зависит от истинного значения. Будем прогонять алгоритм $k$ средних для $k = 1, 2, 4, 8, \ldots$. В конечном итоге мы найдем два значения $v$ и $2v$ такие, что при переходе от одного к другому средний диаметр или иная выбранная мера ком-

пактности кластера почти не уменьшается. Можно сделать вывод, что истинное значение $k$ лежит между $v/2$ и $v$. Применив двоичный поиск (см. ниже) в этом диапазоне, мы сможем найти наилучшее значение $k$, выполнив еще $\log_2 v$ операций кластеризации, так что всего понадобится $2 \log_2 v$ операций. Поскольку истинное значение $k$ не меньше $v/2$, количество выполненных операций кластеризации логарифмически зависит от $k$.

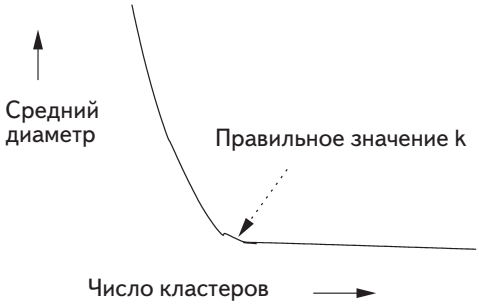



**Рис. 7.9.** Средний диаметр или иная мера размытости кластера резко возрастает, как только число построенных кластеров становится меньше истинного числа кластеров в данных

Так как фраза «изменяется не слишком сильно» расплывчата, мы не можем точно сказать, какое изменение следует считать значительным. Но можно произвести описанный ниже двоичный поиск в предположении, что понятие «изменяется не слишком сильно» может быть формализовано в виде некоторой формулы. Мы знаем, что изменение при переходе от $v/2$ к $v$ не слишком велико, иначе мы не стали бы производить кластеризацию при $2v$ кластерах. Предположим, что в некоторой точке мы сузили границы диапазона $k$ до значений $x$ и $y$. Положим $z = (x + y)/2$. Выполним кластеризацию, взяв $z$ в качестве целевого числа кластеров. Если изменение при переходе от $z$ к $y$ не слишком велико, то истинное значение $k$ лежит между $x$ и $z$. Тогда мы рекурсивно сужаем этот интервал для поиска правильного значения $k$. С другой стороны, если при переходе от $z$ к $y$ изменение чрезмерно велико, то производим двоичный поиск в диапазоне между $z$ и $y$.

## 7.3.4. Алгоритм Брэдли-Файяда-Рейна

Этот алгоритм, который мы будем называть BFR по первым буквам фамилий авторов, является вариантом алгоритма $k$ средних для кластеризации данных в евклидовом пространстве высокой размерности. В нем делается очень сильное предположение о форме кластеров: они должны иметь нормальное распределение относительно центроида. Среднее и стандартное отклонение могут быть разными по разным измерениям, но измерения должны быть независимы. Например, в двумерном пространстве кластер может иметь форму эллипса, но этот эллипс не должен располагаться под углом к осям. См. рис. 7.10.

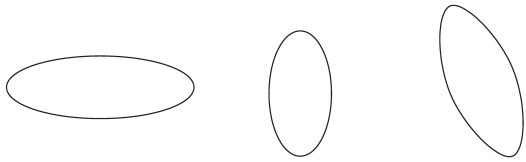



**Рис. 7.10.** Кластеры в данных, к которым применим алгоритм BFR, могут иметь разные стандартные отклонения по осям, но сами оси кластера должны быть параллельны осям координат

В начале алгоритма BFR мы выбираем $k$ точек, используя один из методов, рассмотренных в разделе 7.3.2. Затем читаем из файла данные о точках порциями. В распределенной файловой системе можно взять уже готовые порции, а обычный файл разбить на куски подходящего размера. Количество точек в каждой порции должно быть не слишком велико, чтобы все они поместились в оперативной памяти. Кроме них, в оперативной памяти хранятся обобщенные представления $k$ кластеров и некоторые другие данные, т. е. не вся память отводится под точки. Помимо порции входных данных, в оперативной памяти хранятся объекты трех типов.

1. *Отброшенное множество*. Это простые обобщенные представления самих кластеров. Как именно обобщается кластер, мы рассмотрим чуть ниже. Отметим, что обобщенные представления не «отбрасываются», они-то как раз нужны. А отбрасываются точки, описываемые обобщенным представлением; от них в памяти не остается никаких следов, кроме этого представления.
2. *Сжатое множество*. Это тоже обобщенные представления, подобные представлениям кластеров, но для множеств точек, которые близки друг к другу, но не близки ни к какому кластеру. Точки, представленные в сжатом множестве, также отбрасываются в том смысле, что явно в оперативной памяти не присутствуют. Будем называть представленные таким образом множества точек *миникластерами*.
3. *Сохраненное множество*. Могут существовать точки, которые нельзя отнести ни к какому кластеру, но при этом они недостаточно близки к другим точкам, чтобы представлять их сжатым множеством. Они хранятся в оперативной памяти точно в том виде, в каком прочитаны из файла.

На рис. 7.11 показано, как представлены уже обработанные точки. Отброшенное и сжатое множество представлены $2d + 1$ значениями, где $d$ – размерность пространства. Вот эти значения:

(*а*) Число представленных точек $N$.
(*б*) Сумма координат всех точек по каждому измерению. Обозначим этот $d$-мерный вектор SUM, а его $i$-й элемент – $\text{SUM}_i$.
(*в*) Сумма квадратов координат всех точек по каждому измерению. Обозначим этот $d$-мерный вектор SUMSQ, а его $i$-й элемент – $\text{SUMSQ}_i$.

Наша истинная цель – представить множество точек их количеством, центроидом и стандартным отклонением по каждому измерению. Но эти $2d + 1$ значений как раз и содержат нужную статистику. $N$ – это количество точек. Координата центроида по $i$-му измерению равна $\text{SUM}_i/N$, т. е. сумме координат по этому измерению, поделенной на число точек. Дисперсия по $i$-му измерению равна $\text{SUMSQ}_i/N - (\text{SUM}_i/N)^2$. Стандартное отклонение по каждому измерению равно квадратному корню из дисперсии.

**Пример 7.9.** Пусть кластер состоит из точек (5,1), (6,–2) и (7,0). Тогда $N = 3$, SUM = [18,–1], SUMSQ = [110,5]. Центроидом является точка SUM/$N$, или [6,–1/3]. Дисперсия по первому измерению равна $110/3 - (18/3)^2 = 0.667$, поэтому стандартное отклонение равно $\sqrt{0.667} = 0.816$. По второму измерению дисперсия равна $5/3 - (-1/3)^2 = 1.56$, а стандартное отклонение 1.25.

**Преимущества представления в виде N, SUM, SUMSQ**

Представлять множества точек, как принято в алгоритме BFR, удобнее, чем хранить $N$, центроид и стандартное отклонение по каждому измерению. Посмотрим, что нужно сделать при добавлении в кластер новой точки. Ясно, что $N$ увеличивается на 1. Кроме того, мы можем прибавить вектор, представляющий новую точку, к SUM, получив тем самым новое значение SUM, а сумму квадратов координат вектора прибавить к SUMSQ и получить новое значение SUMSQ. Если бы вместо SUM мы использовали центроид, то не смогли бы пересчитать его с учетом новой точки, не производя вычислений, в которых участвует $N$. И пересчет стандартных отклонений также оказался бы гораздо сложнее. Аналогично, чтобы объединить два множества, нам достаточно было бы сложить соответствующие значения $N$, SUM и SUMSQ, тогда как при использовании центроида и стандартного отклонения вычисления были бы куда сложнее.

## 7.3.5. Обработка данных в алгоритме BFR

Теперь опишем, как производится обработка порции точек.

1. Первым делом все точки, достаточно близкие к центроиду кластера, включаются в этот кластер. Как написано во врезке, очень просто добавить информацию о точке к объектам $N$, SUM и SUMSQ, представляющим кластер. Включенная в кластер точка отбрасывается. На вопрос о том, что такое «достаточно близки», мы вскоре ответим.
2. Точки, недостаточно близкие к центроиду любого кластера, мы объединяем в кластеры вместе с точками в сохраненном множестве. Для этого можно использовать любой алгоритм кластеризации в оперативной памяти, например иерархические методы (см. раздел 7.2). Необходим какой-то крите-

рий, позволяющий решить, когда имеет смысл объединять две точки в кластер или два кластера в один. В разделе 7.2.3 описаны некоторые варианты. Кластеры, содержащие более одной точки, обобщаются и добавляются в сжатое множество. Одноэлементные кластеры включаются в сохраненное множество точек.

3. Теперь у нас есть миникластеры, образовавшиеся при объединении новых точек и старого сохраненного множества, а также миникластеры из старого сжатого множества. И хотя ни один из них нельзя объединить ни с одним из $k$ кластеров, возможно, их удастся объединить между собой. Критерий объединения снова можно взять из числа рассмотренных в разделе 7.2.3. Отметим, что благодаря способу представления сжатых множеств ($N$, SUM и SUMSQ) легко вычислить такие статистики, как дисперсия объединения двух миникластеров.
4. Точки, отнесенные к какому-нибудь кластеру или миникластеру, т. е. не оставшиеся в сохраненном множестве, записываются вместе с информацией о кластере, в который включены, во внешнюю память.

Наконец, если это последняя порция входных данных, то нужно что-то сделать со сжатым и сохраненным множествами. Можно считать, что это выбросы, и вообще не подвергать их кластеризации. Или отнести каждую точку сохраненного множества к кластеру с ближайшим центроидом. Каждый миникластер можно объединить с кластером, центроид которого ближе всего расположен к центроиду миникластера.

Важный момент – как решить, находится ли новая точка $p$ достаточно близко к одному из $k$ кластеров, чтобы имело смысл включать ее в кластер. Было предложено два подхода.

($a$) Включать $p$ в кластер, если не только его центроид расположен ближе всего к $p$, но и крайне маловероятно, что после обработки всех точек найдется другой центроид, который будет ближе к $p$. Это решение требует сложных статистических вычислений. Необходимо предположить, что точки упорядочены случайным образом и что мы знаем, сколько точек будет обработано в будущем. Но достоинство его в том, что если мы обнаружим, что один центроид значительно ближе к $p$, чем все остальные, то сможем включить $p$ в соответствующий кластер и успокоиться на этом, даже если $p$ очень далека от всех центроидов.

($б$) Можно измерить вероятность того, что точка $p$, принадлежащая некоторому кластеру, может быть обнаружена на том расстоянии от его центроида, на каком действительно находится. Это вычисление опирается на предположения о нормальном распределении точек и о параллельности осей распределения осям координат, что позволяет воспользоваться *расстоянием Махалонобиса*, описываемым ниже.

Расстояние Махалонобиса – это, по существу, расстояние между точкой и центроидом кластера, нормированное по каждому измерению на стандартное откло-

нение кластера. Поскольку в алгоритме BFR предполагается, что оси кластера параллельны осям координат, то расстояние Махалонобиса вычисляется особенно просто. Пусть $p = [p_1, p_2, ..., p_d]$ – точка, а $c = [c_1, c_2, ..., c_d]$ – центроид кластера. Обозначим $\sigma_i$ стандартное отклонение точек кластера по $i$-му измерению. Тогда расстояние Махалонобиса между $p$ и $c$ вычисляется по формуле:

$$\sqrt{\sum_{i=1}^{d}\left(\frac{p_i - c_i}{\sigma_i}\right)^2}.$$

То есть мы нормируем разность между $p$ и $c$ по $i$-му измерению путем деления на стандартное отклонение кластера по этому измерению. А затем объединяем нормированные расстояния обычным для евклидова пространства способом.

Решая, следует ли отнести точку $p$ к кластеру, мы вычисляем расстояние Махалонобиса между $p$ и центроидами всех кластеров. Выбирается кластер, для которого расстояние Махалонобиса минимально, и точка включается в этот кластер, если расстояние превосходит заданный порог. Пусть, например, порог принят равным 4. Если данные имеют нормальное распределение, то вероятность, что значение будет отстоять от среднего более чем на четыре стандартных отклонения, меньше одной миллионной. Следовательно, если точки в кластере действительно распределены нормально, то вероятность не включить в кластер точку, которая на самом деле ему принадлежит, меньше $10^{-6}$. И такая точка, скорее всего, в любом случае будет включена в данный кластер, если только не окажется ближе к какому-то другому центроиду, поскольку центроиды смещаются после добавления новых точек.

## 7.3.6. Упражнения к разделу 7.3

**Упражнение 7.3.1.** Рассмотрим точки, изображенные на рис. 7.8. Если при выборе начальных трех точек руководствоваться методом из раздела 7.3.2 и первой выбрать точку (3,4), то какие еще точки будут выбраны?

**!! Упражнение 7.3.2.** Докажите, что при выборе начальных трех точек в соответствии с методом из раздела 7.3.2 мы получим по одной точке в каждом из трех кластеров независимо от того, с какой точки начинали.

*Подсказка:* можно доказать утверждение полным перебором, начиная по очереди с каждой из 12 точек. Но есть и более общий подход: рассмотреть диаметры всех трех кластеров и минимальное межкластерное расстояние (т. е. минимум расстояний между всеми парами точек по одной из каждого кластера). Сможете ли вы доказать общую теорему, основываясь на этих двух параметрах множества точек?

**! Упражнение 7.3.3.** Приведите пример набора данных и выбора $k$ начальных центроидов таких, что после того как на последнем шаге произведено окончательное отнесение точек к кластеру с ближайшим центроидом, хотя бы одна из начальных $k$ точек оказывается не в своем исходном кластере.

**Упражнение 7.3.4.** Для трех кластеров, изображенных на рис. 7.8:

(*а*) Вычислите представление кластера, предписываемое алгоритмом BFR, т. е. *N*, SUM и SUMSQ.

(*б*) Вычислите дисперсию и стандартное отклонение каждого кластера по обоим измерениям.

**Упражнение 7.3.5.** Предположим, что стандартные отклонения трехмерного кластера по осям *x*, *y*, *z* равны соответственно 2, 3, 5. Вычислите расстояние Махалонобиса от начала координат (0, 0, 0) до точки (1, –3, 4).

# 7.4. Алгоритм CURE

Обратимся теперь к другому алгоритму кластеризации больших наборов данных, основанному на идее отнесения точек. Он называется CURE (Clustering Using REpresentatives) и предполагает, что пространство евклидово. Однако не делается никаких предположений о форме кластеров; они не обязаны подчиняться нормальному распределению и могут иметь странные формы, например, S-образную или даже кольцевую. Кластер представляется не центроидом, а набором репрезентативных точек.

**Пример 7.10.** На рис. 7.12 изображены два кластера. Внутренний представляет собой обычный круг, а внешний – кольцо вокруг этого круга. Это не совсем патологический случай. Обитатель другой галактики, наблюдающий за нашей солнечной системой, заметил был, что объекты скапливаются во внутреннем круге (планеты) и во внешнем кольце (пояс Койпера), а между ними почти ничего нет.

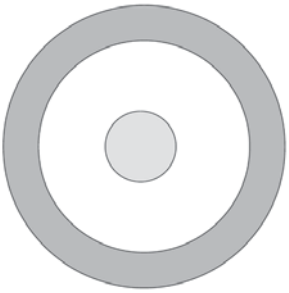


**Рис. 7.12.** Два кластера, один внутри другого

## 7.4.1. Этап инициализации в CURE

Работа алгоритма CURE начинается следующим образом.

1. Взять небольшую выборку данных и выполнять ее кластеризацию в оперативной памяти. В принципе, подойдет любой метод кластеризации, но, поскольку CURE предназначен для обработки кластеров необычной формы, часто рекомендуется использовать иерархическую кластеризацию и объединять кластеры, у которых есть близкая пара точек. Подробнее этот вопрос обсуждается в примере 7.11 ниже.
2. Выбрать небольшой набор точек в каждом кластере, которые станут *репрезентативными точками*. При этом нужно выбирать точки, отстоящие как можно дальше друг от друга, применяя любой метод из описанных в разделе 7.3.2.
3. Сдвинуть каждую репрезентативную точку на фиксированную долю расстояния между текущим положением и центроидом кластера, из которого

она взята. Например, можно взять долю 20 %. Отметим, что для этого шага пространство должно быть евклидовым, потому что иначе понятие отрезка между двумя точками может не иметь смысла.

**Пример 7.11.** Можно было бы применить иерархическую кластеризацию к выборке данных на рис. 7.12. Если за расстояние между кластерами принять минимальное расстояние между парами точек по одной из каждого кластера, то мы правильно найдем оба кластера. То есть части кольца объединятся между собой, части внутреннего круга – между собой, но части круга будут далеко от частей кольца. Отметим, что если бы за расстояние между кластерами мы приняли расстояние между их центроидами, то могли бы получить результат, противоречащий интуиции. Дело в том, что центроиды обоих кластеров совпадают с центром круга.

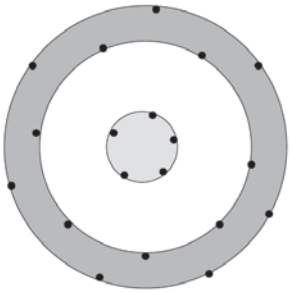


**Рис. 7.13.** Выбрать репрезентативные точки в каждом кластере, отстоящие как можно дальше друг от друга

На втором шаге мы выбираем репрезентативные точки. Если выборка, на основе которой были построены кластеры, достаточно велика, то можно рассчитывать, что выборочные точки, максимально удаленные друг от друга, расположатся на границе кластера. На рис. 7.13 показано, как могли бы выглядеть начальные выборочные точки.

Наконец, сдвигаем репрезентативные точки на фиксированную долю расстояния между их истинным положением и центроидом кластера в направлении центроида. Отметим, что на рис. 7.13 центроиды обоих кластеров совпадают с центром внутреннего круга. Поэтому репрезентативные точки, принадлежащие кругу, сдвигаются внутрь кластера, как и было задумано. Точки, расположенные на внешнем крае кольца, тоже сдвигаются внутрь своего кластера, но точки на внутреннем крае, сдвигаются в сторону от кластера. Окончательные позиции репрезентативных точек показаны на рис. 7.14.

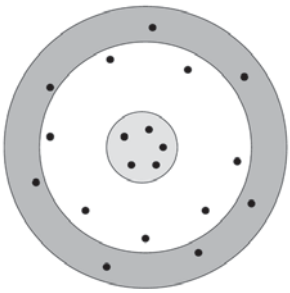


**Рис. 7.14.** Смещение репрезентативных точек в направлении центроида кластера на 20 % расстояния

## 7.4.2. Завершение работы алгоритма CURE

Следующий этап CURE – объединение двух кластеров, если существует достаточно близкая пара принадлежащих им репрезентативных точек. Пользователь может сам определить, какое расстояние считать «близким». Шаг объединения может повторяться, пока не останется достаточно близких кластеров.

**Пример 7.12.** Рис. 7.14 может служить полезной иллюстрацией. Можно поспорить о том, не следует ли объединить круг и кольцо, коль скоро их центроиды совпадают. Например, если бы промежуток между кругом и

кольцом был гораздо меньше, то были бы все основания утверждать, что объединение того и другого в один кластер отражает истинное положение вещей. Скажем, между кольцами Сатурна имеются узкие промежутки, но разумнее представлять их в виде одного, а не нескольких объектов. В случае, изображенном на рис. 7.14, вопрос об объединении обоих кластеров решается с учетом двух соображений:

1. Доля расстояния до центроида, на которую мы сдвигаем репрезентативные точки.
2. Решение о том, насколько далеко должны отстоять репрезентативные точки двух кластеров, чтобы предотвратить объединение.

Последний шаг алгоритма CURE – отнесение точек. Все точки $p$ читаются из внешней памяти и сравниваются с репрезентативными. Точка $p$ относится к тому кластеру, которому принадлежит ближайшая к ней репрезентативная точка.

**Пример 7.13.** В нашем примере точки внутри кольца, без сомнения, будут ближе к какой-то из репрезентативных точек кольца, чем к любой репрезентативной точке круга. Аналогично точки внутри круга, конечно, будут ближе к какой-то из репрезентативных точек круга. Выброс – точка, лежащая вне круга и вне кольца, – будет отнесен к кольцу, если находится снаружи кольца. Если же выброс находится между кольцом и кругом, то может быть отнесен туда или сюда, хотя отнесение к кольцу вероятнее, т. к. его репрезентативные точки были сдвинуты в направлении круга.

### 7.4.3. Упражнения к разделу 7.4

**Упражнение 7.4.1.** Рассмотрим два кластера: круг и окружающее его кольцо, как в примере из этого раздела. Предположим, что:

i. Радиус круга равен $c$.
ii. Внутренний и внешний радиус кольца равны соответственно $i$ и $o$.
iii. Все репрезентативные точки обоих кластеров расположены на их границах.
iv. Репрезентативные точки сдвигаются на 20 % в направлении центроида своего кластера.
v. Два кластера объединяются, если после сдвига найдется пара репрезентативных точек по одной из каждого кластера, удаленных друг от друга на расстояние не более $d$.

В терминах $d$, $c$, $i$, $o$ при каких условиях круг и кольцо будут объединены в один кластер?

## 7.5. Кластеризация в неевклидовых пространствах

Следующим мы рассмотрим алгоритм, который может работать с данными, не помещающимися в оперативную память, но не предполагает, что пространство

евклидово. Будем называть его GRGPF по первым буквам фамилий авторов (V. Ganti, R. Ramakrishnan, J. Gehrke, A. Powell, J. French). В этом алгоритме применяются идеи из обоих подходов: иерархического и с отнесением точек. Как и в CURE, кластеры представляются выборкой точек в оперативной памяти. Однако делается попытка организовать кластеры иерархически, в виде дерева, чтобы новую точку можно было отнести к подходящему кластеру, спускаясь вниз по дереву. В листьях дерева хранятся обобщенные представления некоторых кластеров, а во внутренних узлах – частичная информация, описывающая кластеры, достижимые из этого узла. Алгоритм стремится сгруппировать кластеры по разделяющему их расстоянию, так что кластеры в одном листовом узле близки, и кластеры, достижимые из одного внутреннего узла, также относительно близки.

## 7.5.1. Представление кластеров в алгоритме GRGPF

По мере включения точек кластеры могут расти. Большая часть точек кластера хранится на диске и не принимает участия в принятии решений об отнесении точек, хотя их можно с диска прочитать. Представление кластера в оперативной памяти состоит из нескольких *признаков*. Но прежде чем перечислять их, введем обозначение. Если $p$ – произвольная точка кластера, обозначим ROWSUM($p$) сумму квадратов расстояний между $p$ и всеми остальными точками кластера. Отметим, что хотя пространство неевклидово, в нем существует некая метрика $d$, применимая к точкам, иначе о кластеризации вообще нельзя было бы говорить. В представлении кластера участвуют следующие признаки:

1. Количество точек в кластере $N$.
2. Кластроид кластера, определенный как точка кластера, для которой достигается минимум суммы квадратов расстояний до всех остальных точек. То есть кластроид – точка с наименьшим значением ROWSUM.
3. Величина ROWSUM для кластроида.
4. $k$ точек кластера, ближайших к кластроиду, и значения ROWSUM для них, где $k$ – заранее выбранная константа. Эти точки являются частью представления на случай, если добавление точек в кластер приводит к изменению кластроида. Мы исходим из предположения, что новый кластроид будет одной из $k$ точек, близких к старому.
5. $k$ точек кластера, самых далеких от кластроида, и значения ROWSUM для них. Эти точки являются частью представления, чтобы можно было решить, достаточно ли близки два кластера для объединения. Мы исходим из предположения, что если два кластера близки, то даже пара точек, удаленных от соответствующих кластроидов, будет близка.

## 7.5.2. Инициализация дерева кластеров

Кластеры организуются в дерево, узлы которого могут быть очень велики, быть может несколько дисковых блоков или страниц, как в случае B-дерева или

R-дерева, с которыми у дерева кластеров есть сходство. Каждый листовый узел дерева содержит столько представлений кластеров, сколько помещается. Отметим, что у представления кластера имеется размер, который не зависит от числа точек в кластере.

В каждом внутреннем узле дерева хранится выборка кластроидов тех кластеров, которые представлены в каждом поддереве, а также указатели на корни этих поддеревьев. Размер выборки фиксирован, поэтому число дочерних узлов внутреннего узла не зависит от его уровня. Отметим, что при подъеме вверх по дереву вероятность, что кластроид данного кластера входит в выборку, уменьшается.

Для инициализации дерева кластеров мы формируем выборку из набора данных в оперативной памяти и применяем к ней иерархическую кластеризацию. В результате получается дерево $T$, но это не совсем то дерево, которое нужно алгоритму GRGPF. Далее мы выбираем из $T$ некоторые узлы, представляющие кластеры, размер которых близок к некоему желательному размеру $n$. Это и будут начальные кластеры в алгоритме GRGPF, и мы помещаем их представления в листья дерева кластеров. Затем мы группируем кластеры с общим предком в $T$ и таким образом получаем внутренние узлы дерева кластеров, т. е. в некотором смысле кластеры, находящиеся в потомках одного и того же внутреннего узла, настолько близки, насколько это возможно. В некоторых случаях необходима балансировка дерева представлений кластеров. Эта процедура похожа на реорганизацию B-дерева, и мы не станем останавливаться на ее деталях.

### 7.5.3. Добавление точек в алгоритме GRGPF

Теперь нужно читать точки из внешней памяти и включать каждую в ближайший к ней кластер. Мы начинаем с корня и просматриваем выборки кластроидов для каждого дочернего узла. Для следующего шага выбирается тот узел, в котором обнаружен кластроид, ближайший к новой точке $p$. Эта процедура повторяется, пока мы не достигнем листового узла. Отметим, что некоторые выборочные кластроиды уже могли встречаться на более высоком уровне, но каждый уровень предоставляет больше деталей о кластерах, расположенных ниже, поэтому при переходе к следующему уровню мы можем увидеть много новых выборочных кластроидов.

В конечном итоге мы дойдем до листового узла. В нем хранятся признаки всех представленных в этом узле кластеров, и мы выбираем кластер, ближайший к $p$. Представление этого кластера модифицируется с учетом добавления новой точки. Точнее, выполняются следующие действия:

1. $N$ увеличивается на 1.
2. К ROWSUM($q$) для каждой точки $q$, встречающейся в представлении, прибавляется квадрат расстояния между $p$ и $q$. В состав точек $q$ входит кластроид, $k$ ближайших точек и $k$ самых далеких точек.

Мы также оцениваем величину ROWSUM($p$), если $p$ должна стать частью представления (например, она оказалась одной из $k$ точек, ближайших к кластроиду). Заметим, что точно вычислить ROWSUM($p$) мы не можем, поскольку для этого не-

обходимо прочитать с диска все точки кластера. Оценка вычисляется по следующей формуле:

$$\text{ROWSUM}(p) = \text{ROWSUM}(c) + Nd^2(p, c),$$

где $d(p, c)$ – расстояние между $p$ и кластроидом $c$. Отметим, что $N$ и $\text{ROWSUM}(c)$ в этой формуле – значения признаков до их пересчета в связи с добавлением $p$.

Возникает вопрос, почему эта оценка работает. В разделе 7.1.3 мы обсуждали «проклятие размерности» и, в частности, отметили, что в многомерном евклидовом пространстве почти все углы прямые. Конечно, в алгоритме GRGPF не предполагается, что пространство евклидово, но обычно неевклидовы пространства точно так же страдают от проклятия размерности. Если предположить, что угол между $p$, $c$, и еще какой-то точкой кластера $q$ прямой, то по теореме Пифагора

$$d^2(p, q) = d^2(p, c) + d^2(c, q).$$

Если просуммировать по всем $q$, отличным от $c$, а затем прибавить $d^2(p, c)$ к $\text{ROWSUM}(p)$, чтобы учесть тот факт, что кластроид – одна из точек кластера, то окажется, что

$$\text{ROWSUM}(p) = \text{ROWSUM}(c) + Nd^2(p, c).$$

Теперь нужно посмотреть, является ли новая точка $p$ одной из $k$ самых близких к кластроиду или самых далеких от него точек. Если так, то вычисленная для нее величина ROWSUM становится признаком кластера, заменяя один из прежних признаков – тот, что связан с точкой, переставшей быть одной из самых близких или самых далеких. Кроме того, нужно посмотреть, не оказалось ли, что ROWSUM для одной из $k$ ближайших точек $q$ теперь меньше, чем $\text{ROWSUM}(c)$. Такое может случиться, если $p$ ближе к одной из этих точек, чем текущий кластроид. В таком случае мы меняем $c$ и $q$ ролями. В конечном итоге может получиться так, что истинный кластроид больше не является одной из первоначально выбранных $k$ ближайших точек. Знать этого мы не можем, потому что в оперативной памяти нет других точек кластера. Но все они хранятся во внешней памяти и периодически их можно считывать в оперативную для пересчета признаков кластера.

## 7.5.4. Разделение и объединение кластеров

В алгоритме GRGPF предполагается, что существует ограничение на радиус кластера. Радиус определяется как $\sqrt{\text{ROWSUM}(c)/N}$, где $c$ – кластроид, а $N$ – число точек в кластере. Иными словами, радиус – это квадратный корень из среднеквадратичного расстояния от кластроида до всех точек кластера. Если радиус кластера становится слишком большим, то кластер разделяется на два. Все точки этого кластера считываются в оперативную память и распределяются между двумя кластерами, так чтобы минимизировать величины ROWSUM. Вычисляются признаки обоих кластеров.

В результате в листовом узле разделенного кластера становится на один кластер больше. Дерево кластеров балансируется, как B-дерево, поэтому обычно в

листе имеется место еще для одного кластера. Если же это не так, то листовый узел следует разделить на два. Для этого мы должны добавить еще один указатель и дополнительные выборочные кластроиды в родительский узел. Скорее всего, в нем имеется свободное место, но если нет, то придется разделить и этот узел. При выполнении этой операции мы стремимся минимизировать квадраты расстояний между выборочными кластроидами, отнесенными к разным узлам. Как и в B-дереве, такое разделение может распространиться вверх до корня, который, возможно, тоже придется разделить.

В худшем случае дерево представления кластеров перестанет помещаться в памяти. И тут можно сделать только одну вещь: уменьшить дерево, ослабив ограничение на величину радиуса кластера, и попробовать объединить некоторые пары кластеров. Обычно достаточно рассмотреть кластеры «поблизости» в том смысле, что их репрезентативные точки находятся в одном и том же листе или в листах с общим родителем. Но в принципе можно объединить любые два кластера $C_1$ и $C_2$ в один кластер $C$.

Для объединения кластеров предположим, что кластроидом $C$ будет одна из точек, максимально удаленных от кластроида $C_1$ или от кластроида $C_2$. Пусть мы хотим вычислить ROWSUM относительно $C$ для точки $p$, являющейся одной из $k$ точек в $C_1$, максимально удаленных от кластроида $C_1$. Воспользуемся тем же рассуждением, которое использовали для доказательства того, что в многомерных пространствах все углы приблизительно прямые, для обоснования следующей формулы:

$$\text{ROWSUMC}(p) = \text{ROWSUM}_{C_1}(p) + N_{C_2}(d^2(p, c_1) + d^2(c_1, c_2)) + \text{ROWSUM}_{C_2}(c_2).$$

Здесь $N$ и ROWSUM индексированы кластером, к которому относится признак, а $c_1$ и $c_2$ – кластроиды $C_1$ и $C_2$ соответственно.

Точнее, мы вычисляем сумму квадратов расстояний от $p$ до всех точек объединенного кластера $C$, начав с $\text{ROWSUM}_{C_1}(p)$, чтобы получить члены, соответствующие точкам в том же кластере, что и $p$. Для $N_{C_2}$ точек $q$ в $C_2$ мы рассматриваем путь от $p$ до кластроида $C_1$, затем до кластроида $C_2$ и, наконец, до $q$. Мы предполагаем, что угол между отрезками $(p, c_1)$ и $(c_1, c_2)$ прямой, равно как и угол между отрезками $(p, c_2)$ и $(c_2, q)$. Затем мы пользуемся теоремой Пифагора, чтобы обосновать вычисление квадрата длины пути к каждой точке $q$ как суммы квадратов трех его отрезков.

Далее мы должны закончить вычисление признаков для объединенного кластера. Нам нужно рассмотреть в нем все точки, для которых известна величина ROWSUM, а именно: кластроиды обоих кластеров, $k$ точек, самых близких к кластроиду каждого кластера, и $k$ точек, самых далеких от кластроида каждого кластера, за исключением точки, выбранной в качестве нового кластроида. Мы можем вычислить расстояния от нового кластроида до каждой их этих $4k + 1$ точек. Выберем $k$ точек с наименьшими расстояниями в качестве «близких» и $k$ точек с наибольшими расстояниями в качестве «далеких». Затем вычислим ROWSUM для этих точек по тем же формулам, что были использованы выше для вычисления ROWSUM кандидатов в кластроиды.

### 7.5.5. Упражнения к разделу 7.5

**Упражнение 7.5.1.** Применяя представление кластера из раздела 7.5.1, представьте 12 точек, изображенных на рис. 7.8, как один кластер. Считайте, что количество близких и далеких точек в представлении $k = 2$.

*Подсказка*: поскольку метрика евклидова, квадрат расстояния между двумя точками равен сумме квадратов разностей координат по осям $x$ и $y$.

**Упражнение 7.5.2.** Вычислите радиус в смысле алгоритма GRGPF (квадратный корень из среднеквадратичного расстояния до кластроида) для кластера, состоящего из пяти точек в правом нижнем углу на рис. 7.8. Обратите внимание, что точка (11,4) – кластроид.

## 7.6. Кластеризация для потоков и параллелизм

В этом разделе мы кратко рассмотрим вопрос о кластеризации потока. При этом мы будем пользоваться моделью со скользящим окном (см. раздел 4.1.3) из $N$ точек. Предполагается, что для любого $m \leq N$ можно запросить центроиды или кластроиды наилучших кластеров, образованных последними $m$ точками. Мы изучим также аналогичный подход к кластеризации большого фиксированного множества точек в вычислительном кластере (игра слов не намеренная). В этом разделе предлагается только очень общий обзор возможностей, зависящих от предположений об эволюции кластеров в потоке.

### 7.6.1. Модель потоковых вычислений

Будем предполагать, что каждый элемент потока – точка в некотором пространстве. Скользящее окно состоит из $N$ последних точек. Наша цель – заранее подвергнуть кластеризации подмножества точек потока, чтобы можно было быстро отвечать на вопросы вида «какие кластеры имеются среди последних $m$ точек?» для любого $m \leq N$. Есть несколько разновидностей этого вопроса, зависящих от предположений о том, из чего состоит кластер. Например, можно использовать подход на основе метода $k$ средних, когда мы на самом деле требуем, чтобы последние $m$ точек были сгруппированы ровно в $k$ кластеров. Или можно допустить переменное число кластеров и использовать один из критериев, приведенных в разделе 7.2.3 или 7.2.4, чтобы решить, когда прекращать объединение кластеров в более крупные. Мы не накладываем никаких ограничений на пространство, из которого берутся точки потока. Это может быть евклидово пространство, и тогда ответом на вопрос будут центроиды выбранных кластеров. Или неевклидово – и тогда мы возвращаем кластроиды выбранных кластеров, причем определение «кластроида» может быть любым (см. раздел 7.2.4).

Задача значительно упрощается, если предположить, что все элементы потока имеют неизменные статистические характеристики. Тогда для оценки кластеров достаточно выборки из потока, и по прошествии некоторого времени поток мож-

но вообще игнорировать. Но обычно модель потока предполагает, что статистика распределения элементов потока со временем изменяется. Например, центроиды кластеров могут медленно перемещаться. Возможно также, что кластеры расширяются, сжимаются, разделяются и сливаются.

## 7.6.2. Алгоритм кластеризации потока

В этом разделе мы представим сильно упрощенную версию алгоритма BDMO (по первым буквам фамилий авторов: B. Babcock, M. Datar, R. Motwani, L. O'Callaghan). В настоящем алгоритме используются гораздо более сложные структуры, призванные дать гарантии производительности в худшем случае.

Алгоритм BDMO основан на методике подсчета единиц в потоке, которая была описана в разделе 4.6. Перечислим основные сходства и различия.

- Как и в ранее описанном алгоритме, точки потока разбиваются на интервалы, размеры которых являются степенями двойки, а затем обобщаются. В данном случае размером интервала является число представленных им точек, а не число элементов потока, равных 1.
- Как и раньше, на интервалы налагается ограничение: может существовать один или два интервала каждого размера, не превышающего заданный порог. Однако не предполагается, что последовательность допустимых размеров интервалов начинается с 1. Зато требуется, чтобы они образовывали последовательность, в которой каждый размер вдвое больше предыдущего, например: 3, 6, 12, 24, ... .
- Размеры интервалов, как и раньше, должны не убывать при движении назад во времени. Так же, как в разделе 4.6, мы заключаем, что количество интервалов растет как $O(\log N)$.
- В каждом интервале хранятся:
  1. Размер интервала.
  2. Временная метка интервала, т. е. самая недавняя точка, учтенная в этом интервале. Как и в разделе 4.6, временные метки хранятся по модулю $N$.
  3. Набор записей, представляющих кластеры, в которые сгруппированы точки этого интервала. Каждая запись содержит:
     (*а*) количество точек в кластере;
     (*б*) центроид или кластроид кластера;
     (*в*) прочие параметры, необходимые для объединения кластеров и аппроксимации полного комплекта параметров объединенного кластера. Примеры будут приведены в разделе 7.6.4 в ходе обсуждения процесса объединения.

## 7.6.3. Инициализация интервалов

Пусть размер наименьшего интервала равен $p$ и является степенью двойки. Это значит, что через каждые $p$ элементов потока мы создаем новый интервал, содержащий последние $p$ точек. Временная метка этого интервала равна временной мет-

ке последней точки. Мы можем оставить каждую точку в кластере, состоящем из нее самой, или произвести кластеризацию точек, придерживаясь любой заранее выбранной стратегии. Например, выбрав алгоритм $k$ средних (где $k < p$), мы сможем сгруппировать точки в $k$ кластеров.

При любом способе начальной кластеризации предполагается, что существует возможность вычислить центроиды или кластроиды кластеров и подсчитать количество точек в каждом кластере. Эта информация войдет в запись о кластере. Мы также вычисляем дополнительные параметры кластеров, необходимые для их объединения.

## 7.6.4. Объединение кластеров

Следуя стратегии из раздела 4.6, при создании каждого нового интервала мы должны проанализировать последовательность существующих интервалов. Во-первых, если временная метка какого-то интервала отстоит от текущего момента более чем на $N$ единиц времени, то весь этот интервал выходит за пределы окна и должен быть удален из списка. Во-вторых, мы могли уже создать три интервала размера $p$, и в таком случае необходимо объединить два самых старых. В процессе объединения могут быть созданы два интервала размера $2p$, и тогда нам, возможно, придется объединять интервалы следующего размера. Этот процесс продолжается рекурсивно – так же, как в разделе 4.6.

При объединении двух соседних интервалов должны выполняться следующие правила.

1. Размер нового интервала вдвое больше размеров объединяемых.
2. Временная метка объединенного интервала равна временной метке последнего по времени интервала (более позднего).
3. Необходимо выяснить, следует ли объединять кластеры, и, если да, то вычислить параметры объединенных кластеров. Эту часть алгоритма мы разберем на примерах различных критериев объединения и способов оценки необходимых параметров.

**Пример 7.14.** Пожалуй, простейшим является случай метода $k$ средних в евклидовом пространстве. Каждый кластер представляется количеством точек и своим центроидом. В каждом интервале находится ровно $k$ кластеров, поэтому можно взять $p = k$ или выбрать $p$ большим $k$ и группировать $p$ точек в $k$ кластеров в момент создания интервала, как описано в разделе 7.6.3. Необходимо найти наилучшее соответствие между $k$ кластерами из первого и $k$ кластерами из второго интервала. В данном случае под «наилучшим» понимается соответствие, минимизирующее сумму расстояний между центроидами соответственных кластеров.

Отметим, что мы не рассматриваем объединение кластеров из одного интервала, поскольку предположили, что при переходе от предыдущего интервала к следующему характер кластеров изменяется незначительно. Таким образом, мы ожидаем найти в двух соседних интервалах представления каждого из $k$ «истинных» кластеров, присутствующих в потоке.

Если мы решим объединить два кластера из разных интервалов, то, разумеется, число точек в новом кластере будет равно сумме чисел точек в старых. Центроид объединенного кластера находится как взвешенное среднее центроидов исходных, где в роли веса выступает число точек в кластере. То есть, если в одном кластере $n_1$ точек, а в другом – $n_2$ точек, а их центроиды равны соответственно $\mathbf{c}_1$ и $\mathbf{c}_2$ (имеются в виду $d$-мерные векторы), то в объединенном кластере будет $n = n_1 + n_2$ точек, а его центроид определяется по формуле

$$\mathbf{c} = \frac{n_1\mathbf{c}_1 + n_2\mathbf{c}_2}{n_1 + n_2}.$$

**Пример 7.15.** Метода из примера 7.14 достаточно, когда кластеры изменяются очень медленно. Но если есть основания полагать, что центроиды кластеров перемещаются достаточно быстро, то при сопоставлении центроидов из двух соседних интервалов можно столкнуться с неоднозначной ситуацией, когда непонятно, какой из двух кластеров точнее соответствует данному кластеру из другого интервала. Один из способов защититься от такой ситуации – создавать в каждом интервале более $k$ кластеров, даже если мы знаем, что в ответ на запрос (см. раздел 7.6.5) нам придется объединить их ровно в $k$ кластеров. Например, мы могли бы выбрать $p$ много больше $k$ и объединять кластеры только тогда, когда получающийся кластер достаточно компактен в соответствии с одним из критериев, описанных в разделе 7.2.3. Или можно было бы воспользоваться иерархической стратегией и выбирать оптимальные способы объединения, так чтобы в каждом интервале всегда оставалось $p > k$ кластеров.

Предположим для определенности, что мы хотим ограничить сумму расстояний между всеми точками кластера и его центроидом. Тогда помимо количества точек и центроида мы можем включить в состав записи о кластере еще и оценку этой суммы. В момент инициализации интервала ее можно вычислить точно. Но после объединения кластеров этот параметр становится всего лишь оценкой. Пусть требуется вычислить сумму расстояний для объединенного кластера. Оставим обозначения примера 7.14 и еще обозначим $s_1$ и $s_2$ интересующие нас суммы для обоих кластеров. Тогда оценить радиус объединенного кластера можно так:

$$n_1|\mathbf{c}_1 - \mathbf{c}| + n_2|\mathbf{c}_2 - \mathbf{c}| + s_1 + s_2.$$

Таким образом, в качестве оценки расстояния между произвольной точкой $x$ и новым центроидом $\mathbf{c}$ берется сумма расстояний от этой точки до старых центроидов (последние два члена, $s_1 + s_2$, в выражении выше) плюс сумма расстояний от старых центроидов до нового (первые два члена в выражении выше). Отметим, что в силу неравенства треугольника это оценка сверху.

Другой вариант – заменить сумму расстояний суммой квадратов расстояний от точек до центроида. Если для двух кластеров эти суммы равны $t_1$ и $t_2$ соответственно, то для нового кластера сумму можно оценить как

$$n_1|\mathbf{c}_1 - \mathbf{c}|^2 + n_2|\mathbf{c}_2 - \mathbf{c}|^2 + t_1 + t_2$$

Для многомерных пространств эта оценка близка к правильной в силу «проклятия размерности».

**Пример 7.16.** В третьем примере мы предположим, что пространство неевклидово, а на число кластеров не накладывается ограничений. Заимствуем несколько приемов из алгоритма GRGPF (раздел 7.5). Конкретно, каждый кластер представляется своим кластроидом и величиной ROWSUM (сумма квадратов расстояний между всеми точками кластера и его кластроидом). Включим в состав записи о кластере информацию о множестве точек, находящихся на максимальном удалении от кластроида, а точнее их расстояния до кластроида и значения ROWSUM. Напомним, что наша цель – решить, какие кластеры следует объединять.

При объединении интервалов есть много способов объединить кластеры. Можно, например, рассматривать пары кластеров в порядке возрастания расстояний между центроидами и объединять кластеры, если сумма значений ROWSUM ниже определенного порога. Или можно выполнять объединение, если сумма значений ROWSUM, поделенная на количество точек в кластере ниже порога. Можно применить и любую другую стратегию из числа обсуждавшихся выше при условии, что мы храним данные (например, диаметр кластера), необходимые для принятия решения.

Затем мы должны выбрать новый кластроид из числа точек, наиболее удаленных от кластроидов обоих объединяемых кластеров. Для каждого кандидата в кластроиды мы можем вычислить ROWSUM по формулам из раздела 7.5.4. Мы также следуем описанной там же стратегии выбора далеких точек объединенного кластера из множества далеких точек каждого кластера и вычисления нового значения ROWSUM и расстояния до кластроида для каждой выбранной точки.

## 7.6.5. Ответы на вопросы

Как вы помните, мы предполагаем, что спрашивать нас могут о кластерах среди последних $m$ точек потока, где $m \leq N$. Из-за принятой стратегии объединения интервалов при движении назад во времени мы не всегда сможем найти набор интервалов, в точности покрывающий последние $m$ точек. Но наименьший набор интервалов, покрывающий последние $m$ точек, включает на более $2m$ последних точек. В качестве ответа мы вернем центроиды или кластроиды всех кластеров в отобранных интервалах. Результат будет хорошей аппроксимацией кластеров среди последних $m$ точек, только если статистика распределения точек от $m + 1$ до $2m$ отличается от статистики последних $m$ точек не очень сильно. Мы считаем, что это предположение справедливо. Если же статистика меняется слишком быстро, то, применив более сложную схему построения интервалов (см. раздел 4.6.6), мы сможем гарантированно найти интервалы, покрывающие не более $m(1 + \varepsilon)$ точек для любого $\varepsilon > 0$.

Отобрав желаемые интервалы, мы собираем все присутствующие в них кластеры. Далее нужно решить, какие из них объединять. В примерах 7.14 и 7.16 описаны

два подхода к объединению. Например, если требуется вернуть ровно $k$ кластеров, то можно объединять кластеры с ближайшими центроидами, пока не останется нужное количество, как в примере 7.14. Есть и другие способы, некоторые из которых перечислены в примере 7.16.

### 7.6.6. Кластеризация в параллельной среде

Теперь вкратце обсудим распараллеливание в вычислительном кластере[3]. Предположим, что имеется очень большое множество точек, и требуется распараллелить вычисление центроидов кластеров. Проще всего воспользоваться стратегией MapReduce, но в большинстве случаев накладывается ограничение: не более одной задачи-редуктора.

Для начала создадим много распределителей. Каждому распределителю назначается некое подмножество точек. Функция Map должна кластеризовать переданные ей точки. На выходе она порождает список пар ключ-значение, причем ключом всегда является 1, а значением – описание одного кластера. В разделе 7.6.2 были предложены различные формы описания, например центроид, число точек и диаметр кластера; любое из них годится.

Поскольку во всех парах ключ-значение ключи одинаковы, может существовать всего один редуктор. Он получает описания кластеров, порожденных всеми распределителями, и должен их объединить. В разделе 7.6.4 обсуждались различные стратегии окончательной кластеризации, результат которой и является выходом редуктора.

### 7.6.7. Упражнения к разделу 7.6

**Упражнение 7.6.1.** Выполните алгоритм BDMO с $p = 3$ для следующих данных в одномерном евклидовом пространстве:

1, 45, 80, 24, 56, 71, 17, 40, 66, 32, 48, 96, 9, 41, 75, 11, 58, 93, 28, 39, 77

В качестве алгоритма кластеризации возьмите метод $k$ средних с $k = 3$. Для представления кластера необходимы только центроид и количество точек.

**Упражнение 7.6.2.** Воспользовавшись кластерами, построенными в упражнении 7.6.1, верните наилучшие центроиды в ответ на запрос о кластеризации последних 10 точек.

## 7.7. Резюме

- *Кластеризация*. Кластеры часто являются полезным обобщенным представлением данных, имеющих вид множества точек в некотором пространстве. Для кластеризации нам необходима какая-то метрика. В идеале точки, принадлежащие одному кластеру, должны быть расположены близко друг к другу, а точки из разных кластеров – далеко.

[3] Не забывайте, что в этом разделе термин «кластер» употребляется в двух совершенно разных смыслах.

- *Алгоритмы кластеризации.* Существует два вида алгоритмов кластеризации. В случае иерархической кластеризации каждая точка вначале образует отдельный кластер, после чего близкие кластеры последовательно объединяются. В алгоритмах на основе отнесения поочередно рассматриваются все точки, и каждая относится к наиболее подходящему кластеру.
- *Проклятие размерности.* Поведение точек в многомерных евклидовых, а также в неевклидовых пространствах интуитивно неочевидно. В частности, случайные точки почти всегда находятся на одинаковом расстоянии друг от друга, а случайные векторы почти всегда ортогональны.
- *Центроиды и кластроиды.* В евклидовом пространстве точки кластера можно усреднять, и их среднее называется центроидом. В неевклидовом пространстве понятие «среднего» может не иметь смысла, поэтому мы вынуждены использовать одну из точек кластера в качестве репрезентативного, или типичного элемента. Этот элемент называется кластроидом.
- *Выбор кластроида.* Существует много способов определить репрезентативную точку кластера в неевклидовом пространстве. Например, можно взять точку, для которой достигает минимума либо сумма расстояний до других точек, либо сумма квадратов расстояний. Или точку, максимально удаленную от всех остальных точек кластера.
- *Радиус и диаметр.* Как в евклидовом, так и в неевклидовом пространстве мы можем определить радиус кластера, как максимум расстояний от центроида или кластроида до остальных точек кластера. Диаметр кластера можно определить как максимальное расстояние между любыми двумя его точками. Известны и другие определения, особенно радиуса, например, среднее расстояние от центроида до всех остальных точек.
- *Иерархическая кластеризация.* В этом семействе много алгоритмов, различающихся, прежде всего, в двух отношениях. Во-первых, способом выбора следующих двух кластеров, подлежащих объединению. Во-вторых, критерием остановки процесса объединения.
- *Выбор объединяемых кластеров.* Одна из стратегий выбора наилучшей пары кластеров для объединения заключается в том, чтобы взять кластеры с ближайшими центроидами или кластроидами. Другой вариант – взять два кластера, для которых достигает минимума расстояние между парой точек, по одной из каждого кластера. Третий – использовать среднее расстояние между точками из двух кластеров.
- *Критерий остановки объединения.* Иерархическая кластеризация может продолжаться до тех пор, пока не останется заранее заданное число кластеров. Или можно продолжать процесс, пока существуют два кластера, объединение которых достаточно компактно, то есть его радиус или диаметр не превышают некий порог. Еще один подход – объединять, пока результирующий кластер имеет достаточно высокую «плотность», которую можно определять по-разному, но в любом случае это количество точек, поделенное на некоторую размерную характеристику кластера, например радиус.

- *Алгоритмы k средних.* Это семейство алгоритмов на основе отнесения точек, в которых предполагается, что пространство евклидово. Заранее определяется количество кластеров $k$. После начального выбора $k$ центроидов кластеров все точки поочередно рассматриваются и относятся к ближайшему центроиду. После такого отнесения центроид кластера может сместиться, поэтому на необязательном последнем шаге все точки заново перераспределяются между кластерами, центроиды которых фиксируются в тех положениях, которые заняли после последнего пересчета.
- *Инициализация алгоритма k средних.* Один из способов получить $k$ начальных центроидов состоит в том, чтобы выбрать случайную точку, а затем еще $k - 1$ точек, каждая из которых максимально удалена от ранее выбранных. Альтернатива – начать с небольшой выборки точек и, применив иерархическую кластеризацию, сформировать из них $k$ кластеров.
- *Выбор k в алгоритме k средних.* Если число кластеров неизвестно, то можно использовать технику двоичного поиска, пытаясь выполнить кластеризацию методом $k$ средних для разных значений $k$. Наша цель – найти наибольшее $k$ такое, что при количестве кластеров, меньшем $k$, резко возрастает их средний диаметр. Число операций кластеризации при таком поиске логарифмически зависит от истинного значения $k$.
- *Алгоритм BFR.* Это вариант алгоритма $k$ средних, предназначенный для обработки данных, не помещающихся в оперативной памяти. Предполагается, что кластеры имеют нормальное распределение вдоль осей координат.
- *Представление кластеров в BFR.* Точки читаются с диска порциями. В оперативной памяти каждый кластер представляется количеством точек в нем, вектором сумм координат всех точек по каждому измерению и вектором сумм квадратов координат всех точек по каждому измерению. Другие множества точек, которые слишком далеки от центроида кластера, чтобы быть включенным в него, представляются «миникластерами» так же, как $k$ основных кластеров. Наконец точки, далекие от всех прочих точек, хранятся сами по себе и называются «сохраненными» точками.
- *Обработка точек в BFR.* Большинство точек, загружаемых в оперативную память в составе очередной порции, относятся к ближайшему кластеру, параметры которого соответственно пересчитываются. Не отнесенные ни к какому кластеру точки можно сгруппировать в новые миникластеры, а их, в свою очередь, объединить со старыми миникластерами или с сохраненными точками. После загрузки последней порции миникластеры и сохраненные точки объединяются с ближайшим кластером или признаются выбросами.
- *Алгоритм CURE.* Это также алгоритм на основе отнесения точек. Он ориентирован на евклидово пространство, но кластеры могут иметь любую форму. Алгоритм способен обрабатывать данные, не помещающиеся в оперативной памяти.

- *Представление кластеров в CURE.* Работа алгоритма начинается с кластеризации небольшой выборки точек. Затем в каждом кластере выбираются репрезентативные точки, максимально удаленные друг от друга. Цель – расположить репрезентативные точки на краях кластера. Но впоследствии эти точки пропорционально смещаются в направлении центроида кластера, т. е. оказываются внутри него.
- *Обработка точек в CURE.* После того как репрезентативные точки для каждого кластера созданы, все множество точек можно прочитать с диска и распределить по кластерам. Каждая точка относится к тому кластеру, чья репрезентативная точка окажется ближайшей.
- *Алгоритм GRGPF.* Это тоже алгоритм на основе отнесения точек. Он умеет обрабатывать данные, не помещающиеся в оперативной памяти, и не предполагает, что пространство евклидово.
- *Представление кластеров в GRGPF.* Кластер представляется количеством точек в нем, кластроидом, множеством точек, самых близких к кластроиду, и множеством точек, самых далеких от него. Близкие точки дают возможность изменять кластроид при эволюции кластера, а отдаленные – эффективно объединять кластеры при подходящих условиях. Для каждой точки хранится также ROWSUM – квадратный корень из суммы квадратов расстояний от нее до всех остальных точек кластера.
- *Дерево кластеров в GRGPF.* Представления кластеров организуются в древовидную структуру, напоминающую B-дерево. Узлами дерева обычно являются дисковые блоки, содержащие информацию о нескольких кластерах. В листовых узлах хранятся представления максимально возможного числа кластеров, а во внутренних – выборка кластроидов тех кластеров, которые находятся в нижележащих листьях. Дерево организуется так, чтобы кластеры, репрезентативные точки которых хранятся в поддереве с корнем в любом узле, были максимально близки.
- *Обработка точек в GRGPF.* После инициализации кластеров на основе выборки точек, мы включаем каждую точку в кластер с ближайшим к ней кластроидом. Благодаря древовидной структуре мы можем начать с корня и посещать дочерний узел, для которого выборочный кластроид ближе всего к данной точке. Следуя этому правилу при спуске по дереву, мы дойдем до листа и включим точку в хранящийся в нем кластер с ближайшим кластроидом.
- *Кластеризация потоков.* Для кластеризации точек медленно эволюционирующего потока можно использовать обобщение алгоритма DGIM (подсчета единиц в скользящем окне потока). В алгоритме BDMO используются интервалы, как в DGIM, причем их допустимые размеры образуют последовательность, в которой каждый следующий размер в два раза больше предыдущего.
- *Представление интервалов в BDMO.* Размер интервала – это число представляемых им точек. В самом интервале хранятся только представления

этих точек, а не сами точки. Кластер представлен числом точек, центроидом или кластроидом и прочей информацией, необходимой для объединения кластеров в соответствии с выбранной стратегией.

- *Объединение интервалов в BDMO*. Когда возникает необходимость объединить интервалы, мы ищем кластеры с наилучшим соответствием, по одному из каждого интервала, и объединяем их в пары. Если поток эволюционирует медленно, то можно ожидать, что в соседних интервалах центроиды кластеров будут отличаться очень мало, поэтому такое сопоставление имеет смысл.
- *Ответы на вопросы в BDMO*. Запрос всегда относится к суффиксу скользящего окна. Мы берем все кластеры во всех интервалах, хотя бы частично перекрывающихся с суффиксом, и объединяем их согласно некоторой стратегии. Результирующие кластеры и являются ответом на вопрос.
- *Кластеризация с применением MapReduce*. Мы можем разбить данные на порции и кластеризовать все порции параллельно в задачах-распределителях. Кластеры, сформированные распределителями, можно затем подвергнуть дополнительной кластеризации в редукторе.

# 7.8. Список литературы

Родоначальником всех исследований по кластеризации больших данных является алгоритм BIRCH, описанный в работе [6]. Алгоритм BFR изложен в [2], а алгоритм CURE – в [5].

Алгоритму GRGPF посвящена работа [3]. Необходимые сведения о B-деревьях и R-деревьях можно найти в книге [4]. Исследование кластеризации потоков заимствовано из [1].

1. B. Babcock, M. Datar, R. Motwani, L. O'Callaghan «Maintaining variance and k-medians over data stream windows», Proc. ACM Symp. on Principles of Database Systems, pp. 234–243, 2003.
2. P. S. Bradley, U.M. Fayyad, C. Reina «Scaling clustering algorithms to large databases», Proc. Knowledge Discovery and Data Mining, pp. 9–15, 1998.
3. V. Ganti, R. Ramakrishnan, J. Gehrke, A.L. Powell, J.C. French «Clustering large datasets in arbitrary metric spaces», Proc. Intl. Conf. on Data Engineering, pp. 502–511, 1999.
4. H. Garcia-Molina, J.D. Ullman, J. Widom «Database Systems: The Complete Book», Second Edition, Prentice-Hall, Upper Saddle River, NJ, 2009.
5. S. Guha, R. Rastogi, K. Shim «CURE: An efficient clustering algorithm for large databases», Proc. ACM SIGMOD Intl. Conf. on Management of Data, pp. 73–84, 1998.
6. T. Zhang, R. Ramakrishnan, M. Livny «BIRCH: an efficient data clustering method for very large databases», Proc. ACM SIGMOD Intl. Conf. on Management of Data, pp. 103–114, 1996.

# ГЛАВА 8. Реклама в Интернете

Один из самых больших сюрпризов XXI века – способность разнообразных интересных веб-приложений обеспечивать себя за счет рекламы, а не подписки. Если радио и телевидение давно научились использовать рекламу в качестве основного источника доходов, то большинство СМИ – например, журналы и газеты – вынуждены были применять смешанный подход, извлекая доходы как из рекламы, так и из подписки.

Самым рентабельным местом для размещения онлайновой рекламы являются результаты поиска, и своей эффективностью реклама во многом обязана модели «ключевых слов» (adwords), позволяющей сопоставлять поисковые запросы с объявлениями. В этой главе много внимания будет уделено алгоритмам оптимизации такого сопоставления. Алгоритмы, используемые в этой области, необычны: они жадные и «онлайновые» в весьма узком техническом смысле, который мы обсудим ниже. Поэтому мы ненадолго отвлечемся от задачи о ключевых словах и обсудим оба аспекта – жадность и онлайновость – в общем виде.

Вторая интересная проблема онлайновой рекламы касается выбора товаров, рекламируемых интернет-магазином. Эта проблема тесно связана с «коллаборативной фильтрацией», т. е. попытками найти покупателей с похожим поведением, чтобы предлагать товары, приобретенные похожими покупателями. Эта тема обсуждается в разделе 9.3.

## 8.1. Проблемы онлайновой рекламы

В этом разделе мы перечислим технические проблемы, возникающие в связи с онлайновой рекламой. Начнем с обзора типов рекламных объявлений в Интернете.

### *8.1.1. Возможности рекламы*

Интернет предлагает рекламодателю много способов довести свою рекламу до потенциального покупателя. Перечислим основные рекламные площадки.

1. Некоторые сайты, например eBay, Craig's List и сайты по продаже автомобилей предлагают размещать рекламные объявления прямо у себя – бесплатно, за деньги или за комиссионные отчисления.

2. Рекламные места имеются на многих сайтах. Рекламодатель платит фиксированную цену за *показы* (одно отображение объявления при загрузке страницы пользователем). Обычно при следующей загрузке страницы, даже тем же пользователем, будет показано другое объявление и засчитан другой показ.
3. В интернет-магазинах типа Amazon показывается много объявлений в разных контекстах. Производители рекламируемых товаров за эти показы не платят, магазин выбирает их, чтобы повысить вероятность того, что посетитель проявит интерес к товару. Такой вид рекламы мы обсудим в главе 9.
4. Рекламные объявления размещаются вместе с результатами поиска. Рекламодатели торгуются за право показать свое объявление в ответ на некоторый запрос, но платят, только если посетитель щелкнул (кликнул) по объявлению. Какое конкретно объявление показать, решает сложный процесс, который мы обсудим в этой главе. При принятии решения учитываются поисковые термы, за которые торговался рекламодатель, предложенная им цена, наблюдаемая вероятность щелчка по объявлению и общая сумма, уплаченная рекламодателем за предоставление услуги.

## 8.1.2. Прямое размещение рекламы

В тех случаях, когда рекламодателям разрешено размещать рекламу напрямую, как на сайте бесплатных объявлений Craig's List или в функции «купить сейчас» на eBay, возникает несколько проблем, которые сайту необходимо решить. Объявления показываются в ответ на поисковые термы, например «квартира в Пало-Альто». На сайте может использоваться инвертированный индекс, как в поисковых системах (см. раздел 5.1.1), позволяющий вернуть объявления, содержащие все слова, встречающиеся в запросе. Альтернативно сайт может попросить рекламодателя задать параметры объявления, сохраняемые в базе данных. Например, в объявлении о продаже подержанной машины можно выбрать из меню производителя, модель, цвет и год выпуска, т. е. разрешается использовать лишь четко определенные термы. Посетитель видит такие же меню, когда формулирует запрос.

Ранжирование объявлений несколько более проблематично, потому что нет ничего похожего на веб-ссылки, говорящие о том, какие объявления более «важны». Одна из возможных стратегий – «сначала недавние». Она справедлива, но уязвима для манипулирования: рекламодатель может вносить небольшие изменения в свои объявления через регулярные промежутки времени. Технология выявления очень похожих объявлений рассматривалась в разделе 3.4.

Другой подход – пытаться измерить привлекательность объявления. При каждом показе объявления запоминается, щелкнул по нему посетитель или нет. Предполагается, что по привлекательным объявлениям будут щелкать чаще. Однако при оценке рекламных объявлений следует учитывать несколько факторов.

1. От положения объявления в списке в большой степени зависит, щелкнут по нему или нет. У первого объявления вероятность максимальная, а дальше экспоненциально спадает.
2. Привлекательность объявления может зависеть от поисковых термов. Например, объявление о подержанной машине с откидным верхом будет более привлекательным, если в запросе встречается терм «convertible», хотя оно же может быть показано и в ответ на запросы по марке машины без указания на откидной верх.
3. Все объявления заслуживают шанса быть показанными до тех пор, пока не появится возможность более-менее точно оценить вероятность щелчка. Если в самом начале приписать каждому объявлению вероятность щелчка 0, то мы его никогда покажем и потому не узнаем, привлекательно оно или нет.

## 8.1.3. Акцидентные объявления

Такая форма рекламы в Интернете больше всего напоминает рекламу в традиционных СМИ. Реклама шевроле на страницах Нью-Йорк Таймс – это акцидентное объявление, его эффективность ограничена. Увидят его многие, но большинство увидевших не интересуются покупкой машины, уже купили машину, вообще не водят или еще по какой-то причине не обращают на объявление внимания. Тем не менее, газета, а вместе с ней и рекламодатель уже оплатила печать объявления. Показ аналогичного объявления на домашней странице Yahoo! будет относительно неэффективен по той же причине. Плата за размещения такого объявления обычно составляет доли цента за показ.

В ответ на такое отсутствие сфокусированности традиционные СМИ издают газеты и журналы по интересам. Если производитель клюшек для гольфа разместит свое объявление в журнале Golf Digest, то вероятность, что прочитавший его читатель заинтересуется, возрастет на несколько порядков. Этим и объясняется существование множества специализированных малотиражных журналов. Они имеют возможность брать гораздо большую плату за рекламу, чем универсальные издания типа ежедневной газеты. То же самое относится и к Интернету. Показ объявления клюшек для гольфа в разделе sports.yahoo.com/golf стоит гораздо дороже, чем показ того же объявления на домашней странице Yahoo! или показ рекламы шевроле в разделе, посвященном гольфу.

Однако в Интернете есть возможность настраивать акцидентную рекламу способом, недоступным печатным изданиям: использовать информацию о пользователе для определения того, какое объявление показать, независимо от просматриваемой им страницы. Если известно, что Катя любит гольф, то имеет смысл показать ей объявление о клюшках, пусть даже она читает страницу совсем на другую тему. Узнать, что Катя любит гольф, можно разными способами.

1. Возможно, она входит в группу любителей гольфа в Facebook.
2. Возможно, она часто упоминает слово «гольф» в своих почтовых сообщениях в gmail.

3. Возможно, она проводит много времени на странице Yahoo! о гольфе.
4. Возможно, она часто отправляет запросы со словами, относящимися к гольфу.
5. Возможно, она поставила закладки на сайте, где даются уроки гольфа.

Использование всех этих и многих других методов наталкивается на массу проблем, связанных с конфиденциальностью. В этой книге мы не будем пытаться их решить, тем более что на практике они, скорее всего, и не имеют решения, которое удовлетворило бы всех. С одной стороны, людям нравятся бесплатные сервисы, которые недавно стали поддерживать рекламу, а эти сервисы рассчитывают на то, что реклама будет намного эффективнее традиционной. Все согласны, что уж если реклама должна присутствовать, то лучше все-таки видеть объявления о вещах, которые посетитель мог бы использовать, чем загромождать страницу никому не интересной чепухой. С другой стороны, существует серьезный потенциал для злоупотреблений, если информация покидает пределы машин, исполняющих рекламные алгоритмы, и попадает в руки людей.

# 8.2. Онлайновые алгоритмы

Прежде чем переходить к вопросу о соответствии между рекламными объявлениями и поисковыми запросами, сделаем небольшое отступление и рассмотрим общий класс таких алгоритмов. Они называются «онлайновыми» и основаны на идее «жадности». В следующем разделе мы также приведем предварительный пример жадного онлайнового алгоритм для решения простой задачи: поиска максимального паросочетания.

## 8.2.1. Онлайновые и офлайновые алгоритмы

Типичный алгоритм работает следующим образом. Все необходимые алгоритму данные предоставляются с самого начала. Алгоритм может обращаться к данным в любом порядке. В конце работы алгоритм порождает ответ. Такие алгоритмы называются офлайновыми.

Но бывает так, что алгоритм должен принимать решение, не видя всех данных. В главе 4 рассматривалась добыча данных из потоков, когда необходимо было в любой момент времени отвечать на вопросы обо всем потоке. В худшем случае мы должны порождать какой-то ответ после поступления каждого элемента потока, т. е. принимать решения, касающиеся элемента, вообще ничего не зная о будущем. Алгоритмы такого вида называются *онлайновыми*[1].

Конкретно, выбрать объявления для показа в результатах поиска было бы просто, если бы могли сделать это в офлайновом режиме. Можно было бы изучить историю поисковых запросов за месяц, посмотреть, как рекламодатели торгова-

[1] К сожалению, мы снова сталкиваемся с многозначностью терминов, как и в случае «кластера», когда приходится разбираться во фразах вида «алгоритмы вычисления кластеров, работающие в вычислительных кластерах». В данном случае термин «онлайновый» характеризует природу алгоритма, и его не следует путать со значением слова «онлайновый» в смысле «в Интернете» в таких фразах, как «онлайновые алгоритмы для онлайновой рекламы».

лись за поисковые термы, учесть их месячные рекламные бюджеты и затем сопоставить объявления запросам таким образом, чтобы максимизировать как выручку поисковой системы, так и количество показов для каждого рекламодателя. Беда в том, что посетители не горят желанием месяц ждать ответа на свой запрос.

Таким образом, мы вынуждены использовать онлайновый алгоритм для сопоставления объявлений поисковым запросам, т. е. при поступлении запроса необходимо немедленно выбрать объявления для показа. Мы вправе использовать информацию о прошлом, например, не показывать объявление, если бюджет рекламодателя уже исчерпан, и можем анализировать текущую кликабельность объявления (отношение числа переходов по рекламной ссылке к числу ее показов). Однако о будущих поисковых запросах мы не знаем ничего. Например, не знаем, поступит ли впоследствии куча запросов с термами, на которые рекламодатель поставил большие деньги.

**Пример 8.1.** Рассмотрим очень простой пример ситуации, когда знание будущего могло бы оказаться полезным. Производитель *A* копий антикварной мебели поставил 10 центов на поисковый терм «честерфилд»[2]. Производитель мебели попроще *B* поставил 20 центов на термы «честерфилд» и «диван». Месячные бюджеты того и другого составляют $100 и больше никто за эти термы не торгуется. Сейчас начало месяца, и только что поступил запрос со словом «честерфилд». Нам разрешено включить в результаты только одно объявление.

Очевидное решение – показать объявление *B*, потому что он платит больше. Но предположим, что в этом месяце будет очень много запросов со словом «диван» и очень мало со словом «честерфилд». Тогда *A* не потратит свой бюджет, а *B* потратит все целиком, даже если мы отдадим этот запрос *A*. Точнее, если поступит хотя бы 500 запросов со словами «диван» или «честерфилд», то не будет никакого вреда и даже, возможно, окажется выгодным отдать этот запрос *A*. У *B* все равно остается возможность истратить весь свой бюджет, а заодно и *A* истратит больше. Отметим, что это рассуждение правильно и с точки зрения поисковой системы, желающей максимизировать общую выручку, и с точки зрения *A* и *B*, которые, надо думать, хотят получить столько показов, сколько позволяет бюджет.

Если бы нам было ведомо будущее, то мы знали бы, сколько в этом месяце поступит запросов со словами «диван» и «честерфилд». Если меньше 500, то этот запрос следует отдать *B*, чтобы максимизировать выручку. В противном случае лучше отдать его *A*. Но поскольку будущее покрыто мраком, то онлайновый алгоритм не всегда может справиться с задачей не хуже офлайнового.

## 8.2.2. Жадные алгоритмы

Многие онлайновые алгоритмы относятся к жадным, т. е в ответ на каждый входной элемент принимают решение, стремясь максимизировать некоторую функцию от этого элемента с учетом прошлого.

[2] Честерфилдом называется тип дивана. См., например, www.chesterfields.info.

**Пример 8.2.** Очевидный жадный алгоритм в описанной выше ситуации – отдать запрос рекламодателю, предложившему наибольшую цену и еще не исчерпавшему свой бюджет. В данном случае первые 500 запросов со словами «диван» или «честерфилд» будут отданы *B*. После этого бюджет *B* будет исчерпан, и он больше не получит ни одного запроса. Следующие 1000 запросов со словом «честерфилд» будут отданы *A*, а на запросы со словом «диван» не будет показано никакое объявление, и, следовательно, поисковая система ничего не заработает.

В худшем случае сначала поступит 500 запросов со словом «честерфилд», а затем 500 запросов со словом «диван». Офлайновый алгоритм мог бы принять оптимальное решение: отдать первые 500 запросов *A*, заработав на этом $50, а следующие 500 отдать *B* и заработать еще $100, в итоге $150. Но жадный алгоритм отдаст первые 500 запросов *B*, заработав $100, а на следующие 500 не покажет ничего и, значит, ничего и не заработает.

## 8.2.3. Коэффициент конкурентоспособности

В примере 8.2 мы видели, что онлайновый алгоритм не дает такой же хороший результат, как оптимальный офлайновый алгоритм. Лучшее, на что можно рассчитывать, – что существует некая константа $c$, меньшая 1, такая, что при любых входных данных результат онлайнового алгоритма оказывается не более чем в $c$ раз хуже результата оптимального офлайнового алгоритма. Такая константа, если она существует, называется *коэффициентом конкурентоспособности* онлайнового алгоритма.

**Пример 8.3.** На данных из примера 8.2 жадный алгоритм дает результат, составляющий 2/3 от результата оптимального алгоритма: $100 вместо $150. Это доказывает, что коэффициент конкурентоспособности не больше 2/3. Но, возможно, он меньше. Коэффициент конкурентоспособности алгоритма может зависеть от допустимых входных данных. Даже если ограничить входные данные ситуацией, описанной в примере 8.2, но разрешить варьировать ставки, то можно показать, что коэффициент конкурентоспособности жадного алгоритма не превышает 1/2. Просто нужно поднять ставку *A* до уровня 20 центов минус $\varepsilon$. Когда $\varepsilon$ стремится к 0, жадный алгоритм по-прежнему зарабатывает только $100, а выручка, принесенная оптимальным алгоритмом, стремится к $200. Можно показать, что в этом простом случае выручка никогда не бывает более чем в два раза меньше оптимальной, поэтому коэффициент конкурентоспособности действительно равен 1/2. Но такого рода доказательства мы отложим на потом.

## 8.2.4. Упражнения к разделу 8.2

**! Упражнение 8.2.1.** Популярный пример онлайнового алгоритма, максимизирующего коэффициент конкурентоспособности, – *задача о покупке лыж*[3]. Предпо-

[3] Спасибо за этот пример Анне Карлин.

ложим, что вы можете купить лыжи за $100 или взять напрокат за $10 в сутки. Вы хотите попробовать встать на лыжи, но не знаете, понравится ли. Разрешается пробовать любое число дней, а потом отказаться. Качество алгоритма измеряется удельной платой за лыжи, и мы хотим эту стоимость минимизировать.

Один из возможных онлайновых алгоритмов – «купить лыжи сразу». Если вы попробуете скатиться с горы один раз, упадете и сдадитесь, то этот алгоритм обойдется в $100 за день, тогда как оптимальным офлайновым алгоритмом было бы взять лыжи напрокат, заплатив всего $10 за тот единственный день, когда вы ими пользовались. Следовательно, коэффициент конкурентоспособности алгоритма «купить лыжи сразу» не более 1/10, и на самом деле это его точное значение, поскольку использование лыж в течение всего одного дня – худший исход для этого алгоритма. С другой стороны, онлайновый алгоритм «всегда брать лыжи напрокат» может иметь произвольно малый коэффициент конкурентоспособности. Если кататься вам понравится и вы будете делать это регулярно, то по прошествии $n$ дней заплатите $10$n$ или $10/день, тогда как оптимальный офлайновый алгоритм состоял бы в покупке лыж сразу с уплатой всего $100, или $100/$n$ в день.

А вот и вопрос: придумать онлайновый алгоритм решения задачи о покупке лыж, который дает наилучший коэффициент конкурентоспособности. Чему равен этот коэффициент?

*Подсказка:* поскольку вы в любой момент можете упасть и отказаться от катания, то единственная информация, которую онлайновый алгоритм может использовать при принятии решения, – сколько дней вы уже откатались.

# 8.3. Задача о паросочетании

Рассмотрим упрощенный вариант задачи о сопоставлении рекламных объявлений поисковым запросам. Это так называемая задача о «максимальном паросочетании» – абстрактная проблема, связанная с *двудольными графами* (так называется граф, вершины которых распадаются на два множества – левое и правое – таким образом, что любое ребро графа соединяет вершину из левого множества с вершиной из правого). На рис. 8.1 приведен пример двудольного графа. Вершины 1, 2, 3, 4 образуют левую долю, а вершины $a$, $b$, $c$, $d$ – правую.

## 8.3.1. Паросочетания и совершенные паросочетания

Пусть имеется двудольный граф. *Паросочетанием* называется такое подмножество ребер, что никакая вершина не является концом двух или более ребер. Говорят, что паросочетание *совершенное*, если в него входят все вершины. Отметим, что паросочетание может быть совершенным, только если размеры левой и правой доли одинаковы. Паросочетание, размер которого не меньше размера любого другого паросочетания в данном графе, называется *максимальным*.

**Пример 8.4.** Множество ребер $\{(1, a), (2, b), (3, d)\}$ является паросочетанием в двудольном графе на рис. 8.1. Каждый элемент этого множества – ребро графа, и ни одна вершина не встречается в нем более одного раза. Множество ребер

$$\{(1, c), (2, b), (3, d), (4, a)\}$$

является совершенным паросочетанием, оно показано жирными линиями на рис. 8.2. Каждая вершина встречается в нем ровно один раз. На самом деле, это единственное совершенное паросочетание в данном графе, хотя бывают двудольные графы с несколькими совершенными паросочетаниями. Паросочетание на рис. 8.2 также является максимальным, поскольку любое совершенное паросочетание максимально.

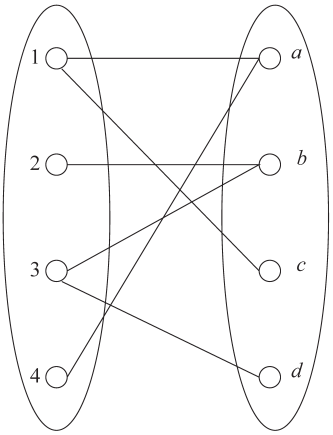



**Рис. 8.1.** Двудольный граф

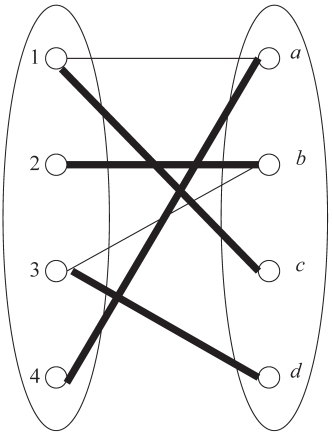



**Рис. 8.2.** Единственное совершенное паросочетание в графе на рис. 8.1

## 8.3.2. Жадный алгоритм нахождения максимального паросочетания

Офлайновые алгоритмы нахождения максимального паросочетания изучались десятки лет, для графа с $n$ вершинами это можно сделать за время, очень близкое к $O(n^2)$. Онлайновые алгоритмы решения этой задачи тоже исследовались, именно они нас и будут интересовать. Конкретно, жадный алгоритм нахождения максимального паросочетания работает следующим образом. Рассматриваем ребра в том порядке, в каком они подаются на вход. Ребро $(x, y)$ включается в паросочетание, если ни $x$, ни $y$ не являются концами какого-нибудь ребра, уже включенного в паросочетание. В противном случае ребро $(x, y)$ пропускается.

**Пример 8.5.** Рассмотрим работу жадного алгоритма для графа на рис. 8.1. Предположим, что ребра упорядочены лексикографически, т. е. сначала по левой вершине, а когда левые вершины совпадают, то по правой. Таким об-

разом, ребра рассматриваются в порядке (1, *a*), (1, *c*), (2, *b*), (3, *b*), (3, *d*), (4, *a*). Первое ребро, (1, *a*), естественно, становится частью паросочетания. Второе ребро, (1, *c*), выбирать нельзя, потому что вершина 1 уже встречалась. Третье ребро, (2, *b*), выбирается, т. к. вершины 2 и *b* пока еще не входят в паросочетание. Ребро (3, *b*) отвергается, т. к. *b* уже встречалась, затем ребро (3, *d*) добавляется, поскольку вершины 3 и *d* еще не встречались. Наконец, ребро (4, *a*) отвергается, т. к. вершина *a* уже встречалась. Таким образом, при заданном порядке ребер жадный алгоритм нашел паросочетание {(1, *a*), (2, *b*), (3, *d*)}. Как мы видели, оно не максимально.

**Пример 8.6.** Жадный алгоритм может вести себя еще хуже, чем в примере 8.5. Для графа, изображенного на рис. 8.1, при любом упорядочении, начинающемся ребрами (1, *a*) и (3, *b*) в любом порядке, в паросочетание будут включены обе эти пары, после чего включить вершины 2 и 4 уже не удастся. Следовательно, размер получившегося паросочетания – всего 2.

## 8.3.3. Коэффициент конкурентоспособности жадного алгоритма паросочетания

Можно показать, что коэффициент конкурентоспособности жадного алгоритма паросочетания из раздела 8.3.2 равен 1/2. Прежде всего, он не может быть больше 1/2. Мы уже видели, что для графа на рис. 8.1 существует совершенное паросочетание размера 4. Но если ребра подаются в порядке, описанном в примере 8.6, то алгоритм найдет только паросочетание размера 2, т. е. в два раза меньше оптимального. Поскольку коэффициент конкурентоспособности алгоритма равен минимуму по всем возможным входным данным отношения результата алгоритма на этих данных к оптимальному результату, то получается, что 1/2 – верхняя граница коэффициента конкурентоспособности.

Пусть $M_o$ – максимальное паросочетание, а $M_g$ – паросочетание, найденное жадным алгоритмом. Обозначим $L$ множество левых вершин, имеющих пару в $M_o$, но не в $M_g$. Обозначим $R$ множество правых вершин, соединенных с любой вершиной из $L$. Мы утверждаем, что для каждой вершины в $R$ есть пара в $M_g$. Допустим, что это не так, и пусть у вершины $r$ из $R$ нет пары в $M_g$. Тогда жадный алгоритм рано или поздно рассмотрит какое-то ребро $(l, r)$, где $l$ принадлежит $L$. В этот момент ни у одной вершины этого ребра еще нет пары, потому что, по предположению, жадный алгоритм не сопоставлял пару ни $l$, ни $r$. Но это наблюдение противоречит определению способа работы жадного алгоритма – он обязан составить пару $(l, r)$. Следовательно, мы заключаем, что для каждой вершины из $R$ есть пара в $M_g$.

Итак, мы знаем несколько фактов о размерах множеств и паросочетаний.

1. $|M_o| \le |M_g| + |L|$, т. к. из левых вершин только вершины, принадлежащие $L$, могут иметь пару в $M_o$, но не в $M_g$.
2. $|L| \le |R|$, поскольку в $M_o$ все вершины из $L$ имеют пару.
3. $|R| \le |\mathrm{M}_g|$, потому что у каждой вершины из $R$ есть пара в $\mathrm{M}_g$.

Из (2) и (3) следует, что $|L| \le |M_g|$. Учитывая (1), получаем, что $|M_o| \le 2|M_g|$, или $|M_g| \ge 1/2\,|Mo|$. Последнее неравенство означает, что коэффициент конкурентоспособности не меньше 1/2. Поскольку ранее мы доказали, что этот коэффициент не больше 1/2, то он в точности равен 1/2.

### 8.3.4. Упражнения к разделу 8.3

**Упражнение 8.3.1.** Пусть граф $G_n$ имеет $2n$ вершин

$$a_0, a_1, ..., a_{n-1}, \quad b_0, b_1, ..., b_{n-1}$$

и ребра, определенные следующим образом. Каждая вершина $a_i$ для $i = 0, 1, ..., n - 1$ соединена с вершинами $b_j$ и $b_k$, где

$$j = 2i \bmod n \quad \text{и} \quad k = (2i + 1) \bmod n$$

Например, граф $G_4$ имеет следующие ребра $(a_0, b_0)$, $(a_0, b_1)$, $(a_1, b_2)$, $(a_1, b_3)$, $(a_2, b_0)$, $(a_2, b_1)$, $(a_3, b_2)$ и $(a_3, b_3)$.

(*а*) Найдите совершенное паросочетание в $G_4$.

(*б*) Найдите совершенное паросочетание в $G_5$.

!! (*в*) Докажите, что для любого $n$ в графе $G_n$ существует совершенное паросочетание.

! **Упражнение 8.3.2.** Сколько совершенных паросочетаний в графах $G_4$ и $G_5$ из упражнения 8.3.1?

! **Упражнение 8.3.3.** Находит ли жадный алгоритм совершенное паросочетание в графе на рис. 8.1, зависит от порядка рассмотрения ребер. Сколько из 6! возможных порядков шести ребер позволяют найти совершенное паросочетание? Приведите простой критерий, который различает порядки, для которых совершенное паросочетание находится и не находится.

## 8.4. Задача о ключевых словах

Теперь мы переходим к фундаментальной проблеме поисковой рекламы, которую мы назвали «задачей о ключевых словах», потому что впервые она встретилась в системе Google Adwords. Затем мы обсудим жадный алгоритм «Balance», дающий хороший коэффициент конкурентоспособности. Мы проанализируем этот алгоритм для упрощенной задачи о ключевых словах.

### 8.4.1. История поисковой рекламы

Примерно в 2000 году компания Overture (позднее приобретенная Yahoo!) изобрела новый вид поиска. Рекламодатели торговались за ключевые слова (слова в поисковом запросе), а когда пользователь искал по данному ключевому слову, отображались ссылки на всех рекламодателей в порядке убывания предложенной цены. Если пользователь щелкал по ссылке рекламодателя, тот платил предложенную ранее цену.

Такой поиск очень полезен, когда пользователь ищет именно рекламные объявления, но практически бесполезен, если его интересует информация. Вспомните – в разделе 5.1.1 мы говорили, что если поисковая система не может надежно отвечать на общие запросы, то никто не захочет ей пользоваться для поиска товаров.

Спустя несколько лет Google адаптировала эту идею в своей системе Adwords. К тому времени надежность Google была общепризнанна, поэтому люди доверяли показанной в ней рекламе. Google ранжировала результаты поиска по PageRank и другим объективным критериям независимо от списка объявлений, поэтому одна и та же система была полезна и пользователю, которому нужна просто информация, и тому, кто хотел что-то купить.

Система Adwords улучшила более раннюю систему в разных направлениях, и в результате выбор объявлений стал более сложным.

1. Для каждого запроса Google показывала ограниченное количество объявлений. Если Overture просто упорядочивала все объявления для заданного ключевого слова, то Google приходилось решать, какие объявления показать и в каком порядке.
2. Пользователи системы Adwords задавали бюджет: сумму, которую они были готовы потратить за все щелчки по их объявлениям в течение месяца. Эти ограничения еще усложнили задачу сопоставления объявлений с поисковыми запросами, что мы и продемонстрировали в примере 8.1.
3. Google упорядочивала объявления не только по предложенной цене, но и по ожидаемой выручке за показ. То есть система подсчитывала коэффициент кликабельности каждого объявления на основе истории его показов. Ценность объявления вычислялась как произведение предложенной цены на коэффициент кликабельности .

## 8.4.2. Постановка задачи о ключевых словах

Разумеется, решение о том, какие объявления показывать, должно приниматься в режиме реального времени. Поэтому мы будем рассматривать только онлайновые алгоритмы решения задачи о ключевых словах. Задача ставится следующим образом:

- Дано:
  1. Множество предложений рекламодателей для поисковых запросов.
  2. Коэффициент кликабельности для каждой пары рекламодатель-запрос.
  3. Бюджеты всех рекламодателей. Мы будем предполагать, что бюджет рассчитан на месяц, хотя можно использовать любую единицу времени.
  4. Ограничение на количество объявлений, отображаемых в ответ на один поисковый запрос.
- Требуется для каждого поискового запроса вернуть такой набор рекламодателей, что:

1. Размер набора не больше ограничения на количество объявлений.
2. Каждый рекламодатель предложил цену за этот запрос.
3. У каждого рекламодателя осталось достаточно денег для оплаты щелчка по этому объявлению.

*Выручкой* от выбора объявлений называется общая ценность выбранных объявлений, где под *ценностью* понимается произведение предложенной цены на коэффициент кликабельности данного сочетания объявления и запроса. Качество онлайнового алгоритма измеряется суммарной выручкой за месяц (единицу времени, на которую рассчитаны бюджеты). Мы попытаемся измерить коэффициент конкурентоспособности алгоритмов, т. е. минимальную суммарную выручку для любой последовательности поисковых запросов, поделенную на выручку оптимального офлайнового алгоритма для той же последовательности запросов.

## 8.4.3. Жадный подход к задаче о ключевых словах

Поскольку для решения задачи о ключевых словах подходят только онлайновые алгоритмы, исследуем сначала качество очевидного жадного алгоритма. Примем ряд упрощающих предположений; наша конечная цель – показать, что существуют алгоритмы, лучшие, чем очевидный жадный. Вот эти предположения:

(*а*) для каждого запроса показывается одно объявление;
(*б*) бюджеты всех рекламодателей одинаковы;
(*в*) все коэффициенты кликабельности равны;
(*г*) предложенная цена равна 0 или 1. Вместо этого можно предположить, что ценность всех объявлений (произведение предложенной цены и коэффициента кликабельности) одинакова.

Жадный алгоритм для каждого запроса выбирает произвольного рекламодателя, предложившего за этот запрос цену 1. Коэффициент конкурентоспособности этого алгоритма равен 1/2, как видно из следующего примера.

**Пример 8.7.** Пусть имеется два рекламодателя *A* и *B* и всего два возможных запроса *x* and *y*. Рекламодатель *A* торгуется только за *x*, а *B* – за *x* и *y*. Бюджеты обоих рекламодателей равны 2. Отметим сходство этой ситуации с рассмотренной в примере 8.1, разница лишь в том, что цены, предложенные обоими рекламодателями, одинаковы, а бюджеты меньше.

Пусть поступила последовательность запросов *xxyy*. Жадный алгоритм может отдать два первых *x* рекламодателю *B*, после чего не останется никого с бюджетом, позволяющим заплатить за оба *y*. Выручка жадного алгоритма в этом случае равна 2. Однако оптимальный офлайновый алгоритм отдаст оба *x* рекламодателю *A*, оба *y* – рекламодателю *B* и получит выручку 4. Таким образом, коэффициент конкурентоспособности жадного алгоритма не превышает 1/2. Воспользовавшись той же идеей, что в разделе 8.3.3, можно показать, что для любой последовательности запросов отношение выручки жадного и оптимального алгоритмов не меньше 1/2.

**Аспекты ключевых слов, не отраженные в нашей модели**

Реальная система AdWords отличается от нашей упрощенной модели в нескольких отношениях.

*Сопоставление предложений и поисковых запросов.* В нашей упрощенной модели рекламодатели торгуются за наборы слов, и объявление рекламодателя может быть показано, только если в запросе участвуют в точности те же слова, что в ценовом предложении. На самом деле, Google, Yahoo! и Microsoft предлагают рекламодателям вариант *широкого соответствия*, при котором допускается показ объявления, даже если запрос неточно соответствует ключевым словам, указанным в заявке. В частности, множество слов в запросе может быть подмножеством или надмножеством ключевых слов и, кроме того, допускается использование в запросе слов с очень близкими значениями. В таком случае поисковая система тарифицирует показ по сложной формуле, учитывающей близость запроса к тому, за что был готов заплатить рекламодатель. В разных поисковых системах применяются различные формулы, они не публикуются.

*Взимание платы за щелчок.* В нашей упрощенной модели, когда пользователь щелкает по объявлению, с рекламодателя списывается указанная в его предложении сумма. Такая политика называется *аукционом первой цены*. На самом деле, в поисковых системах применяется более сложная система, называемая *аукционом второй цены*, при которой каждый рекламодатель платит примерно ту цену, которую указал следующий за ним (вторую максимальную цену). Например, рекламодатель, предложивший наибольшую цену, может заплатить цену, указанную рекламодателем, занявшим второе место, плюс один цент. Было показано, что аукционы второй цены менее уязвимы к манипулированию со стороны рекламодателей, чем аукционы первой цены, и приносят большую выручку поисковой системе.

## 8.4.4. Алгоритм Balance

Существует несложное усовершенствование жадного алгоритма, при котором в простом случае из раздела 8.4.3 коэффициент конкурентоспособности составит 3/4. Это алгоритм Balance, который отдает запрос тому из предложивших за него цену рекламодателей, у которого неизрасходованный бюджет максимален. Если кандидатов несколько, выбирается произвольный.

**Пример 8.8.** Рассмотрим ту же ситуацию, что в примере 8.7. Алгоритм Balance может отдать первый запрос $x$ как $A$, так и $B$, потому что оба предложили цену за $x$ и у обоих одинаковые бюджеты. Однако второй запрос $x$ должен быть отдан альтернативному рекламодателю, потому что у того бюджет больше. Первый запрос $y$ будет отдан $B$, потому что у него ненулевой бюджет и только он торговался за $y$. Последний запрос $y$ отдать некому, т. к. у $B$ не осталось денег, а $A$ не торговался. Следовательно, полная выручка алгоритма Balance на таких данных равна 3. Для сравнения отметим, что

полная выручка оптимального офлайнового алгоритма равна 4, потому что он может отдать оба запроса *x* рекламодателю *A*, оба запроса *y* – *B*. Мы заключаем, что в упрощенной задаче о ключевых словах из раздела 8.4.3 коэффициент конкурентоспособности алгоритма Balance не больше 3/4. Далее мы увидим, что при наличии всего двух рекламодателей коэффициент конкурентоспособности в точности равен 3/4, а при увеличении их количества коэффициент снижается до 0.63 (на самом деле $1-1/e$), но не меньше.

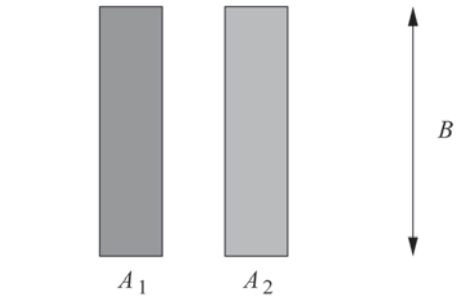
## 8.4.5. Нижняя граница коэффициента конкурентоспособности в алгоритме Balance

В этом разделе мы докажем, что в простом рассматриваемом случае коэффициент конкурентоспособности алгоритма Balance равен 3/4. В условиях примера 8.8 нужно лишь доказать, что полная выручка, заработанная алгоритмом Balance, составляет не менее 3/4 от выручки оптимального офлайнового алгоритма. Итак, рассмотрим ситуацию, когда имеются два рекламодателя $A_1$ и $A_2$, каждый с бюджетом $B$. Будем предполагать, что каждый запрос отдается рекламодателю по оптимальному алгоритму. Если для какого-то запроса это не так, то мы можем удалить его, не изменив выручку оптимального алгоритма и, возможно, уменьшив выручку Balance. Следовательно, наименьший возможный коэффициент конкурентоспособности достигается, когда последовательность запросов состоит только из объявлений, отданных оптимальным алгоритмом.

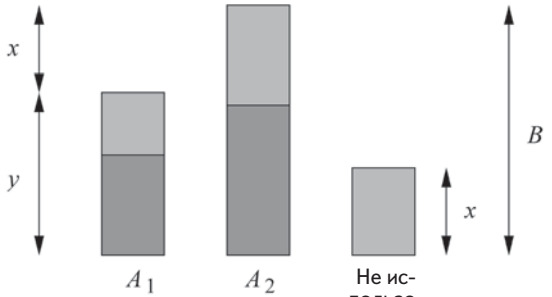
Мы также будем предполагать, что оптимальный алгоритм полностью исчерпывает бюджеты обоих рекламодателей. Если это не так, можно уменьшить бюджеты, снова приведя тот аргумент, что выручка оптимального алгоритма при этом не уменьшится, а выручка Balance может только сократиться. Такое изменение может вынудить нас использовать разные бюджеты для двух рекламодателей, но мы продолжим предполагать, что оба бюджета равны $B$. Оставляем в качестве упражнения обобщение доказательства на случай, когда бюджеты рекламодателей различаются.

На рис. 8.3 показано, как $2B$ запросов распределяются между рекламодателями обоими алгоритмами. В случае (*а*) мы видим, что оптимальный алгоритм отдает $A_1$ и $A_2$ по $B$ запросов. Предположим теперь, что те же самые запросы распределяются алгоритмом Balance. Сразу отметим, что Balance обязательно исчерпает бюджет хотя бы одного рекламодателя, скажем $A_2$. Иначе остался бы запрос, не отданный ни одному рекламодателю, хотя бюджеты обоих ненулевые. Мы знаем, что по крайней мере один рекламодатель торговался за каждый запрос, потому что этот запрос был кому-то отдан оптимальным алгоритмом. Это противоречит определению алгоритма Balance – он всегда отдает запрос, если может.

Таким образом, на рис. 8.3(*б*) мы видим, что рекламодателю $A_2$ отдано $B$ запросов. Оптимальный алгоритм мог бы отдать эти запросы как $A_1$, так и $A_2$. На том же рисунке $y$ обозначает число запросов, отданных $A_1$, а $x$ равно $B - y$. Мы хотим показать, что $y \geq x$. Это неравенство означает, что выручка алгоритма Balance не меньше $3B/2$, или 3/4 от выручки оптимального алгоритма.




**(а)** Оптимальный




**(б)** Balance

**Рис. 8.3.** Распределение запросов между рекламодателями в оптимальном алгоритме и в алгоритме Balance

Отметим, что $x$ – также число нераспределенных запросов в алгоритме Balance и что все нераспределенные запросы оптимальным алгоритмом были бы отданы $A_2$. Причина в том, что у $A_1$ бюджет никогда не кончается, поэтому за любой запрос, который оптимальный алгоритм отдал рекламодателю $A_1$, тот наверняка торговался. Поскольку за время работы алгоритма Balance бюджет $A_1$ не кончается, алгоритм обязательно отдаст этот запрос либо $A_1$, либо $A_2$.

Существует два случая в зависимости от того, кому алгоритм Balance отдаст больше запросов из тех, что оптимальный алгоритм отдал $A_1$.

1. Предположим, что не менее половины таких запросов Balance отдал $A_1$. Тогда $y \geq B/2$, поэтому наверняка $y \geq x$.
2. Предположим, что более половины таких запросов Balance отдал $A_2$. Рассмотрим последний запрос $q$, отданный $A_2$ алгоритмом Balance. В этот момент бюджет $A_2$ должен был быть никак не меньше бюджета $A_1$, иначе Balance отдел бы запрос $A1$, как сделал бы и оптимальный алгоритм. Поскольку более половины всех $B$ запросов, отданных оптимальным алгоритмом $A_1$, алгоритм Balance отдал $A_2$, то мы знаем, что в момент отдачи запроса $q$ остаток бюджета $A_2$ был меньше $B/2$. Поэтому в тот момент остаток бюджета

$A_1$ также был меньше $B/2$. Поскольку бюджет может только убывать, мы знаем, что $x < B/2$. Отсюда следует, что $y > x$, так как $x + y = B$.

Итак, в обоих случаях $y \geq x$, поэтому коэффициент конкурентоспособности алгоритма Balance равен 3/4.

## 8.4.6. Алгоритм Balance при большом числе участников аукциона

Когда рекламодателей много, коэффициент конкурентоспособности алгоритма может оказаться ниже 3/4, но ненамного. Худший случай выглядит так:

1. Имеется $N$ рекламодателей, $A_1, A_2, ..., A_N$.
2. Бюджет каждого рекламодателя равен $B = N!$.
3. Имеется $N$ запросов $q_1, q_2, ..., q_N$.
4. Рекламодатель $A_i$ торгуется за запросы $q_1, q_2, ..., q_i$ и только за них.
5. Последовательность запросов состоит из $N$ раундов. На $i$-ом раунде поступает $B$ запросов $q_i$ и больше ничего.

Оптимальный офлайновый алгоритм отдает $B$ запросов $q_i$ на $i$-ом раунде рекламодателю $A_i$ для всех $i$. Таким образом, за каждый запрос кто-то платит и полная выручка оптимального алгоритма равна $NB$.

Но алгоритм Balance на раунде 1 равномерно распределяет запросы между всеми $N$ рекламодателями, потому что все торговались за $q_1$, а Balance предпочитает участника с наибольшим остатком бюджета. Следовательно, каждый рекламодатель получит $B/N$ запросов $q_1$. Теперь рассмотрим запросы $q_2$ на раунде 2. Все рекламодатели, кроме $A_1$, торговались за эти запросы, поэтому они равномерно распределятся между $A_2, ..., A_N$, так что каждый из этих $N - 1$ участников получит $B/(N - 1)$ запросов. Картина, показанная на рис. 8.4, повторяется на каждом раунде $i = 3, 4, ...$, когда каждый из участников $A_i, ..., A_N$ получает $B/(N - i)$ запросов.

Но в конечном итоге бюджеты рекламодателей с большими номерами исчерпаются. Это произойдет на раунде с наименьшим номером $j$, при котором

$$B\left(\frac{1}{N}+\frac{1}{N-1}+...+\frac{1}{N-j+1}\right)\geq B,$$

то есть

$$\frac{1}{N}+\frac{1}{N-1}+...+\frac{1}{N-j+1}\geq 1.$$

Еще Эйлер показал, что с ростом $k$ сумма $\sum_{i=1}^{k} 1/i$ стремится к $\log_e k$. Поэтому мы можем аппроксимировать приведенное выражение значением $\log_e N - \log_e(N - j)$.

Таким образом, мы ищем такое $j$, что $\log_e N - \log_e(N - j) \approx 1$. Если заменить $\log_e N - \log_e(N - j)$ выражением $\log_e(N/(N - j))$ и потенцировать обе части уравнения $\log_e(N/(N - j)) = 1$, то получим $N/(N - j) = e$. Решив это уравнение относительно $j$, найдем

$$j = N\left(1 - \frac{1}{e}\right).$$

Приблизительно при таком значении $j$ либо у всех рекламодателей кончится бюджет, либо на оставшиеся запросы никто не будет претендовать. Поэтому выручка алгоритм Balance составит примерно $BN(1 - 1/e)$, т. е. плату за запросы на первых $j$ раундах. Следовательно, коэффициент конкурентоспособности равен $1 - 1/e$, или приблизительно 0.63.

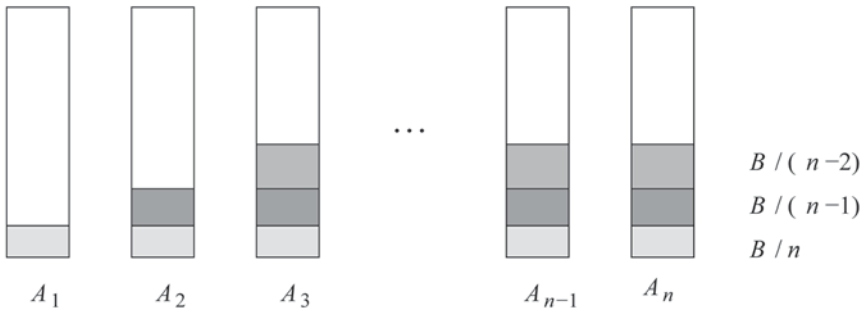



**Рис. 8.4.** Распределение запросов между $N$ рекламодателями в худшем случае

## 8.4.7. Обобщенный алгоритм Balance

Алгоритм Balance хорошо работает, если все предложенные цены равны 0 или 1. Однако на практике цена может быть произвольной, и при произвольных ценах и бюджетах Balance неправильно взвешивает размеры. В следующем примере поясняется, в чем трудность.

**Пример 8.9.** Пусть есть два рекламодателя $A_1$ и $A_2$ и один запрос $q$. Предложенные цены за $q$ и бюджеты показаны в таблице ниже

| Рекламодатель | Цена | Бюджет |
|---|---|---|
| $A_1$ | 1 | 110 |
| $A_2$ | 10 | 100 |

Если запрос $q$ поступит 10 раз, то оптимальный офлайновый алгоритм отдаст всё рекламодателю $A_2$ и получит выручку 100. Но поскольку бюджет $A_1$ больше, алгоритм Balance отдаст все десять запросов $A_1$ и выручит всего 10. На самом деле, эту идею легко обобщить и показать, что в подобных ситуациях коэффициент конкурентоспособности алгоритма Balance не может быть равен никакому значению, большему 0.

Чтобы Balance мог работать в более общих ситуациях, необходимо внести две модификации. Во-первых, следует отдавать предпочтение более высоким ценам

предложения. Во-вторых, не надо смотреть на абсолютные величины бюджетов. Вместо этого мы рассматриваем оставшуюся долю бюджета, т. е. готовы понемножку «откусывать» от бюджетов всех рекламодателей. Последнее изменение делает алгоритм Balance «менее склонным к риску»; он стремится не оставлять не потраченной слишком большую часть бюджета каждого рекламодателя. Можно показать (см. список литературы в конце главы), что у описанного ниже обобщения алгоритма Balance коэффициент конкурентоспособности равен $1 - 1/e = 0.63$.

- Предположим, что поступает запрос $q$, и рекламодатель $A_i$ предложил за него цену $x_i$ (при этом $x_i$ может быть равно 0). Предположим также, что в данный момент еще не потрачена доля $f_i$ бюджета $A_i$. Пусть $\Psi_i = x_i(1 - e^{-f_i})$. Тогда следует отдать $q$ такому рекламодателю $A_i$, для которого $\Psi_i$ достигает максимума. Если таких несколько, выбрать произвольного.

**Пример 8.10.** Рассмотрим работу обобщенного алгоритма Balance на данных из примера 8.9. При первом поступлении запроса $q$

$$\Psi_1 = 1 \times (1 - e^{-1})$$

т. к. $A_1$ предложил цену 1, а оставшаяся доля бюджета $A_1$ равна 1. Таким образом

$$\Psi_1 = 1 - 1/e = 0.63$$

С другой стороны, $\Psi_2 = 10 \times (1 - e^{-1}) = 6.3$. Следовательно, первый запрос $q$ будет отдан $A_2$.

То же самое происходит и для всех остальных $q$. Иначе говоря, $\Psi_1$ остается равным 0.63, а $\Psi_2$ убывает, но никогда не становится меньше 0.63. Даже для десятого запроса $q$, когда потрачено уже 90 % бюджета $A_2$, $\Psi_2 = 10 \times (1 - e^{-1/10})$. Напомним (см. раздел 1.3.5) разложение в ряд Тейлора $e^x = 1+x+x^2/2!+x^3/3!+\ldots$ . Отсюда

$$e^{1/10} = 1 - \frac{1}{10} + \frac{1}{200} - \frac{1}{6000} + \ldots$$

или приближенно $e^{-1/10} = 0.905$. Следовательно, $\Psi_2 = 10 \times 0.095 = 0.95$.

Мы не доказали, что коэффициент конкурентоспособности этого алгоритма равен $1 - 1/e$. Мы также оставили не доказанным еще один удивительный факт: никакой онлайновый алгоритм решения задачи о ключевых словах в том виде, как она поставлена в этом разделе, не может иметь коэффициент конкурентоспособности больше $1 - 1/e$.

## 8.4.8. Заключительные замечания по поводу задачи о ключевых словах

Алгоритм Balance, описанный выше, не учитывает, что коэффициенты кликабельности могут различаться для разных объявлений. Легко умножить предложенную

цену на коэффициент кликабельности при вычислении $\Psi_i$, при этом ожидаемая выручка достигает максимума. Мы можем даже включить информацию о коэффициенте кликабельности для каждого объявления на каждом запросе, за который предложена ненулевая цена. Когда потребуется решить судьбу запроса $q$, мы добавим в вычисление каждого $\Psi$ множитель, равный коэффициенту кликабельности объявления на этом запросе.

Еще один практический вопрос – частотная история запросов. Если, к примеру, мы знаем, что бюджет рекламодателя $A_i$ достаточно мал, так что в текущем месяце наверняка будет достаточно запросов, чтобы исчерпать его целиком, то нет смысла увеличивать $\Psi_i$, если какая-то часть бюджета $A_i$ уже потрачена. То есть мы оставляем $\Psi_i = x_i(1 - e^{-1})$ до тех пор, пока вправе ожидать, что до конца месяца поступит достаточно запросов, чтобы исчерпать бюджет $A_i$. При таком изменении поведение алгоритма Balance может ухудшиться, если последовательность запросов контролируется противником, который может внезапно прекратить поток запросов, за которые торговался $A_i$. Однако поисковые системы получают так много запросов и поступают они в таком случайном порядке, что нет необходимости предусматривать значительное отклонение от нормы.

### 8.4.9. Упражнения к разделу 8.4

**Упражнение 8.4.1.** В упрощающих предположениях из примера 8.7 допустим, что имеется три рекламодателя: $A$, $B$ и $C$ и три запроса: $x$, $y$ и $z$. Бюджеты всех рекламодателей равны 2. Рекламодатель $A$ торгуется только за $x$; $B$ – за $x$ и $y$, с $C$ – за $x$, $y$ и $z$. Отметим, что для последовательности запросов $xxyyzz$ оптимальный офлайновый алгоритм выручил бы 6, поскольку все запросы можно раздать.

! ($а$) Покажите, что жадный алгоритм отдаст по меньшей мере 4 из этих 6 запросов.

!! ($б$) Найдите такую последовательность запросов, что жадный алгоритм сможет раздать не более половины от того количества запросов, которое раздаст оптимальный алгоритм.

!! **Упражнение 8.4.2.** Обобщите доказательство из раздела 8.4.5 на случай двух рекламодателей с разными бюджетами.

! **Упражнение 8.4.3.** Покажите, как модифицировать пример 8.9, изменив цены предложения и (или) бюджеты, чтобы коэффициент конкурентоспособности оказался произвольно близким к 0.

## 8.5. Реализация алгоритма Adwords

Мы теперь имеем представление о том, как выбираются объявления, включаемые в результаты запросы, но пока ничего не говорили о проблеме поиска предложений, сделанных в ходе торговли за данный запрос. Если предметом торговли является точный набор слов в запросе, то решение сравнительно простое. Однако существует ряд расширений процесса сопоставления запроса и предложения, которые осложняют жизнь. Детали мы рассмотрим в этом разделе.

### 8.5.1. Сопоставление предложений с поисковыми запросами

В том виде, в каком мы сформулировали задачу о ключевых словах и в каком она обычно встречается на практике, рекламодатели торгуются за наборы слов. Если в поисковом запросе встречается в точности указанный набор слов в каком-то порядке, то говорят, что предложение соответствует запросу и становится кандидатом на выбор. Мы можем обойти трудности, связанные с порядком слов, сохраняя все наборы слов, за которые шла торговля, в лексикографическом (алфавитном) порядке. Список отсортированных слов образует ключ хэширования предложения, а сами предложения можно хранить в хэш-таблице, играющей роль индекса (см. раздел 1.3.2).

Слова в составе поискового запроса также предварительно сортируются. Используя отсортированный список в качестве ключа, мы находим в хэш-таблице все предложения для данного набора слов. Найти их можно быстро, потому что нужно просматривать только содержимое одной ячейки.

К тому же, вполне может статься, что вся хэш-таблица помещается в оперативной памяти. Если рекламодателей миллион, каждый торгуется за 100 запросов и запись об одном ценовом предложении занимает 100 байтов, то потребуется 10 ГБ оперативной памяти – объем, вполне доступный одной машине. Если нужно больше места, то можно распределить ячейки хэш-таблицы между несколькими машинами. Поисковый запрос хэшируется и посылается нужной машине.

На практике поисковые запросы могут поступать слишком быстро для одной машины или группы машины, совместно обрабатывающих по одному запросу за раз. В таком случае поток запросов разбивается на нужное количество частей, и каждая часть обрабатывается отдельной группой машин. На самом деле, для поиска ответа на запрос, даже вне связи с объявлениями, все равно потребуется группа параллельно работающих машин, чтобы обработку запроса можно было вести целиком в оперативной памяти.

### 8.5.2. Более сложные задачи сопоставления

Но возможности сопоставления ценовых предложений с объектами не ограничиваются случаем, когда объектами являются поисковые запросы, а критерием соответствия служит совпадение набора слов. Например, Google сопоставляет ключевые слова и с почтовыми сообщениями. В этом случае критерий соответствия не основан на равенстве множеств. Предложение, связанное с набором слов $S$, соответствует почтовому сообщению, если все входящие в $S$ слова встречаются где-то в сообщении. Такая задача сопоставления намного сложнее. Мы по-прежнему можем хранить индекс предложений в виде хэш-таблицы, но количество подмножеств слов в сообщении из сотни слов слишком велико, поэтому невозможно просмотреть все подмножества или хотя бы небольшие – состоящие, к примеру, из трех и менее слов. Существуют и другие потенциальные приложения такого рода сопоставления, которые на момент написания книги не реализованы, хотя и могли

бы быть. Все они относятся к *постоянным запросам* – таким, которые пользователи отправляют на сайт и ждут, что сайт уведомит их, если на нем произойдет что-то, отвечающее запросу. Например:

1. Twitter позволяет отслеживать все твиты данного человека. Но можно было бы разрешить пользователям задавать набор слов типа

```
ipod free music
```

   и видеть все твиты, в которых эти слова встречаются, необязательно в указанном порядке и необязательно подряд.
2. Новостные сайты часто позволяют пользователям выбирать некоторые ключевые слова или фразы, например «здравоохранение» или «Барак Обама», и получать уведомления, когда появится новость, содержащая указанное слово или последовательность слов. Эта задача проще, чем сопоставление почтовых сообщений с ключевыми словами, по нескольким причинам. Сравнение одиночных слов или последовательностей слов даже в длинной статье, занимает меньше времени, чем сравнение небольших множеств слов. К тому же, множество термов, которые разрешается искать, ограничено, т. е. «ценовых предложений» не так много. И даже если много народу захочет получать уведомления об одном и том же терме, понадобится всего одна запись в индексе, содержащая список всех заинтересованных пользователей. Но более сложная система могла бы разрешить пользователям запрашивать уведомления для произвольного множества слов в новости, как Adwords позволяет всем торговаться за слова в почтовых сообщениях.

## 8.5.3. Алгоритм сопоставления документов и ценовых предложений

Мы предложим алгоритм, который сопоставляет многие «предложения» со многими «документами». Как и раньше, предложением будет множество слов (обычно небольшое), а документом – гораздо большее множество слов, например: почтовое сообщение, твит или новость. Предполагается, что каждую секунду могут поступать сотни документов, хотя если их действительно так много, поток документов можно распределить между несколькими машинами или группами машин. Предполагается также, что предложений много, порядка сотен миллионов или миллиарда. Как обычно, мы хотим по возможности производить обработку в оперативной памяти.

Как и раньше, мы будем представлять предложение списком слов в определенном порядке. Но добавим в представление два новых элемента. Во-первых, для каждого списка слов включим *состояние* – целое число, показывающее, сколько начальных слов списка сопоставились с текущим документом. Для предложения, хранящегося в индексе, состояние всегда равно 0.

Во-вторых, хотя слова можно было бы хранить в лексикографическом порядке, объем работы уменьшится, если список упорядочить по возрастанию частоты встречаемости (самые редкие слова в начале). Но т. к. количество различных слов,

которые могут встречаться в письмах, практически не ограничено, упорядочить таким образом все слова невозможно. В качестве компромисса мы можем определить $n$ самых употребительных слов в вебе или в выборке из потока обрабатываемых документов. Здесь порядок $n$ может достигать сотен тысяч или миллиона. Эти $n$ слов сортируются по частоте и располагаются в конце списка, причем наиболее частые слова в самом конце. Можно считать, что все слова, не встречающиеся среди $n$ самых частых, являются одинаково редкими, и упорядочить их лексикографически. После этого можно будет упорядочить слова любого документа. Если слово не встречается среди $n$ самых частых, помещаем его в начало, а между собой такие слова сортируем лексикографически. Те же слова из документа, которые встречаются в списке самых частых, должны располагаться после редких слов в порядке возрастания частоты (т. е. самые частые слов в конце).

**Пример 8.11.** Рассмотрим такой документ:

```
'Twas brillig, and the slithy toves
```

«The» – самое часто встречающее слово в английском языке, а «and» встречается ненамного реже. Предположим, что «twas» входит в список частых слов, хотя его частота, конечно, меньше, чем у «the» или «and». Остальные слова в список частых не входят.

Тогда в конце списка окажутся слова «twas», «and» и «the» именно в таком порядке – по возрастанию частоты. Остальные три слова помещаются в начало списка в лексикографическом порядке. Таким образом, последовательность слов из документа будет выглядеть так:

```
brillig slithy toves twas and the
```

Предложения хранятся в хэш-таблице, ключом которой является первое слово предложения в смысле описанного выше упорядочения. В состав записи о предложении входит также информация о том, что делать, когда это предложение сопоставится с документом. Состояние равно 0 и явно не хранится. Существует еще одна хэш-таблица, в которой хранятся копии частично сопоставленных предложений, т. е. таких, для которых состояние равно как минимум 1, но меньше числа слов в множестве. Если состояние равно $i$, то ключом этой таблицы будет $(i + 1)$-ое слово. Организация обеих хэш-таблиц показана на рис. 8.5. Обработка документа производится следующим образом:

1. Отсортировать слова документа в описанном выше порядке. Устранить дубликаты.
2. Для каждого слова $w$ выполнить:
   (*a*) Используя $w$ в качестве ключа таблицы частично сопоставленных предложений, найти предложения с ключом $w$.
   (*b*) Для каждого такого предложения $b$, если $w$ – последнее слово $b$, переместить $b$ в список сопоставленных предложений.

(*c*) Если *w* – не последнее слово *b*, увеличить состояние *b* на 1 и заново хэшировать *b*, взяв в качестве ключа слово, позиция которого на 1 больше нового состояния.

(*d*) Используя *w* в качестве ключа таблицы всех предложений, найти предложения, для которых слово *w* стоит на первом месте в порядке сортировки.

(*e*) Для каждого такого предложения *b*, если в его списке присутствует всего одно слово, скопировать предложение в список сопоставленных.

(*f*) Если *b* содержит больше одного слова, добавить его с состоянием 1 в таблицу частично сопоставленных предложений, используя второе слово *b* в качестве ключа.

3. Вывести список сопоставленных предложений.

Теперь преимущества упорядочения по убыванию частоты наглядно видны. Предложение копируется во вторую хэш-таблицу, только если самое редкое из присутствующих в нем слов встречается в документе. Если бы при сравнении использовался лексикографический порядок, то во вторую таблицу копировалось бы больше предложений. Благодаря минимизации этой таблицы мы не только сократили объем работы на шагах 2(*a*) – 2(*f*), но и повысили вероятность того, что вся таблица поместится в оперативной памяти.

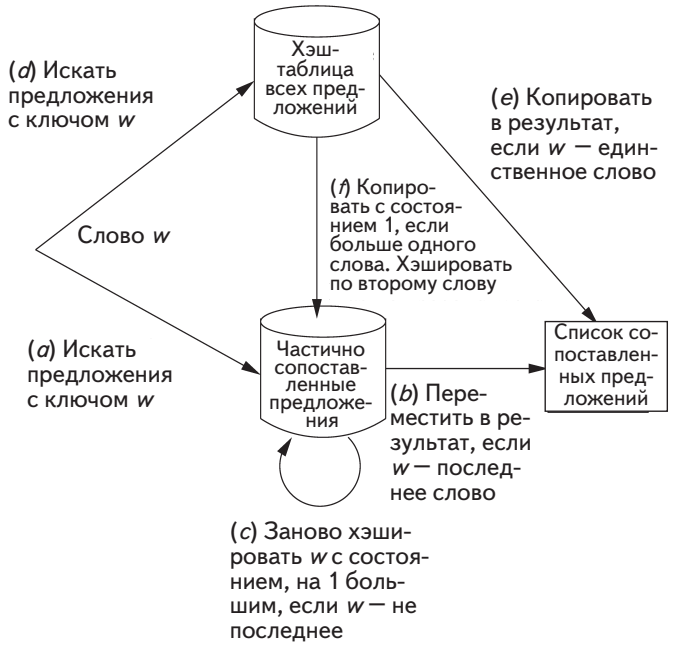



**Рис. 8.5.** Управление большим числом предложений и документов

# 8.6. Резюме

- *Таргетированная реклама.* Большое преимущество рекламы в Интернете по сравнению с обычными СМИ, например газетами, заключается в том, что в вебе можно демонстрировать пользователю те объявления, которые соответствуют его интересам. Благодаря этому многие веб-сервисы существуют исключительно за счет доходов от рекламы.
- *Онлайновые и офлайновые алгоритмы.* Традиционный алгоритм, который может увидеть все данные, перед тем как вернуть ответ, называется офлайновым. Онлайновый алгоритм должен порождать ответ для каждого элемента потока немедленно, располагая только информацией о прошлом, но ничего не зная о будущих элементах потока.
- *Жадные алгоритмы.* Многие онлайновые алгоритмы являются жадными в том смысле, что принимают решение на каждом шаге, стремясь минимизировать некоторую целевую функцию.
- *Коэффициент конкурентоспособности.* Качество онлайнового алгоритма можно измерить путем минимизации по всем возможным входным данным ценности его результата по сравнению с ценностью результата оптимального офлайнового алгоритма.
- *Паросочетания в двудольном графе.* В этой задаче речь идет о двух множествах вершин и множестве ребер, соединяющих вершины из каждого множества. Требуется найти максимальное паросочетание – наибольшее множество ребер такое, что никакие два ребра не имеют общей вершины.
- *Онлайновое решение задачи о паросочетаниях.* Один из возможных алгоритмов поиска паросочетаний в двудольном графе (да и вообще в любом графе) заключается в том, чтобы, как-то упорядочив ребра, перебирать их по очереди и для каждого ребра добавлять его в паросочетание, если ни одна из вершин нового ребра не совпадает с вершинами уже добавленных ребер. Можно доказать, что коэффициент конкурентоспособности этого алгоритма равен 1/2, т. е. количество узлов, для которых он находит пары, по сравнению с оптимальным офлайновым алгоритмом отличается не более чем вдвое.
- *Управление поиском объявлений.* Поисковая система получает ценовые предложения от рекламодателей, торгующихся за поисковые запросы. Вместе с результатами каждого запроса показываются какие-то рекламные объявления, и поисковая система взимает плату в размере предложенной цены, только если пользователь щелкнул по объявлению. У каждого рекламодателя имеется бюджет – общая сумма, которую он готов уплатить за щелчки в течение месяца.
- *Задача о ключевых словах.* Данными для задачи о ключевых словах являются множества предложений рекламодателей по некоторым поисковым запросам, бюджеты всех рекламодателей и информация об истории коэф-

фициента кликабельности для каждой пары (объявление, запрос). Вторая часть данных – поток запросов, получаемый поисковой системой. Задача состоит в том, чтобы в онлайновом режиме выбрать для каждого запроса множество объявлений фиксированного размера, так чтобы максимизировать выручку поисковой системы.

- *Упрощенная задача о ключевых словах*. Чтобы разобраться в нюансах выбора объявлений, мы рассмотрели упрощенный вариант, в котором предложения могут принимать только два значения – 0 и 1, в результатах запроса демонстрируется только одно объявление, а бюджеты всех рекламодателей одинаковы. При такой модели очевидный жадный алгоритм отдает объявление любому, кто торговался за него и не израсходовал свой бюджет. Можно показать, что коэффициент конкурентоспособности такого алгоритма равен 1/2.
- *Алгоритм Balance*. Это улучшение простого жадного алгоритма. Запрос отдается тому рекламодателю, который торговался за него и имеет максимальный остаток бюджета. Если таких несколько, выбирается произвольный.
- *Коэффициент конкурентоспособности алгоритма Balance*. Для упрощенной модели ключевых слов коэффициент конкурентоспособности алгоритма Balance равен 3/4 в случае двух рекламодателей и $1 - 1/e$, или приблизительно 63 % при большом числе рекламодателей.
- *Алгоритм Balance для обобщенной задачи о ключевых словах*. Если рекламодатели могут подавать разные ценовые предложения и иметь разные бюджеты, а коэффициенты кликабельности могут различаться для разных запросов, то алгоритм Balance отдает запрос рекламодателю с наибольшим значением функции $\Psi = x(1 - e^{-f})$. Здесь $x$ – произведение предложенной цены на коэффициент кликабельности для данного рекламодателя и данного запроса, а $f$ – неизрасходованная доля бюджета рекламодателя.
- *Реализация алгоритма Adwords*. Простейший вариант реализации работает в случае, когда все слова из оплаченного набора должны встречаться в запросе буквально. Мы можем представить запрос отсортированным списком его слов. Предложения хранятся в хэш-таблице или другой подобной структуре, ключом которой является отсортированный список слов. Тогда поисковый запрос можно сопоставить с предложениями прямым поиском в таблице.
- *Сопоставление наборов слов с документами*. В более сложном варианте задачи о ключевых словах допускается сопоставлять предложения, которые по-прежнему являются небольшими наборами слов, как и в случае поискового запроса, с более объемными документами, например почтовыми сообщениями или твитами. Предложение соответствует документу, если все его слова встречаются в документе, в любом порядке и необязательно подряд.

- *Хранение наборов слов в хэш-таблице*. Удобно хранить слова каждого предложения в списке, отсортированном в порядке возрастания частоты встречаемости. Слова документа сортируются в таком же порядке. Наборы слов хранятся в хэш-таблице, ключом которой служит первое слово в порядке возрастания частоты, т. е. самое редкое.
- *Обработка документов для сопоставления с предложениями*. Мы обрабатываем слова документа, начиная с самых редких. Наборы слов, первое слово которых является текущим, копируются во временную хэш-таблицу, ключом которой является второе слово. При просмотре наборов, уже находящихся в этой таблице, проверяется, совпадает ли слово-ключ с текущим словом. Если да, набор хэшируется заново по следующему слову. Наборы, для которых сопоставилось последнее слово, копируются в результат.

# 8.7. Список литературы

В работе [1] исследуется, как позиция объявления влияет на коэффициент кликабельности. Алгоритм Balance разработан в [2], а его применение к задаче о ключевых словах описано в [3].

1. N. Craswell, O. Zoeter, M. Taylor, W. Ramsey «An experimental comparison of click-position bias models», Proc. Intl. Conf. on Web Search and Web Data Mining pp. 87–94, 2008.
2. B. Kalyanasundaram, K.R. Pruhs «An optimal deterministic algorithm for b-matching», Theoretical Computer Science 233: 1–2, pp. 319–325, 2000.
3. A Mehta, A. Saberi, U. Vazirani, V. Vazirani «Adwords and generalized on-line matching», IEEE Symp. on Foundations of Computer Science, pp. 264–273, 2005.

# ГЛАВА 9. Рекомендательные системы

Существует обширный класс веб-приложений, в которых требуется предсказывать реакцию пользователя на предложения. Такие приложения называются рекомендательными системами. Мы начнем эту главу с обзора наиболее важных систем такого рода. Но чтобы понять, о чем речь, сразу упомянем два примера.

1. Предложение новостей читателям онлайновых изданий на основе прогнозирования их интересов.
2. Предложение покупателям Интернет-магазинов товаров, которые могли бы их заинтересовать, на основе истории прошлых покупок и (или) поиска товаров.

В рекомендательных системах применяется целый ряд технологий. Можно выделить две большие группы.

- В *системах на основе фильтрации содержимого* исследуются свойства рекомендуемых объектов. Например, если пользователь системы Netflix смотрел много ковбойских фильмов, то ему порекомендуют фильм, отнесенный в базе к жанру «ковбойский».
- *Системы коллаборативной фильтрации* рекомендуют объекты на основе сходства между пользователями и (или) объектами. Пользователю рекомендуются объекты, которые предпочитают похожие на него пользователи. В основу таких рекомендательных систем можно положить механизмы, описанные в главе 3 о поиске по сходству и в главе 7 о кластеризации. Однако одних этих технологий недостаточно, существуют новые алгоритмы, доказавшие свою эффективность именно для рекомендательных систем.

## 9.1. Модель рекомендательной системы

В этом разделе мы познакомимся с моделью рекомендательных систем, основанной на матрице предпочтений. Мы введем понятие «длинного хвоста», которое объясняет, в чем преимущество Интернет-магазина над реальным. Затем дадим краткий обзор приложений, в которых рекомендательные системы доказали свою полезность.

## 9.1.1. Матрица предпочтений

В рекомендательных системах есть два класса сущностей: *пользователи* и *объекты*. Пользователи предпочитают некоторые объекты, и эти предпочтения необходимо вывести из данных. Сами данные представлены в виде *матрицы предпочтений*, в которой каждой паре пользователь-объект сопоставлено значение, представляющее то, что известно о степени предпочтения данного объекта данным пользователем. Значения берутся из упорядоченного множества, например, это могут быть целые числа от 1 до 5, соответствующие числу звездочек в оценке, которую пользователь поставил объекту. Мы предполагаем, что матрица разреженная, т. е. ее элементы по большей части «неизвестны». Если оценка неизвестна, значит, у нас нет явной информации о предпочтениях пользователя в части этого объекта.

**Пример 9.1.** На рис. 9.1 показана матрица предпочтений, отражающая оценки, поставленные пользователями фильмам по пятибалльной шкале, в которой 5 – наивысший балл. Отсутствие значения означает, что пользователь не оценивал фильм. Расшифруем названия фильмов: HP1, HP2, HP3, – Гарри Поттер I, II и III, TW – Сумерки, SW1, SW2, SW3 – Звездные войны, эпизод 1, 2 и 3. Пользователи представлены буквами от *A* до *D*.

| | HP1 | HP2 | HP3 | TW | SW1 | SW2 | SW3 |
|---|---|---|---|---|---|---|---|
| *A* | 4 | | | 5 | 1 | | |
| B | 5 | 5 | 4 | | | | |
| C | | | | 2 | 4 | 5 | |
| D | | 3 | | | | | 3 |

**Рис. 9.1.** Матрица предпочтений, представляющая оценки фильмов по пятибалльной шкале

Отметим, что ячейки, соответствующие большинству пар пользователь-фильм, пусты, т. е. пользователь не оценивал фильм. Реальная матрица разрежена еще сильнее, потому что типичный пользователь оценивает лишь малую толику всех имеющихся фильмов.

Задача рекомендательной системы – предсказать, как будут заполнены пустые ячейки в матрице предпочтений. Например, понравится ли пользователю *A* фильм SW2? В такой крохотной матрице, как на рис. 9.1, свидетельств немного. Мы могли бы спроектировать рекомендательную систему, так чтобы она принимала в расчет такие характеристики фильмов, как продюсер, режиссер, известные актеры и даже сходство названий. В таком случае можно было бы заметить, что названия фильмов SW1 и SW2 похожи, и заключить, что раз *A* не понравился SW1, то, наверное, не понравится и SW2. Но при наличии гораздо большего объема данных можно было бы поступить и по-другому: мы могли бы заметить, что люди, оценившие одновременно SW1 и SW2, давали им похожие оценки. Отсюда можно сделать вывод, что *A* поставил бы SW2 низкую оценку, похожую на ту, что он поставил SW1.

Следует также иметь в виду несколько иную цель, которая во многих приложениях также имеет смысл. Необязательно предсказывать значение в каждой пустой ячейке матрицы предпочтений. Нужно лишь предсказать несколько значений в каждой строке, которые с большой вероятностью высоки. В большинстве приложений рекомендательная система не ранжирует все объекты, а показывает пользователю лишь те немногие, которые пользователь должен был бы оценить высоко. Не нужно даже находить все объекты с максимальными ожидаемыми оценками, хватит и достаточно большого подмножества таких объектов.

## 9.1.2. Длинный хвост

Прежде чем приступать к обсуждению основных приложений рекомендательных систем, поразмышляем о явлении *длинного хвоста*, из-за которого они и необходимы. Для физических систем доставки товаров характерна нехватка ресурсов. В реальных магазинах мало места на полках, так что покупателю можно показать лишь небольшую часть всего богатства выбора. С другой стороны, Интернет-магазин может предложить покупателю все, что имеется. Так, на полках физического книжного магазина выставлено несколько тысяч книг, а Amazon предлагает миллионы. В физической газете ежедневно печатается несколько десятков статей, а на новостном сайте – тысячи.

В физическом мире рекомендование устроено просто. Прежде всего, невозможно подстроить магазин под каждого покупателя. Поэтому выбор ассортимента определяется статистикой и ничем больше. Типичный магазин выставляет только самые популярные книги, а газета печатает лишь те статьи, которые, по ее мнению, будут интересны большинству. В первом случае, выбор продиктован данными о продажах, во втором – мнением редактора.

Различие между физическим и онлайновым миром, получившее название феномена *длинного хвоста*, изображено на рис. 9.2. По вертикальной оси откладывается *популярность* (сколько раз объект был выбран). Объекты на горизонтальной оси упорядочены по популярности. Физические организации предлагают только самые популярные объекты, расположенные слева от вертикальной линии, а онлайновые – весь спектр объектов: не только популярные, но и хвост.

Феномен длинного хвоста заставляет онлайновые организации рекомендовать объект отдельным пользователям. Невозможно показать пользователю все объекты, как в реальной организации. И нельзя ожидать, что пользователь слышал о каждом объекте, который мог бы ему понравиться.

## 9.1.3. Применения рекомендательных систем

Мы уже упомянули несколько важных применений рекомендательных систем, но теперь соберем их все воедино.

1. *Рекомендация товаров*. Пожалуй, важнее всего рекомендательные системы для Интернет-торговли. Мы уже сказали, что Amazon и другие подобные торговцы всеми силами стремятся показать каждому регулярному посети-

телю рекомендации каких-то товаров, которые могут ему понравиться. Эти рекомендации не случайны, а основаны на покупках, сделанных похожими посетителями, или на других методах, которые мы обсудим ниже.

2. *Рекомендации фильмов*. Компания Netflix рекомендует своим клиентам фильмы, которые могли бы им понравиться. Рекомендации, основаны на оценках, поставленных пользователями, – по тому же принципу, как в матрице предпочтений на рис. 9.1. Важность точного предсказания оценки настолько высока, что Netflix предложила приз в миллион долларов авторам первого алгоритма, который сумеет улучшить ее рекомендательную систему на 10 %[1]. В 2009 году, по прошествии трех лет с момента объявления, приз выиграла исследовательская группа «Bellkor's Pragmatic Chaos».
3. *Новости*. Новостные службы пытаются выявить статьи, представляющие интерес для читателей, на основе статей, которые они читали раньше. Сходство может определяться по сходству важных слов в документах или по статьям, которые читали люди с похожими пристрастиями. Те же принципы применяются к рекомендованию конкретных блогов из миллионов существующих, видеороликов на YouTube и содержимого на других подобных сайтах.

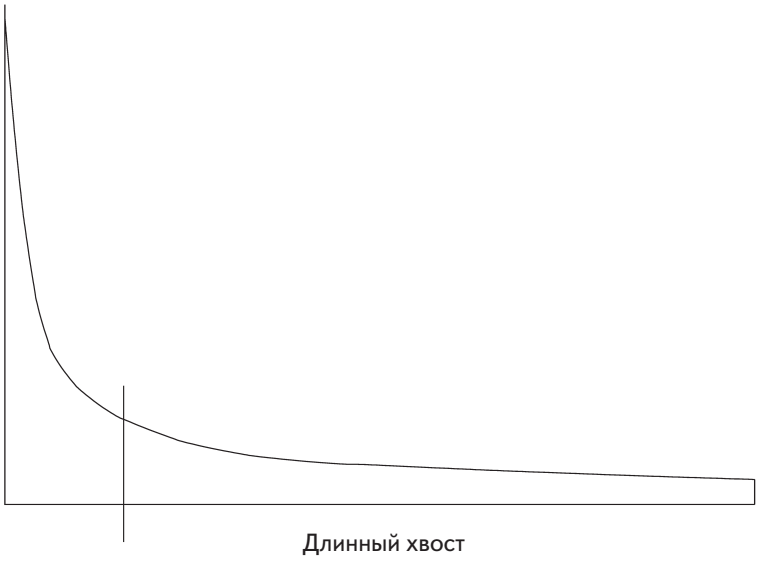



**Рис. 9.2.** Длинный хвост: физическая организация может предложить только то, что популярно, а онлайновая – весь имеющийся ассортимент

[1] Точнее требовалось, чтобы среднеквадратичная ошибка алгоритма была хотя бы на 10 % меньше, чем у алгоритма Netflix на тестовом наборе данных, содержащем реальные оценки, поставленные пользователями Netflix. Претендентам предлагался обучающий набор данных, также взятый из реальных данных Netflix.

**Книги «Into Thin Air» и «Touching the Void»**

Крайний случай того, как длинный хвост вкупе с хорошо спроектированной рекомендательной системой может повлиять на события, поведал Крис Андерсон, рассказавший о книге «Touching the Void» (Касаясь пустоты). Эта книга о жизни альпинистов в свое время не была бестселлером, но спустя много лет после ее издания вышла другая книга на ту же тему, «Into Thin Air» (И след простыл). Рекомендательная система Amazon заметила, что несколько человек купили обе книги, и начала рекомендовать «Touching the Void» тем, кто купил или интересовался книгой «Into Thin Air». Если бы не книжный Интернет-магазин, «Touching the Void» могла бы так и не найти своей аудитории, но в онлайновом мире она в конце концов обрела популярность – и даже большую, чем «Into Thin Air».

## 9.1.4. Заполнение матрицы предпочтений

Без матрицы предпочтений было бы почти невозможно рекомендовать объекты. Но собрать необходимые для ее заполнения данные зачастую нелегко. Существует два подхода к определению ценности объектов с точки зрения пользователей.

1. Можно попросить пользователей оценивать объекты. Оценки фильмов обычно так и собираются, и некоторые Интернет-магазины пытаются запрашивать оценки у покупателей. Контентные сайты, например новостные или YouTube, тоже просят пользователей оценивать объекты. Но эффективность такого подхода невелика, потому что в общем случае пользователи не горят желанием давать отклики, а информация, полученная от тех, кто согласен на это, может быть искажена самим фактом, что исходит от людей, желающих проставлять оценки.

2. Можно делать выводы из поведения людей. Самый очевидный подход: если человек что-то купил на Amazon, посмотрел ролик на YouTube или прочел новость, значит, этот объект ему «понравился». Но в такой системе оценивания есть только одно значение: 1 означает, что пользователю нравится объект. Часто встречаются матрицы предпочтений такого вида – только вместо пропуска ставится 0, который означает, что пользователь не купил или не смотрел объект. Но в данном случае не считается, что оценка 0 меньше 1; это просто отсутствие всякой оценки. В общем случае сделать выводы о предпочтениях можно и из других видов поведения, не обязательно только из покупки. Например, если посетитель сайта Amazon просматривал информацию о каком-то товаре, то можно заключить, что этот товар его интересует, даже если он его не купил.

# 9.2. Рекомендации на основе фильтрации содержимого

В начале этой главы мы говорили, что есть два основных типа рекомендательных систем:

1. *Системы на основе фильтрации содержимого* обращают внимание на характеристики объектов. Сходство объектов устанавливается путем измерения сходства их характеристик.
2. *Системы коллаборативной фильтрации* акцентируют внимание на связях между пользователями и объектами. Сходство двух объектов определяется по сходству оценок этих объектов пользователями, оценившими оба объекта.

В этом разделе мы займемся рекомендательными системами на основе фильтрации содержимого, а в следующем – системами коллаборативной фильтрации.

## 9.2.1. Профили объектов

В системе на основе фильтрации содержимого необходимо построить профиль каждого объекта – одну или несколько записей, описывающих его существенные характеристики. В простых случаях профиль включает легко определяемые характеристики. Например, рассмотрим признаки фильма, имеющие смысл для рекомендательной системы.

1. Актерский состав. Некоторые зрители предпочитают фильмы со своими любимыми актерами.
2. Режиссер. Некоторые зрители предпочитают работы определенных режиссеров.
3. Год выхода фильма. Одни зрители предпочитают старые фильмы, другие смотрят только новинки.
4. Жанр фильма. Одни смотрят только комедии, другие – только драмы или любовные истории.

Существует много других признаков, которые можно было использовать в качестве характеристик фильмов. За исключением жанра, всю эту информацию можно взять прямо из описания фильма. Понятие же жанра расплывчато. Но в некоторых обзорах фильмам приписывают условные общеупотребительные жанры. Так, в базе данных *Internet Movie Database* (*IMDB*) каждому фильму приписан один или несколько жанров. Алгоритмический способ определения жанра мы обсудим в разделе 9.3.3.

Для многих других классов объектов признаки тоже можно получить из имеющихся данных, даже если на каком-то этапе эти данные вводятся вручную. Например, производитель обычно составляет описание, в котором перечислены характеристики, существенные для данного класса изделий (скажем, размер экрана и цвет корпуса для телевизора). Для книг, как и для фильмов, тоже имеются описания, из

которых можно извлечь такие признаки, как автор, год издания и жанр. У музыкальных товаров, например компакт-дисков и MP3-файлов, имеются такие признаки, как исполнитель, композитор и жанр.

## 9.2.2. Выявление признаков документа

Существуют и другие классы объектов, для которых не очевидно, что брать в качестве признаков. Рассмотрим два примера: коллекции документов и изображения. С документами связаны особые проблемы, и в этом разделе мы обсудим технику извлечения признаков из документов. Изображения обсуждаются в разделе 9.2.3 как важный пример ситуации, когда некоторую надежду можно возлагать на признаки, сообщенные пользователем.

Рекомендательные системы могут оказаться полезны для разных типов документов. Например, каждый день публикуется много новостей, и мы не в состоянии прочитать все. Рекомендательная система может предложить материалы на интересующие пользователя темы, но как узнать, что это за темы? Веб-страницы – тоже документы. Можем ли мы порекомендовать пользователю страницы, которые могли бы его заинтересовать? Точно так же можно было бы рекомендовать блоги, если бы могли классифицировать их по темам.

К сожалению, в такого рода документах нет готовой информации о признаках. Но на практике нашлась полезная замена: идентифицировать слова, характеризующие тему документа. Как это делается, было описано в разделе 1.3.1. Сначала мы исключаем стоп-слова – несколько сотен широко употребительных слов, которые мало что говорят о теме документа. Для оставшихся слов вычисляем оценку TF.IDF каждого слова. Слова с максимальной оценкой и являются характерными для документа.

Далее можно в качестве признаков документа взять $n$ слов с наивысшими оценками TF.IDF. Значение $n$ может быть одинаково для всех документов или интерпретироваться как фиксированный процент слов в документе. Можно также считать признаками все слова, для которых оценка TF.IDF превышает заданный порог.

Итак, документы представлены множествами слов. Интуитивно мы ожидаем, что эти слова отражают темы или основные мысли документа. Например, можно предположить, что в тексте новости наибольшая оценка TF.IDF будет у имен упомянутых людей, у необычных свойств описываемого события или у географического названия места события. Для измерения сходства двух документов можно использовать естественные метрики.

1. Расстояние Жаккара между множествами слов (см. раздел 3.5.3).
2. Косинусное расстояние между множествами, рассматриваемыми как векторы (см. раздел 3.5.4).

Для вычисления косинусного расстояния (п. 2) множества слов с высокой оценкой TF.IDF интерпретируются как векторы, в которых элементы соответствуют словам. Элемент равен 1, если слово присутствует в множестве, и 0 в противном случае. Поскольку множество общих слов у любых двух документов конечно, бес-

конечномерность векторного пространства несущественна. Почти все элементы обоих векторов равны 0, а нули не влияют на скалярное произведение. При такой интерпретации скалярное произведение равно размеру пересечения двух множеств, а длина вектора – квадратному корню из числа слов в соответствующем множестве. Зная это, мы можем вычислить косинус угла между векторами как скалярное произведение, поделенное на произведение длин векторов.

**Два вида сходства документов**

Напомним, что в разделе 3.4 мы описали метод нахождения «похожих» документов с помощью разбиения на шинглы, вычисления минхэшей и LSH-хэширования. Тогда нас интересовало лексическое сходство – два документа похожи, если они содержат длинные одинаковые последовательности символов. В случае рекомендательных систем понятие сходства иное. Нас интересует только наличие в обоих документах большого количества важных слов, даже если лексически они совсем не похожи. Однако методика поиска похожих документов почти не меняется. Располагая метрикой, мы можем использовать минхэши (для расстояния Жаккара) или случайные гиперплоскости (для косинусного расстояния, см. раздел 3.7.2) и подать данные на вход алгоритма LSH для поиска пар документов, похожих в смысле наличия большого числа общих ключевых слов,

## 9.2.3. Получение признаков объектов из меток

Рассмотрим базу данных изображений в качестве примера того, как можно получить признаки объектов. Проблема заключается в том, что данные изображения, обычно массив пикселей, мало что говорят о характерных признаках. Мы можем вычислить по пикселям кое-какие простые свойства, например среднее количество красного цвета, но мало найдется пользователей, которые ищут красные картинки или предпочитают их всем остальным.

Предпринималось много попыток получить информацию о признаках объектов, предлагая пользователям *помечать* их описательными словами или фразами. Например, одно изображение с преобладанием красного цвета можно было бы пометить «площадь Тяньаньмэнь», а другое – «закат в Малибу». Это различие не улавливается современными программами анализа изображений.

Почти любые данные могут обладать признаками, выводимыми из меток. Одной из самых первых попыток разметить большой массив данных был сайт del.icio.us, позже купленный Yahoo!, который предлагал пользователям снабжать метками веб-страницы. При этом ставилась задача создать новый метод поиска, когда пользователь вводит набор меток, а система находит страницы, снабженные такими метками. Но метки можно использовать и как основу рекомендательной системы. Если подмечено, что пользователь извлекает или ставит закладки на много

страниц с определенным набором меток, то мы сможем рекомендовать ему другие страницы с такими же метками.

Проблема использования меток как средства выявления признаков заключается в том, что этот подход работает, только когда пользователи готовы тратить время на создание меток, и при этом меток настолько много, что случайные ошибки не оказывают влияния на качество предсказания.

**Метки и компьютерные игры**

Интересное направление к стимулированию разметки – «игровой» подход, впервые предложенный Луисом фон Аном (Luis von Ahn). Он дал возможность двум игрокам совместно придумывать метки изображения. Игра состоит из раундов, на каждом из которых оба игрока придумывают метки и обмениваются ими. Если метки совпали, засчитывается «выигрыш», а если нет, проводится следующий раунд с тем же изображением, и игроки снова пытаются придумать одинаковую метку. Хотя поэкспериментировать в этом направлении можно, сомнительно, что публика проявит интерес, достаточный для удовлетворения потребности в бесплатно размеченных данных.

## 9.2.4. Представление профиля объекта

Конечная цель системы на основе фильтрации содержимого – создать профили объектов, состоящие из пар признак-значение, и профили пользователей, описывающие предпочтения в виде строки матрицы предпочтений. В разделе 9.2.2 мы предложили один способ построения профиля объекта, а именно вектор из нулей и единиц, в котором 1 представляет слово из документа с высокой оценкой TF.IDF. Поскольку признаками документа являются слова и только они, мы получаем простое представление профиля.

Попытаемся обобщить векторный подход на все виды признаков. Это просто, если признаками являются наборы дискретных значений. Например, если один из признаков фильма – актерский состав, то можно представлять фильм вектором, в котором элемент равен 1, если актер в нем играет, и 0, если не играет. Точно так же можно включить в вектор всех возможных режиссеров и все возможные жанры. Каждый из этих признаков можно представить только с помощью нулей и единиц.

Существует другой класс признаков, который не удается представить булевыми векторами, – числовые признаки. Например, признаком может быть средняя оценка фильма[2], а это вещественное число. Бессмысленно заводить по одному элементу для каждого возможного значения средней оценки, к тому же это привело бы к потере структуры, неявно присутствующей в числах. Точнее, оценки, которые близки, но не одинаковы, следует считать более похожими, чем сильно различающиеся оценки. Аналогично числовые признаки различных изделий, на-

[2] Оценка – не очень надежный признак, но для примера сойдет.

пример размер экрана или емкость диска ПК, разумно считать похожими, если их значения мало отличаются.

Числовому признаку должен соответствовать один элемент вектора, представляющего объект, в котором хранится точное значение признака. Ничего страшного, если одни элементы вектора будут булевыми, а другие – целочисленными или вещественными. Мы все равно сможем вычислить косинусное расстояние между векторами, хотя должны будем при этом подумать о подходящем масштабировании небулевых элементов, чтобы они и не доминировали, и не оказывались несущественными.

**Пример 9.2.** Предположим, что единственными признаками фильма являются актерский состав и средняя оценка. Рассмотрим два фильма с пятью актерами в каждом. Два актера играют в обоих фильмах. Средняя оценка одного фильма равна 3, другого – 4. Их векторы выглядят так:

$$0\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ 3\alpha$$
$$1\ 1\ 0\ 1\ 0\ 1\ 1\ 0\ 4\alpha$$

Однако в принципе существует бесконечное число дополнительных элементов, которые представляют всех возможных актеров, не снимавшихся ни в одном фильме, и равны 0 в обоих векторах. Поскольку косинусное расстояние между векторами не зависит от нулевых элементов, то об актерах, не игравших в фильмах, можно не думать.

Последний из показанных элементов представляет среднюю оценку. Мы включили неизвестный масштабный коэффициент $\alpha$. Косинус угла между векторами вычисляется в терминах $\alpha$. Скалярное произведение равно $2 + 12\alpha^2$, а длины векторов – $\sqrt{5+9\alpha^2}$ и $\sqrt{5+16\alpha^2}$. Следовательно, косинус угла равен

$$\frac{2+12\alpha^2}{\sqrt{25+125\alpha^2+144\alpha^4}}.$$

Если положить $\alpha = 1$, т. е. брать средние оценки безо всякого масштабирования, то это выражение будет равно 0.816. Если же взять $\alpha = 2$, т. е. удвоить оценки, то косинус будет равен 0.940. То есть при $\alpha = 1$ векторы кажутся гораздо ближе по направлению. Аналогично, если положить $\alpha = 1/2$, то косинус будет равен 0.619, т. е. векторы выглядят совсем по-другому. Мы не можем сказать, какое значение $\alpha$ «правильно», но видим, что от выбора масштабного коэффициента для числовых признаков зависит, насколько похожи будут объекты.

## 9.2.5. Профили пользователей

Нам нужны не только векторы, описывающие объекты, но и векторы с такими же элементами, описывающие предпочтения пользователей. Матрица предпочтений описывает связь между пользователями и объектами. Напомним, что в непустых

элементах матрицы могут находиться как просто единицы, говорящие о том, что пользователь купил объект или что-то в этом роде, так и произвольные числа, представляющие оценку или степень симпатии пользователя к объекту.

Располагая этой информацией, лучшее, что мы можем сделать для оценивания того, какие объекты нравятся данному пользователю, – произвести некое агрегирование профилей объектов. Если матрица предпочтений может содержать только единицы, то естественным агрегатом будет среднее значение элементов векторов, представляющих профили тех объектов, для которых в строке матрицы предпочтений, соответствующей пользователю, стоят единицы.

> **Пример 9.3.** Пусть объектами являются фильмы, представленные булевыми профилями, в которых элементы соответствуют актерам. Предположим также, что элемент матрицы предпочтений равен 1, если пользователь видел фильм, и не заполнен в противном случае. Если в 20 % фильмов, которые нравятся пользователю *U*, снималась Джулия Робертс, то в профиле этого пользователя элемент, соответствующий Джулии Робертс, будет равен 0.2.

Если матрица предпочтений содержит не булевы значения, а, например, оценки от 1 до 5, то мы можем взвесить векторы, представляющие профили объектов, взяв в качестве весов предпочтения. Имеет смысл нормировать предпочтения путем вычитания среднего значения для пользователя. Тогда мы получим отрицательные веса для элементов с оценкой ниже средней и положительные – для элементов с оценкой выше средней. Мы узнаем, почему это полезно, когда в разделе 9.2.6 будем обсуждать, как найти объекты, которые могут понравиться пользователю.

> **Пример 9.4.** Рассмотрим ту же информацию о фильмах, что в примере 9.3, но теперь будем предполагать, что в матрице предпочтений непустые элементы содержат оценки от 1 до 5. Пусть среднее значение оценок, поставленных пользователем *U*, равно 3. Джулия Робертс играла в трех фильмах, получивших оценки 3, 4 и 5. Тогда в профиле *U* элемент, соответствующий Джулии Робертс, будет иметь значение, полученное усреднением чисел 3 – 3, 4 – 3 и 5 – 3, т. е. 1.
>
> С другой стороны, для пользователя *V* средняя оценка равна 4, и он также оценил три фильма с участием Джулии Робертс (неважно, совпадают они с фильмами, которые оценивал *U*, или нет). Пользователь *V* поставил этим трем фильмам оценки 2, 3 и 5. В профиле пользователя *V* элемент, соответствующий Джулии Робертс, равен среднему 2 – 4, 3 – 4 и 5 – 4, т. е. –2/3.

## 9.2.6. Рекомендование объектов пользователям на основе содержимого

Имея векторы профилей для пользователей и объектов, мы можем оценить степень симпатии пользователя к объекту, вычислив косинусное расстояние между векто-

рами пользователя и объекта. Как и в примере 9.2, имеет смысл масштабировать небулевы элементы. Для распределения профилей объектов по ячейкам можно применить методы случайных гиперплоскостей и LSH-хэширования. А зная, какому пользователю требуется рекомендовать объекты, мы можем применить те же два метода, чтобы определить, в каких ячейках искать объекты, для которых косинусное расстояние с данным пользователем невелико.

**Пример 9.5.** Рассмотрим сначала данные из примера 9.3. Элемент профиля пользователя, соответствующий некоторому актеру, пропорционален вероятности того, что актер играет в фильме, который нравится пользователю. Следовательно, наилучшие рекомендации (с наименьшим косинусным расстоянием) нужно дать фильмам, в которых играют многие актеры, снимавшиеся во многих нравящихся данному пользователю фильмах. Поскольку ничего, кроме актерского состава, мы о фильме не знаем, то, вероятно, это лучшее, что можно сделать[3].

Теперь обратимся к примеру 9.4. Как мы отметили, элементы вектора пользователя, соответствующие актерам, снимавшимся в фильмах, которые пользователю нравятся, положительны, а снимавшимся в фильмах, которые ему не нравятся, – отрицательны. Рассмотрим фильм, в котором снималось много актеров, нравящихся пользователю, и мало или совсем не снимались несимпатичные ему актеры. Косинус угла между векторами пользователя и фильма будет большим и положительным. Это означает, что угол близок к нулю, а, следовательно, косинусное расстояние между векторами мало.

Далее рассмотрим фильм, в котором снималось много симпатичных и много несимпатичных пользователю актеров. В этом случае косинус угла между векторами пользователя и фильма близок 0, поэтому угол между ними близок к 90 градусам. Наконец, возьмем фильм, в котором играют в основном актеры, не нравящиеся пользователю. Тогда косинус будет большим и отрицательным, а угол между векторами близок к 180 градусам – максимально возможному косинусному расстоянию.

## 9.2.7. Алгоритм классификации

Существует совершенно другой подход к построению рекомендательной системы с использованием профилей объектов и матрицы предпочтений – рассматривать ее как задачу машинного обучения. Будем трактовать имеющиеся данные как обучающий набор и для каждого пользователя построим классификатор, который предсказывает оценки всех объектов. Есть много разных классификаторов, и подробно изучать эту тему здесь мы не станем. Но следует знать о самой возможности разработки классификатора для выработки рекомендаций, поэтому вкратце рассмотрим один часто применяемый – решающие деревья.

[3] Отметим, что тот факт, что все элементы вектора пользователя малы, не влияет на рекомендацию, потому что при вычислении косинусного расстояния производится деление на длину вектора. Конечно, векторы пользователей гораздо короче векторов фильмов, но значение имеет только направление вектора.

*Решающее дерево* представляет собой множество узлов, организованных в виде двоичного дерева. В листовых узлах находятся решения; в нашем случае «нравится» или «не нравится». В каждом внутреннем узле хранится условие, по которому классифицируются объекты; в нашем случае условием будет предикат, включающий один или несколько признаков объекта.

Для классификации объекта мы начинаем обход с корня и применяем к объекту хранящийся в корне предикат. Если он равен true, переходим в левый дочерний узел, если false – в правый. Повторяем этот процесс для каждого посещенного узла, пока не дойдем до листового. Листовый узел классифицирует объект как понравившийся или не понравившийся.

Для построения решающего дерева требуется выбирать, какой предикат поместить в каждый внутренний узел. Существует много способов выбрать оптимальный предикат, но конечная цель всегда одинакова: один из дочерних узлов должен получить все или хотя бы большую часть положительных примеров из обучающего набора (в нашем случае объектов, нравящихся пользователю), а второй узел – все или большую часть отрицательных примеров (объектов, которые пользователю не нравятся).

Выбрав предикат для узла *N*, мы разделяем объекты на две группы: удовлетворяющие и не удовлетворяющие этому предикату: Для каждой группы снова ищем предикат, наилучшим образом разделяющий положительные и отрицательные примеры. Найденные предикаты сохраняются в дочерних узлах *N*. Процесс разделения примеров и построения дочерних узлов может продолжаться долго. Мы останавливаемся и создаем листовый узел, если группа в очередном узле однородна, т. е. содержит только положительные или только отрицательные примеры.

Однако иногда имеет смысл остановиться и создать узел с мажоритарным решением, даже если группа содержит примеры обоих видов. Причина в том, что статистическая значимость небольшой группы может быть недостаточна, чтобы полагаться на нее. Поэтому у описанной стратегии имеется вариант: создать ансамбль решающих деревьев, каждое со своими предикатами, но разрешить деревьям быть глубже, чем оправдано имеющимися данными. Такие деревья называют переобученными. Для классификации объекта мы применяем все входящие в ансамбль деревья и смотрим, как они проголосуют. Этот вариант мы рассматривать не будем, но приведем простой пример решающего дерева.

**Пример 9.6.** Предположим, что объектами являются новости, а признаками – слова с высокой оценкой TF.IDF (ключевые слова) в документах. Пусть имеется пользователь *U*, которому нравятся статьи про бейсбол, кроме статей о команде New York Yankees. Элемент строки матрицы предпочтений для *U* содержит 1, если *U* читал статью, и пуст, если не читал. Будем интерпретировать единицы как «нравится», а пустые элементы – как «не нравится». В роли предикатов будут выступать булевы выражения, составленные из ключевых слов.

Поскольку *U*, вообще говоря, любит бейсбол, мы, возможно, обнаружим, что наилучшим предикатом для корня дерева является «homerun» OR («batter» AND «pitcher»)[4]. Этому предикату будут удовлетворять, в основном, положительные примеры (новости, содержащие 1 в строке, соответствующей *U* в матрице предпочтений), а не будут удовлетворять, в основном, отрицательные (незаполненные элементы в строке пользователя *U*). На рис. 9.3 показан корень и остальные узлы решающего дерева.

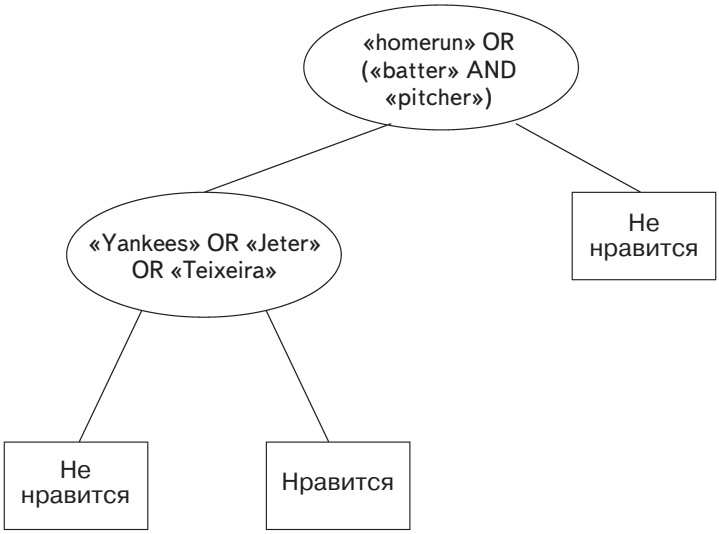



**Рис. 9.3.** Решающее дерево

Предположим, что группа новостей, не удовлетворяющих предикату, включает настолько мало положительных примеров, что можно заключить, что все такие объекты принадлежат классу «не нравится». Тогда можно сделать листовый узел с решением «не нравится» правым потомком корня. Однако среди новостей, удовлетворяющих предикату, есть много таких, которые *U* не нравятся, а именно те, где упоминается команда Yankees. Поэтому для левого потомка корня мы строим еще один предикат. Возможно, наилучшим индикатором того, что новость относится к бейсболу и касается Yankees, окажется предикат «Yankees» OR «Jeter» OR «Teixeira». На рис. 9.3 мы видим слева от корня узел, применяющий этот предикат. Оба потомка являются листовыми узлами, т. к. можно предполагать, что объекты, удовлетворяющие этому предикату, преимущественно отрицательные, а не удовлетворяющие – преимущественно положительные.

К сожалению, построение классификатора любого типа занимает много времени. Если мы захотим воспользоваться решающими деревьями, то понадобится по одному дереву на каждого пользователя. Для построения дерева придется не толь-

[4] Homerun – удар, после которого бьющий пробегает через все базы и возвращается в дом. Pitcher – питчер, игрок защищающейся команды, подающий мяч. Batter – игрок нападения с битой, бьющий. – *Прим. перев.*

ко проанализировать профили всех объектов, но и рассмотреть много разных предикатов, а это могут быть сложные комбинации признаков. Поэтому такой подход пригоден лишь для относительно небольших задач.

## 9.2.8. Упражнения к разделу 9.2

**Упражнение 9.2.1.** Для трех компьютеров *A*, *B*, *C* определены следующие числовые признаки:

| Признак | A | B | C |
|---|---|---|---|
| Быстродействие процессора | 3.06 | 2.68 | 2.92 |
| Емкость диска | 500 | 320 | 640 |
| Объем оперативной памяти | 6 | 4 | 6 |

Можно считать, что этими значениями определяется вектор каждого компьютера; например, вектором *A* будет [3.06, 500, 6]. Мы можем вычислить косинусные расстояния между каждой парой векторов, но если ничего не масштабировать, то емкость диска доминирует, так что различия между остальными элементами едва заметны. Пусть для быстродействия процессора масштабный коэффициент равен 1, для емкости диска – $\alpha$, а для объема оперативной памяти – $\beta$.

(*а*) Выразите через $\alpha$ и $\beta$ косинусы углов между векторами для каждой пары компьютеров.

(*б*) Каковы углы между векторами при $\alpha = \beta = 1$?

(*в*) Каковы углы между векторами при $\alpha = 0.01$, $\beta = 0.5$?

! (*г*) Один из справедливых способов задания масштабных коэффициентов – сделать каждый обратно пропорциональным среднему значению соответствующего элемента. Каковы при этом будут значения $\alpha$ и $\beta$ и углы между векторами?

**Упражнение 9.2.2.** Другой способ масштабирования элементов вектора – начать с нормировки векторов, т. е. вычислить среднее для каждого элемента и вычесть его из значения этого элемента в каждом векторе.

(*а*) Нормируйте векторы для всех трех компьютеров, описанных в упражнении 9.2.1.

!! (*б*) В этом упражнении нет трудных вычислений, но требуется серьезно подумать о том, что же такое угол между векторами. Если все элементы неотрицательны, как в упражнении 9.2.1, то угол между векторами не может быть больше 90 градусов. Но когда мы нормируем векторы, то могут (и должны) появиться отрицательные элементы, поэтому угол может оказаться любым в диапазоне от 0 до 180 градусов. Кроме того, средние значения каждого элемента теперь равны 0, поэтому рекомендация, приведенная в части (*г*) упражнения 9.2.1 – масштабировать обратно пропорционально среднему, – теряет смысл. Предложите способ нахождения подходящего масштаба для каждого элемента нормированного вектора. Как бы вы интерпретировали

большой и малый угол между нормированными векторами? Каковы будут углы между нормированными векторами для данных из упражнения 9.2.1?

**Упражнение 9.2.3.** Некий пользователь поставил трем компьютерам из упражнения 9.2.1 такие оценки: $A$ – 4 звезды, $B$ – 2 звезды, $C$ – 5 звезд.

(*а*) Нормируйте оценки этого пользователя.

(*б*) Вычислите профиль пользователя, если данные о быстродействия процессора, емкости диска и объема оперативной памяти взяты из упражнения 9.2.1.

# 9.3. Коллаборативная фильтрация

Теперь мы займемся совершенно другим подходом к выработке рекомендаций. Вместо того чтобы определять сходство объектов по их признакам, мы будем анализировать сходство оценок, поставленных этим объектам пользователям. То есть вместо вектора профиля объекта мы возьмем соответствующий ему столбец матрицы предпочтений. А вместо того чтобы с муками изобретать вектор профиля пользователя, представим пользователя его строкой в матрице предпочтений. Пользователи считаются похожими, если их векторы близки с точки зрения некоторой метрики, например расстояния Жаккара или косинусного расстояния. Тогда для выработки рекомендации пользователю $U$ мы смотрим, какие пользователи больше всего похожи на $U$ в описанном выше смысле, и рекомендуем те объекты, которым им нравятся. Процесс выявления похожих пользователей и рекомендования того, что нравится похожим пользователям, называется *коллаборативной фильтрацией*.

## 9.3.1. Измерение сходства

Прежде всего, нужно ответить на вопрос о том, как измерить сходство пользователей или объектов по их строкам или столбцам в матрице предпочтений. Ниже на рис. 9.4 повторена таблица на рис. 9.1. Данных слишком мало, чтобы на их основе делать надежные выводы, зато они позволяют наглядно продемонстрировать некоторые подводные камни, возникающие при выборе метрики. Обратите особое внимание на пользователей $A$ и $C$. Они оценили одни и те же фильмы, но придерживаются диаметрально противоположных мнений. Поэтому хорошая метрика должна была бы развести их на большое расстояние. Рассмотрим несколько альтернативных метрик.

| | HP1 | HP2 | HP3 | TW | SW1 | SW2 | SW3 |
|---|---|---|---|---|---|---|---|
| *A* | 4 | | | 5 | 1 | | |
| B | 5 | 5 | 4 | | | | |
| C | | | | 2 | 4 | 5 | |
| D | | 3 | | | | | 3 |

**Рис. 9.4.** Повтор матрицы предпочтений на рис. 9.1

## Расстояние Жаккара

Можно было бы игнорировать значения элементов матрицы и обращать внимание только на множества оцененных объектов. Если бы в матрице предпочтений отражался только факт покупки, то эта была бы подходящая метрика. Но если представлены более детальные оценки, то при вычислении расстояния Жаккара теряется важная информация.

**Пример 9.7.** Размер пересечения *A* и *B* равен 1, а размер объединения – 5. Следовательно, коэффициент сходства Жаккара равен 1/5, а расстояние Жаккара – 4/5, т. е. пользователи находятся очень далеко друг от друга. Для *A* и *C* коэффициент Жаккара равен 2/4, так что расстояние Жаккара равно 1/2. Поэтому *A* находится ближе к *C*, чем к *B*. Но интуитивно такой вывод кажется неправильным. *A* и *C* совершенно не согласны в части оценки просмотренных фильмов, тогда как *A* и *B* понравился один из фильмов, который они оба видели.

## Косинусное расстояние

Будем рассматривать незаполненные ячейки как значение 0. Это решение не бесспорно, потому что отсутствие оценки при этом оказывается больше похожим на «не нравится», чем на «нравится».

**Пример 9.8.** Косинус угла между *A* и *B* равен

$$\frac{4\times 5}{\sqrt{4^2+5^2+1^2}\sqrt{5^2+5^2+4^2}} = 0.380 .$$

Косинус угла между *A* и *C* равен

$$\frac{5\times 2+1\times 4}{\sqrt{4^2+5^2+1^2}\sqrt{2^2+4^2+5^2}} = 0.322 .$$

Поскольку большой положительный косинус соответствует малому углу и, следовательно, малому расстоянию, то эта метрика говорит, что *A* немного ближе к *B*, чем к *C*.

## Округление данных

Мы могли бы попытаться исключить кажущееся сходство между фильмами, получившими высокие и низкие оценки, округлив оценки. Например, можно было бы заменить 3, 4 и 5 на 1, а оценки 1 и 2 считать как «не оценен». Тогда матрица предпочтений приняла бы вид, показанный на рис. 9.5. Теперь расстояние Жаккара между *A* и *B* равно 3/4, а между *A* и *C* – 1, т. е. *C* кажется дальше от *A*, чем *B*, что согласуется с интуицией. Применив к матрице на рис. 9.5 косинусное расстояние, придем к тому же выводу.

| | HP1 | HP2 | HP3 | TW | SW1 | SW2 | SW3 |
|---|---|---|---|---|---|---|---|
| *A* | 1 | | | 1 | | | |
| B | 1 | 1 | 1 | | | | |
| C | | | | | 1 | 1 | |
| D | | 1 | | | | | 1 |

**Рис. 9.5.** Предпочтения 3, 4, 5 заменены на 1, а 1 и 2 стерты

## Нормировка оценок

Если нормировать оценки, вычтя из каждой среднюю оценку, поставленную данным пользователем, то низкие оценки станут отрицательными, а высокие – положительными. Если затем вычислить косинусное расстояние, то окажется, что векторы пользователей, придерживающихся противоположных взглядов на фильмы, направлены почти в противоположные стороны, т. е. можно считать, что пользователи удалены на максимально возможное расстояние. При этом для пользователей с похожими мнениями угол между векторами будет относительно небольшим.

**Пример 9.9.** На рис. 9.6 показана та же матрица, что на рис. 9.4, после нормировки оценок. Интересно, что оценки *D* по существу исчезли, т. к. с точки зрения косинусного расстояния 0 не отличается от отсутствия оценки. Отметим, что *D* ставил только оценку 3 и не различал фильмы, так что вполне возможно, что его оценки не стоит принимать всерьез.

Вычислим косинус угла между *A* и *B*:

$$\frac{(2/3)\times(1/3)}{\sqrt{(2/3)^2+(5/3)^2+(-7/3)^2}\sqrt{(1/3)^2+(1/3)^2+(-2/3)^2}} = 0.092$$

| | HP1 | HP2 | HP3 | TW | SW1 | SW2 | SW3 |
|---|---|---|---|---|---|---|---|
| *A* | 2/3 | | | 5/3 | –7/3 | | |
| B | 1/3 | 1/3 | –2/3 | | | | |
| C | | | | –5/3 | 1/3 | 4/3 | |
| D | | 0 | | | | | 0 |

**Рис. 9.6.** Матрица предпочтения на рис. 9.1 после нормировки

Косинус угла между *A* и *C* равен

$$\frac{(5/3)\times(-5/3)+(-7/3)\times(1/3)}{\sqrt{(2/3)^2+(5/3)^2+(-7/3)^2}\sqrt{(-5/3)^2+(1/3)^2+(4/3)^2}} = -0.559$$

Заметим, что при таком вычислении расстояния пользователи *A* и *C* отстоят гораздо дальше, чем *A* и *B*, и ни одна пара не находится очень близко друг к другу. Оба эти наблюдения согласуются с интуицией, если учесть, что *A* и *C* ни в чем не согласны по поводу фильмов, которые смотрели оба, тогда как *A* и *B* поставили похожие оценки одному и тому же фильму.

## 9.3.2. Двойственность сходства

Можно считать, что матрица предпочтений сообщает информацию о пользователях, об объектах или о том и другом. Важно понимать, что любой метод поиска похожих пользователей, предложенный в разделе 9.3.1, можно применить к столбцам матрицы предпочтений и найти похожие объекты. На практике эта симметрия нарушается двумя способами.

1. Мы можем использовать информацию о пользователях для рекомендования объектов. То есть, зная пользователя, ищем похожих на него, возможно, с применением методов из главы 3. И при выработке рекомендации полагаемся на решения, принятые похожими пользователями, т. е. рекомендуем объекты, которые большинство из них купило или высоко оценило. Но симметрии здесь нет. Даже если мы найдем пары похожих объектов, придется выполнить дополнительный шаг, чтобы рекомендовать объекты пользователям. Этот момент будет подробнее объяснен ниже.
2. Существует разница между типичным поведением пользователей и объектов в части сходства. Интуитивно кажется, что объекты можно классифицировать на основе простых понятий. Например, музыкальное произведение обычно относятся к какому-то одному жанру. Не может быть, что одно и то же произведение является и роком 1960-х годов, и барочной музыкой XVIII века. С другой стороны, есть люди, которым нравится и рок шестидесятых и барокко, так что они покупают то и другое. Отсюда следует, что гораздо проще найти похожие объекты, т. к. они принадлежат одному жанру, чем распознать двух похожих пользователей, потому что оба могут любить один жанр, но полностью расходиться во мнениях относительно другого.

В п. 1 мы предложили способ предсказания значения в матрице предпочтений для пользователя $U$ и объекта $I$: найти $n$ пользователей (для некоторого наперед заданного $n$), максимально похожих на $U$, и взять средние оценки, поставленные ими объекту $I$, учитывая лишь тех, кто оценивал $I$. В общем случае имеет смысл сначала нормировать матрицу, т. е. для каждого из $n$ пользователей вычесть среднюю оценку из оценки, поставленной каждому объекту $i$. Затем усредняем разности для тех пользователей, кто оценивал $I$, и прибавляем полученное значение к результату усреднения оценок $U$ по всем оцененным им объектам. Такая нормировка корректирует оценку в случае, когда $U$ ставит очень высокие или очень низкие оценки или когда значительная часть похожих пользователей, оценивших $I$ (каковых может быть немного), ставят, в основном, очень высокие или очень низкие оценки.

Двойственное решение заключается в том, чтобы воспользоваться сходством объектов для оценки элемента матрицы для пользователя $U$ и объекта $I$. Находим $m$ объектов, максимально похожих на $I$ (для некоторого наперед заданного $m$), и усредняем оценки, поставленные $U$, по этим $m$ объектам. Как и при определении сходства пользователей, рассматриваем только те из $m$ объектов, которые $U$ оценивал, и, вероятно, имеет смысл сначала нормировать оценки.

Отметим, что при любом подходе к предсказанию элементов матрицы предпочтений недостаточно найти только один элемент. Чтобы рекомендовать объекты пользователю *U*, мы должны оценить все элементы соответствующей ему строки матрицы или, по крайней мере, найти все или большинство пустых элементов этой строки, для которых предсказанная оценка высока. При этом надо решить, от чего отталкиваться: от похожих пользователей или от похожих объектов.

- Если мы ищем похожих пользователей, то это нужно будет сделать только один раз для пользователя *U*. Имея множество похожих пользователей, мы можем оценить все пустые элементы матрицы предпочтений в строке *U*. Если начинаем с похожих объектов, то должны вычислить похожие почти для всех объектов и только потом сможем заполнить пустые элементы в строке *U*.
- С другой стороны, вычисление сходства объектов дает более надежную информацию из-за описанного выше явления: проще найти объекты одного жанра, чем пользователей, которым нравятся объекты только одного жанра.

Какой бы метод ни выбрать, необходимо заранее вычислить предпочитаемые объекты для каждого пользователя, а не ждать завершения вычисления для принятия решения. Поскольку матрица предпочтений изменяется медленно, обычно достаточно пересчитывать ее сравнительно редко и считать, что между пересчетами она не меняется.

## 9.3.3. Кластеризация пользователей и объектов

Найти сходство между пользователями или объектами трудно, потому что у нас мало информации о парах пользователь-объект в разреженной матрице предпочтений. Продолжая тему раздела 9.3.2, можно предположить, что даже если два объекта относятся к одному жанру, вероятно, найдется очень мало пользователей, купивших или оценивших оба. И аналогично, даже если двум пользователям нравится один или несколько жанров, маловероятно, что они купили одни и те же объекты.

Один из способов борьбы с этим явлением состоит в том, чтобы кластеризовать объекты и/или пользователей. Воспользуемся любой из метрик, предложенных в разделе 9.3.1, или какой-нибудь другой для кластеризации, например, объектов. Подойдет любой из методов, описанных в главе 7. Однако, как мы увидим, не имеет особого смысла пытаться сразу строить небольшое число кластеров. На первом шаге может оказаться достаточно иерархического подхода, когда мы оставляем много кластеров не объединенными. Например, можно построить в два раза меньше кластеров, чем имеется объектов.

| | HP | TW | SW |
|---|---|---|---|
| *A* | 4 | 5 | 1 |
| B | 4.67 | | |
| C | | 2 | 4.5 |
| D | 3 | | 3 |

**Рис. 9.7.** Матрица предпочтений для пользователей и кластеров объектов

**Пример 9.10.** На рис. 9.7 показано, как изменится матрица на рис. 9.4, если объединить все три фильма о Гарри Поттере в один кластер, обозначенный HP, а три фильма о звездных войнах – в кластер SW.

После ограниченной кластеризации объектов мы можем пересчитать матрицу предпочтений, так что столбцы будут представлять кластеры объектов, а в элементе на пересечении пользователя $U$ и кластера $C$ будет находиться средняя оценка, поставленная $U$ тем членам кластера $C$, которые $U$ оценивал. Отметим, что $U$ мог не оценить ни один член кластера, и тогда элемент на пересечении $U$ и $C$ так и останется пустым.

Эту пересчитанную матрицу можно использовать для кластеризации пользователей, опять-таки взяв наиболее подходящую метрику. При этом мы снова оставим много кластеров, скажем, вполовину меньше, чем имеется пользователей. Пересчитаем матрицу предпочтений, так чтобы строки соответствовали кластерам пользователей, как столбцы соответствуют кластерам объектов. Элемент матрицы для кластера пользователей получается усреднением оценок, поставленных всеми пользователями из этого кластера.

Эту процедуру при желании можно повторить несколько раз. То есть сначала кластеризуем кластеры объектов с последующим объединением столбцов матрицы предпочтений, принадлежащих одному кластеру. Затем кластеризуем кластеры пользователей. И повторяем, пока не останется согласующееся с интуицией количество кластеров каждого вида.

Построив кластеры пользователей и объектов и вычислив кластерную матрицу предпочтений, мы можем следующим образом оценить элементы исходной матрицы. Пусть требуется предсказать элемент для пользователя $U$ и объекта $I$.

(*а*) Найти кластеры, которым принадлежат $U$ и $I$; пусть это будут кластеры $C$ и $D$ соответственно.

(*б*) Если в кластерной матрице предпочтений элемент на пересечении $C$ и $D$ не пуст, использовать его значение, как оценку элемента на пересечении $U$ и $I$ в исходной матрице.

(*в*) Если элемент на пересечении $C$ и $D$ пуст, воспользоваться методом из раздела 9.3.2 для оценки этого элемента, рассмотрев кластеры, похожие на $C$ или $D$. Результат взять в качестве оценки элемента на пересечении $U$ и $I$.

## 9.3.4. Упражнения к разделу 9.3

**Упражнение 9.3.1.** На рис. 9.8 приведена матрица предпочтений, в которой представлены оценки по шкале от 1 до 5, поставленные восьми объектам от $a$ до $h$ тремя пользователями $A$, $B$ и $C$. Выполните для этой матрицы следующие действия:

(*а*) Рассматривая матрицу предпочтений как булеву, вычислите расстояние между каждой парой пользователей.

(*б*) Сделайте то же самое для косинусного расстояния.

(*в*) Замените оценки 3, 4, 5 на 1, а 1, 2 и отсутствующую – на 0. Вычислите расстояние между каждой парой пользователей.

(*г*) Сделайте то же самое для косинусного расстояния.

(*д*) Нормируйте матрицу путем вычитания из каждого непустого элемента средней оценки, поставленной соответствующим пользователем.

(*е*) Для такой нормированной матрицы вычислите косинусное расстояние между каждой парой пользователей.

**Упражнение 9.3.2.** В этом упражнении мы кластеризуем объекты, представленные в матрице на рис. 9.8. Выполните следующие действия.

(*а*) Иерархически кластеризуйте все восемь объектов в четыре кластера, применив описанный ниже метод. Заменить оценки 3, 4 и 5 на 1, а 1, 2 и отсутствующие – на 0. Вычислить расстояние между получившимися векторами-столбцами. Для кластеров, содержащих более одного элемента, в качестве расстояния между кластерами возьмите минимум расстояний между парами элементов по одному из каждого кластера.

(*б*) Затем постройте из исходной матрицы на рис. 9.8 новую матрицу, строки которой соответствуют пользователям, как и раньше, а столбцы – кластерам. Вычислите элемент для каждого пользователя и кластера объектов путем усреднения непустых элементов, соответствующих этому пользователю и всем объектам в этом кластере.

(*в*) Вычислите косинусное расстояние между каждой парой пользователей на основе матрицы, полученной в п. (*б*).

| | a | b | c | d | e | f | g | h |
|---|---|---|---|---|---|---|---|---|
| A | 4 | 5 | | 5 | 1 | | 3 | 2 |
| B | | 3 | 4 | 3 | 1 | 1 | 1 | |
| C | 2 | | 1 | 5 | | 4 | 5 | 3 |

**Рис. 9.8.** Матрица предпочтений для упражнений

# 9.4. Понижение размерности

Совершенно другой подход к оцениванию незаполненных элементов матрицы предпочтений заключается в том, чтобы представить эту матрицу в виде произведения двух матриц – длинных и узких. Это осмысленно, если имеется сравнительно немного признаков объектов и пользователей, которые определяют реакцию большинства пользователей на большинство объектов. В этом разделе мы кратко опишем один подход к нахождению матриц-сомножителей; он называется «UV-декомпозиция» и является частным случаем общей теории *сингулярного разложения* (singular-value decomposition, SVD).

## 9.4.1. UV-декомпозиция

Будем рассматривать в качестве объектов фильмы. Большинство пользователей реагируют на небольшое число признаков: им нравятся определенные жанры, определенные актеры и актрисы и, быть может, некоторые режиссеры. Если начать с матрицы предпочтений $M$ с $n$ строками (пользователями) и $m$ столбцами (объектами), то, возможно, мы сумеем найти матрицу $U$ с $n$ строками и $d$ столбцами и матрицу $V$ с $d$ строками и $m$ столбцами такие, что $UV$ хорошо аппроксимирует $M$ на непустых элементах. В таком случае мы нашли $d$ измерений, позволяющих точно охарактеризовать и пользователей, и объекты. Тогда мы сможем использовать элемент произведения $UV$ для оценки соответственного пустого элемента матрицы предпочтений $M$. Эта процедура называется *UV-декомпозицией* матрицы $M$.

**Пример 9.11.** Далее будем использовать матрицу $M$ размерности $5 \times 5$, в которой известны все элементы, кроме двух. Мы хотим разложить $M$ в произведение матрицы $U$ размерности $5 \times 2$ и матрицы $V$ размерности $2 \times 5$. Матрицы $M$, $U$ и $V$ показаны на рис. 9.9, где в матрице $M$ представлены известные элементы, а элементы матриц $U$ и $V$ – неизвестные, которые предстоит найти. Это по сути дела наименьший нетривиальный случай, когда известных элементов больше, чем элементов в $U$ и $V$ вместе взятых, поэтому можно ожидать, что наилучшая декомпозиция не даст произведения, которое точно совпадает с $M$ на всех непустых элементах.

$$\begin{bmatrix} 5 & 2 & 4 & 4 & 3 \\ 3 & 1 & 2 & 4 & 1 \\ 2 & & 3 & 1 & 4 \\ 2 & 5 & 4 & 3 & 5 \\ 4 & 4 & 5 & 4 & \end{bmatrix} = \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \\ u_{31} & u_{32} \\ u_{41} & u_{42} \\ u_{51} & u_{52} \end{bmatrix} \times \begin{bmatrix} v_{11} & v_{12} & v_{13} & v_{14} & v_{15} \\ v_{21} & v_{22} & v_{23} & v_{24} & v_{25} \end{bmatrix}$$

**Рис. 9.9.** UV-декомпозиция матрицы $M$

## 9.4.2. Среднеквадратичная ошибка

Существует несколько способов измерить близость произведения $UV$ к $M$, но обычно применяется среднеквадратичная ошибка (СКО), вычисляемая следующим образом.

1. Просуммировать по всем непустым элементам $M$ квадраты разностей между этим элементом и соответственным элементом $UV$.
2. Вычислить среднее значение, разделив эту сумму на количество слагаемых (непустых элементов $M$).
3. Извлечь квадратный корень из среднего.

Поскольку минимизация суммы квадратов – все равно, что минимизация квадратного корня из усредненного значения этой суммы, обычно мы будем опускать последние два шага.

**Пример 9.12.** Предположим, мы высказали гипотезу, что все элементы матриц *U* и *V* должны быть равны 1, как на рис. 9.10. Эта гипотеза неудачна, поскольку все элементы произведения равны 2, а это намного меньше среднего значения элементов *M*. Тем не менее, вы можем вычислить СКО для таких *U* и *V*; благодаря регулярности элементов сделать это особенно просто. Рассмотрим первые строки *M* и *UV*. После вычитания 2 (значение любого элемента *UV*) из всех элементов первой строки *M* получим 3, 0, 2, 2, 1. Сумма квадратов этих чисел равна 18. Для второй строки аналогично получаем 1, –1, 0, 2, –1 и сумму квадратов 7. В третьей строке второй элемент пуст, поэтому при вычислении СКО он игнорируется. Разности равны 0, 1, –1, 2, а сумма их квадратов – 6. В четвертой строке сумма квадратов разностей 0, 3, 2, 1, 3 равна 23. В пятой строке последний элемент пуст, поэтому разности равны 2, 2, 3, 2, а сумма квадратов – 21. Просуммировав все суммы квадратов, получаем 18 + 7 + 6 + 23 + 21 = 75. На этом можно и остановиться, но если мы захотим вычислить СКО, то должны будем разделить эту сумму на 23 (количество непустых элементов *M*) и извлечь квадратный корень. В данном случае $\sqrt{75/23} = 1.806$, это и есть СКО.

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \end{bmatrix}$$

**Рис. 9.10.** Матрицы U и V, в которых все элементы равны 1

## 9.4.3. Инкрементное вычисление UV-декомпозиции

Для нахождения UV-декомпозиции с наименьшей СКО нужно начать с произвольно выбранных *U* и *V* и постепенно корректировать их для уменьшения СКО. Мы будем рассматривать только корректировки одного элемента *U* или *V*, хотя, в принципе, возможны и более сложные преобразования. Но при любом способе корректировки в типичном случае будет много локальных минимумов – таких матриц *U* и *V*, что никакая допустимая корректировка не приводит к уменьшению СКО. К несчастью, только один из локальных минимумов является глобальным, при котором достигается наименьшее возможное значение СКО. Чтобы повысить шансы на отыскание глобального минимума, мы должны рассмотреть много разных начальных точек, т. е. начальных матриц *U* и *V*. Однако гарантии, что лучший из найденных локальных минимумов действительно окажется глобальным, все равно нет.

Начнем с матриц *U* и *V*, все элементы которых равны 1 (рис. 9.10) и выполним корректировку нескольких элементов, стремясь найти такие значения, при которых улучшение СКО максимально. Из приведенных примеров должно быть понятно, как производится вычисление в общем случае, но мы сопроводим примеры формулой для минимизации СКО путем изменения одного элемента. Ниже элементы *U* и *V* обозначаются $u_{11}$ и т. д., как показано на рис 9.9.

**Пример 9.13.** Пусть в начале выбраны матрицы $U$ и $V$, показанные на рис. 9.10, и мы решили изменить $u_{11}$, чтобы уменьшить СКО, насколько удастся. Пусть искомое значение $u_{11}$ равно $x$. Тогда новые $U$ и $V$ можно записать, как показано на рис. 9.11.

$$\begin{bmatrix} x & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} x+1 & x+1 & x+1 & x+1 & x+1 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \end{bmatrix}$$

**Рис. 9.11.** Превращение $u_{11}$ в переменную

Заметим, что изменились только элементы произведения в первой строке. Поэтому при сравнении $UV$ и $M$ вклад в изменение СКО дает только первая строка, и этот вклад равен

$$(5 - (x + 1))^2 + (2 - (x + 1))^2 + (4 - (x + 1))^2 + (4 - (x + 1))^2 + (3 - (x + 1))^2.$$

После упрощения получаем

$$(4 - x)^2 + (1 - x)^2 + (3 - x)^2 + (3 - x)^2 + (2 - x)^2.$$

Чтобы вычислить минимум этой суммы, возьмем производную и приравняем ее нулю:

$$-2 \times ((4 - x) + (1 - x) + (3 - x) + (3 - x) + (2 - x)) = 0$$

или $-2 \times (13 - 5x) = 0$, откуда $x = 2.6$.

$$\begin{bmatrix} 2.6 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 3.6 & 3.6 & 3.6 & 3.6 & 3.6 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \end{bmatrix}$$

**Рис. 9.12.** Оптимальное значение $u_{11}$ равно 2.6

На рис. 9.12 показаны $U$ и $V$ после того, как $u_{11}$ присвоено значение 2.6. Заметим, что сумма квадратов ошибок в первой строке уменьшилась с 18 до 5.2, так что полная СКО (без учета усреднения и квадратного корня) снизилась с 75 до 62.2.

$$\begin{bmatrix} 2.6 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} y & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 2.6y+1 & 3.6 & 3.6 & 3.6 & 3.6 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \end{bmatrix}$$

**Рис. 9.13.** Полагаем $v_{11}$ равным $y$

Следующим будем изменять элемент $v_{11}$. Обозначим его искомое значение $y$, как на рис. 9.13. От $y$ зависит только первый столбец, поэтому нам нужно вычислить сумму квадратов разностей между соответственными элементами первого столбца $M$ и $UV$. Эта сумма равна

$$(5 - (2.6y + 1))^2 + (3 - (y + 1))^2 + (2 - (y + 1))^2 + (2 - (y + 1))^2 + (4 - (y + 1))^2.$$

После упрощения получаем

$$(4 - 2.6y)^2 + (2 - y)^2 + (1 - y)^2 + (1 - y)^2 + (3 - y)^2.$$

Как и раньше, находим минимальное значение этого выражения, приравнивая производную нулю:

$$-2 \times (2.6(4 - 2.6y) + (2 - y) + (1 - y) + (1 - y) + (3 - y)) = 0.$$

$$\begin{bmatrix} 2.6 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1.617 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 5.204 & 3.6 & 3.6 & 3.6 & 3.6 \\ 2.617 & 2 & 2 & 2 & 2 \\ 2.617 & 2 & 2 & 2 & 2 \\ 2.617 & 2 & 2 & 2 & 2 \\ 2.617 & 2 & 2 & 2 & 2 \end{bmatrix}$$

**Рис. 9.14.** Подставляем $y = 1.617$

Из этого уравнения находим $y = 17.4/10.76 = 1.617$. На рис. 9.14 показаны улучшенные оценки $U$ и $V$.

Произведем еще одно изменение, чтобы показать, что происходит, когда элементы $M$ пусты. Изменим элемент $u_{31}$, временно обозначив его $z$. Новые $U$ и $V$ показаны на рис. 9.15. Значение $z$ влияет только на элементы в третьей строке.

$$\begin{bmatrix} 2.6 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1.617 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 5.204 & 3.6 & 3.6 & 3.6 & 3.6 \\ 2.617 & 2 & 2 & 2 & 2 \\ 1.617z+1 & z+1 & z+1 & z+1 & z+1 \\ 2.617 & 2 & 2 & 2 & 2 \\ 2.617 & 2 & 2 & 2 & 2 \end{bmatrix}$$

**Рис. 9.15.** Полагаем $u_{31}$ равным $z$

Сумму квадратов ошибок можно записать в виде

$$(2 - (1.617z + 1))^2 + (3 - (z + 1))^2 + (1 - (z + 1))^2 + (4 - (z + 1))^2.$$

Заметим, что элемент на пересечении второго столбца и третьей строки не дает никакого вклада, потому что в матрице $M$ он пуст. После упрощения получаем

$$(1 - 1.617z)^2 + (2 - z)^2 + (-z)^2 + (3 - z)^2.$$

Приравнивание производной нулю дает уравнение

$$-2 \times (1.617(1 - 1.617z) + (2 - z) + (-z) + (3 - z)) = 0.$$

из которого находим $z = 6.617/5.615 = 1.178$. Очередная оценка разложения в произведение $UV$ показана на рис. 9.16.

$$\begin{bmatrix} 2.6 & 1 \\ 1 & 1 \\ 1.178 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1.617 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 5.204 & 3.6 & 3.6 & 3.6 & 3.6 \\ 2.617 & 2 & 2 & 2 & 2 \\ 2.905 & 2.178 & 2.178 & 2.178 & 2.178 \\ 2.617 & 2 & 2 & 2 & 2 \\ 2.617 & 2 & 2 & 2 & 2 \end{bmatrix}$$

**Рис. 9.16.** Подставляем $z = 1.178$

## 9.4.4. Оптимизация произвольного элемента

Познакомившись с примерами выбора оптимального значения одного элемента матрицы $U$ или $V$, выведем общую формулу. Как и раньше, будем предполагать, что $M$ – матрица предпочтений размерности $n \times m$, в которой некоторые элементы пусты, а $U$ и $V$ – матрицы размерности $n \times d$ и $d \times m$ для некоторого $d$. Будем обозначать $m_{ij}$, $u_{ij}$ и $v_{ij}$ элементы на пересечении строки $i$ и столбца $j$ матриц $M$, $U$, $V$ соответственно. Кроме того, обозначим $P = UV$, а $p_{ij}$ – элемент на пересечении строки $i$ и столбца $j$ произведения $P$.

Пусть мы хотим варьировать $u_{rs}$ и найти такое значение этого элемента, при котором достигается минимум СКО между $M$ и $UV$. Заметим, что от $u_{rs}$ зависят только элементы строки $r$ произведения $P = UV$. Поэтому нас интересуют только элементы

$$p_{rj} = \sum_{k=1}^{d} u_{rk} v_{kj} = \sum_{k \neq s} u_{rk} v_{kj} + x v_{sj}$$

для всех таких $j$, что элемент $m_{rj}$ не пуст. В этом выражении мы заменили варьируемый элемент $u_{rs}$ переменной $x$ и воспользовались таким обозначением:

- $\sum_{k \neq s}$ – сокращенная запись суммирования по всем $k = 1, 2, \ldots, d$, кроме $k = s$.

Вклад непустого элемента $m_{rj}$ матрицы $M$ в сумму квадратов ошибок равен

$$(m_{rj} - p_{rj})^2 = (m_{rj} - \sum_{k \neq s} u_{rk} v_{kj} - x v_{sj})^2$$

Введем еще одно обозначение:

- $\sum_{j}$ – сокращенная запись суммирования по всем $j$, для которых $m_{rj}$ не пуст.

Тогда сумму квадратов ошибок, зависящую от $x = u_{rs}$, можно записать в виде

$$\sum_{j} (m_{rj} - \sum_{k \neq s} u_{rk} v_{kj} - x v_{sj})^2$$

Продифференцируем это выражение по $x$ и приравняем производную 0, чтобы найти значение $x$, доставляющее минимум СКО.

$$\sum_j -2v_{sj}(m_{rj} - \sum_{k \neq s} u_{rk}v_{kj} - xv_{sj}) = 0$$

Как и в предыдущих примерах, общий множитель –2 можно сократить. Решая это уравнение относительно $x$, находим

$$x = \frac{\sum_j v_{sj}(m_{rj} - \sum_{k \neq s} u_{rk}v_{kj})}{\sum_j v_{sj}^2}$$

Аналогичная формула существует для вычисления оптимального значения элемента $V$. Если мы варьируем $v_{rs} = y$, то значение $y$, доставляющее минимум СКО, равно

$$y = \frac{\sum_i u_{ir}(m_{is} - \sum_{k \neq r} u_{ik}v_{ks})}{\sum_i u_{ir}^2}$$

Здесь $\sum_i$ – сокращенная запись суммирования по всем $i$, для которых элемент $m_{is}$ не пуст, а $\sum_{k \neq r}$ – сумма по всем $k = 1, 2, \ldots, d$, кроме $k = r$.

## 9.4.5. Построение полного алгоритма UV-декомпозиции

Теперь у нас есть все инструменты для поиска глобально оптимальной декомпозиции матрицы предпочтений $M$. Есть четыре аспекта алгоритма, и для каждого из них мы обсудим различные варианты.

1. Предварительная обработка матрицы $M$.
2. Инициализация $U$ и $V$.
3. Порядок оптимизации элементов $U$ и $V$.
4. Прекращение попыток оптимизации.

### Предварительная обработка

Поскольку различия в качестве объектов и шкал пользовательских оценок очень важны для определения отсутствующих элементов матрицы $M$, часто бывает полезно прежде всего устранить влияние этих факторов. Идея была изложена в разделе 9.3.1. Мы можем вычесть из каждого непустого элемента $m_{ij}$ среднюю оценку пользователя $i$. Затем получившуюся матрицу можно модифицировать, вычтя среднюю оценку (после первой модификации) объекта $j$. Можно поступить наоборот – сначала вычесть среднюю оценку объекта $j$, а затем – среднюю оценку пользователя $i$. Результаты будут не одинаковы, но близки. Третий вариант нормировки – вычесть из $m_{ij}$ среднее из двух средних оценок: пользователя $i$ и объекта $j$, т. е. полусумму среднего по пользователям и среднего по объектам.

Если мы решили предварительно нормировать $M$, то в процессе предсказания значений необходимо выполнить обратную операцию. То есть если применяемый метод предсказал значение $e$ для элемента $m_{ij}$ нормированной матрицы, то в качестве значения соответственного элемента истинной матрицы предпочтений мы должны взять $e$ плюс то значение, которое было вычтено из элемента на пересечении строки $i$ и столбца $j$ в процессе нормировки.

## Инициализация

Как уже было сказано, в процесс поиска оптимального решения необходимо внести элемент случайности, поскольку из-за существования многих локальных минимумов мы пытаемся выполнить оптимизацию много раз с различными начальными данными в надежде хотя бы в одном прогоне найти глобальный минимум. Мы можем варьировать начальные $U$ и $V$ или способ поиска оптимума (обсуждается ниже) или то и другое.

Простое начальное приближение для $U$ и $V$ – матрицы, все элементы которых равны одному и тому же значению, и в качестве этого значения имеет смысл взять такое, что каждый элемент произведения $UV$ равен среднему всех непустых элементов $M$. Отметим, что если мы предварительно нормировали матрицу, то это среднее обязательно равно 0. Если $d$ – длина коротких сторон матриц $U$ и $V$, а $a$ – среднее непустых элементов $M$, то каждый элемент $U$ и $V$ должен быть равен $\sqrt{a/d}$.

Если мы хотим выбрать много начальных матриц $U$ и $V$, то можем немного пошевелить каждый элемент $\sqrt{a/d}$, так чтобы изменения были случайными и независимыми. Такое возмущение можно внести разными способами. Мы можем выбирать распределение вероятностей разности. Например, можно прибавить к каждому элементу случайную величину, имеющую нормальное распределение со средним 0 и заданным стандартным отклонением. Или прибавить случайную величину, равномерно распределенную в диапазоне от $-c$ до $+c$ при некотором $c$.

## Выполнение оптимизации

Чтобы выйти на локальный минимум, начав с заданных $U$ и $V$, мы должны определить, в каком порядке будем посещать элементы $U$ и $V$. Проще всего выбрать какой-то порядок элементов $U$ и $V$, например по строкам, и посещать их поочередно. Отметим, что из того, что мы один раз оптимизировали некий элемент, вовсе не следует, что мы не сможем найти лучшее значение после того, как будут скорректированы все остальные элементы. Поэтому мы должны повторять цикл посещения элементов, пока на основании каких-то фактов не решим, что дальнейшее улучшение невозможно. Есть и другая возможность: оставить единственное начальное значение, но менять путь оптимизации, случайным образом выбирая оптимизируемый элемент. Для гарантии того, что на каждом раунде будут рассмотрены все элементы, мы можем в начале раунда зафиксировать перестановку элементов и посещать их в порядке этой перестановки.

## Сходимость к минимуму

В идеале в какой-то момент СКО обратится в 0, и тогда улучшение точно невозможно. На практике, поскольку в $M$ обычно гораздо больше непустых элементов, чем всего элементов в $U$ и $V$ вместе взятых, нет никаких оснований ожидать, что нам удастся уменьшить СКО до 0. Поэтому нужно каким-то образом устанавливать, что очередное посещение элементов $U$ и (или) $V$ не принесет заметного выигрыша. Мы можем отслеживать улучшение СКО, полученное после одного раунда оптимизации, и останавливаться, когда улучшение окажется меньше заданного порога. Вариация на ту же тему – отслеживать улучшение отдельных элементов в результате оптимизации и останавливаться, когда максимальное улучшение после раунда оказалось меньше порога.

### Градиентный спуск

Метод нахождения UV-декомпозиции, рассмотренный в разделе 9.4, является примером *градиентного спуска*. Даны некоторые точки – непустые элементы матрицы $M$ – и для каждой точки мы ищем, изменение в каком направлении приводит к наибольшему уменьшению функции ошибок: СКО между текущим произведением $UV$ и матрицей $M$. Мы еще вернемся к градиентному спуску в разделе 12.3.4. Стоит также отметить, что в нашей интерпретации этот метод сводится к многократному посещению всех непустых элементов $M$, пока не будет найдена декомпозиция с минимальной ошибкой. Но для большой матрицы объем работы может оказаться слишком большим. Поэтому в процессе минимизации ошибки можно вместо этого просматривать лишь случайную выборку данных. Этот подход, именуемый *стохастическим градиентным спуском*, обсуждается в разделе 12.3.5.

## Предотвращение переобучения

При выполнении UV-декомпозиции часто возникает следующая проблема: мы находим один из многих локальных минимумов, хорошо соответствующий данным, но полученная при этом аппроксимация не отражает истинный процесс, приведший к возникновению исходных данных. Это означает, что хотя СКО мала, найденная декомпозиция не пригодна для предсказания будущих данных. Справиться с этой проблемой, которую статистики называют *переобучением*, можно несколькими способами.

1. Не отдавать предпочтения первым оптимизируемым элементам, для чего изменять элемент не на всю разность между текущим и оптимизированным значением, а, скажем, только на половину этой разности.
2. Прекращать повторные посещения элементов $U$ и $V$ задолго до того, как процесс сойдется.

3. Взять несколько разных UV-декомпозиций и, когда требуется предсказать значение какого-то элемента матрицы $M$, вычислять среднее результатов, взятых из каждой декомпозиции.

## 9.4.6. Упражнения к разделу 9.4

**Упражнение 9.4.1.** Начав с декомпозиции на рис. 9.10, мы можем в качестве первого оптимизируемого элемента выбрать любой из двадцати элементов $U$ или $V$. Выполните оптимизацию, когда первым берется элемент (*а*) $u_{32}$ (*б*) $v_{41}$.

**Упражнение 9.4.2.** Если мы захотим начать с $U$ и $V$, состоящих из одинаковых элементов, как на рис. 9.10, то при каком значении будет достигаться минимум СКО для нашей матрицы $M$?

**Упражнение 9.4.3.** Начав с матриц $U$ и $V$, показанных на рис. 9.16, выполните следующие действия в указанном порядке:

(*а*) Заново рассмотрите значение $u_{11}$. Найдите новое наилучшее значение после уже произведенных изменений.
(*б*) Затем найдите наилучшее значение $u_{52}$.
(*в*) Затем найдите наилучшее значение $v_{22}$.

**Упражнение 9.4.4.** Выведите формулу для $y$ (оптимальное значение элемента $v_{rs}$ в обозначениях конца раздела 9.4.4).

**Упражнение 9.4.5.** Нормируйте матрицу $M$ из нашего сквозного примера следующими способами:

(*а*) Сначала вычесть из каждого элемента среднее по строке, затем вычесть из каждого элемента среднее по (модифицированному) столбцу.
(*б*) Сначала вычесть из каждого элемента среднее по столбцу, затем вычесть из каждого элемента среднее по (модифицированной) строке.

Есть ли различия между результатами операций (*а*) и (*б*)?

# 9.5. Задача NetFlix

Значительным стимулом для исследования рекомендательных систем стало обещание компании NetFlix выплатить приз в 1 миллион долларов первому человеку или коллективу, который сумеет улучшить ее собственный алгоритм рекомендования под названием CineMatch на 10 %. После трех лет работы приз был вручен в сентябре 2009 года.

Данными в задаче NetFlix был опубликованный набор данных с оценками, проставленными примерно полумиллионом пользователей приблизительно 17 000 фильмам (типичный пользователь оценивал лишь малое подмножество всех фильмов). Эти данные представляли собой выборку из большего набора данных, а для предложенных алгоритмов тестировалась способность предсказать оставшуюся часть набора, которая держалась в секрете. В состав информации о каждой паре (пользователь, фильм) входила оценка (от 1 до 5 звезд) и дата ее выставления.

Для измерения качества алгоритмов использовалась среднеквадратичная ошибка. У алгоритма CineMatch СКО была равна приблизительно 0.95, т. е. типичное отклонение от истинной оценки составляло почту целую звезду. Чтобы получить приз, было необходимо предложить алгоритм, для которого СКО было не больше 90 % СКО CineMatch.

В списке литературы к данной главе есть ссылки на описания победивших алгоритмов. Здесь мы упомянем лишь некоторые интересные и, пожалуй, противоречащие интуиции факты, касающиеся этой задачи.

- Алгоритм CineMatch был не особенно хорош. На самом деле, довольно быстро обнаружилось, что очевидный алгоритм предсказания оценки, которую пользователь $u$ поставил бы фильму $m$, как среднего двух чисел:
  1. Средней оценки, поставленной $u$ всем оцененным им фильмам, и
  2. Средней оценки, поставленной $m$ всеми пользователями, оценившими этот фильм

  был лишь на 3 % хуже CineMatch.
- Как показали три студента (Майкл Харрис, Джеффри Уонг и Дэвид Камм), алгоритм UV-декомпозиции, описанный в разделе 9.4, в сочетании с нормировкой и некоторыми другими хитростями давал семипроцентное улучшение CineMatch.
- Победивший алгоритм представлял собой комбинацию нескольких различных алгоритмов, разработанных независимо. Другая группа, которая победила бы, если бы представила свое решение на несколько минут раньше, также применила комбинацию независимых алгоритмов. Эта стратегия – комбинирование различных алгоритмов – и раньше применялась для решения ряда трудных задач, так что ее стоит запомнить.
- Было предпринято несколько попыток использовать данные, хранящиеся в базе данных фильмов в Интернете (IMDB), с целью сопоставить названия фильмов из задачи NetFlix с названиями в IMDB и таким образом извлечь полезную информацию, отсутствующую в данных NetFlix. В IMDB хранятся сведения об актерах и режиссерах, а сами фильмы классифицированы по 28 жанрам. Но выяснилось, что жанровая и прочая информация ничем не помогает. Одна из возможных причин заключается в том, что алгоритмы машинного обучения и так смогли выделить нужную информацию, а другая – в том, что задачу отождествления фильмов в NetFlix и IMDB по названиям трудно решить точно.
- Время выставления оценки оказалось полезным. Оказалось, что вероятность высокой оценки больше, когда человек оценивает фильм сразу после просмотра, чем по прошествии времени. Примером может служить фильм «Целитель Адамс». И наоборот, есть фильмы, которые не понравились тем, кто оценивал их сразу, но получали более высокую оценку, если оценивались спустя некоторое время; в качестве примера приводили фильм «Помни». Хотя из данных нельзя выяснить, сколько времени прошло между

просмотром и оцениванием, можно с большой долей уверенности предположить, что большинство смотрит фильм вскоре после его выхода на экран. Поэтому можно проанализировать оценки фильма на предмет того, повышаются они с течением времени или снижаются.

# 9.6. Резюме

- *Матрицы предпочтений.* Рекомендательные системы имеют дело с пользователями и объектами. В матрице предпочтений хранится известная информация о том, насколько объект нравится пользователю. Обычно большинство ее элементов неизвестны, а задача состоит в том, чтобы рекомендовать объекты пользователям, предсказав значения неизвестных элементов на основе известных.
- *Два класса рекомендательных систем.* Такие системы пытаются предсказать реакцию пользователя на объект, определив, как пользователь реагировал на похожие объекты. К одному классу относятся рекомендательные системы на основе фильтрации содержимого: они измеряют сходство объектов путем поиска общих признаков. К другому классу относятся системы коллаборативной фильтрации: они измеряют сходство пользователей по тому, какие объекты они предпочитают, или сходство объектов по тому, каким пользователям они нравятся.
- *Профили объектов.* Профиль включает признаки объектов. У объектов разных видов имеются различные признаки, на основе которых можно сделать вывод о сходстве. Признаками документов обычно являются важные или необычные слова. У изделий есть такие атрибуты, как размер экрана телевизора. Признаками медийных материалов, например фильмов, могут служить жанр и актер или исполнитель. В качестве признаков можно использовать также метки, если найдутся заинтересованные пользователи, готовые их проставлять.
- *Профили пользователей.* Система коллаборативной фильтрации может строить профили пользователей, измеряя частоты встречаемости признаков в объектах, нравящихся пользователям. Затем можно оценить, в какой мере пользователю понравится объект, вычислив близость профиля объекта и профиля пользователя.
- *Классификация объектов.* Альтернативой построению профилей пользователей является построение классификатора для каждого пользователя, например в виде решающего дерева. Обучающими данными при этом становится строка матрицы предпочтений, относящаяся к этому пользователю, а задача классификатора – предсказать реакцию пользователя на все объекты, в том числе и не встречавшиеся в строке.
- *Сходство строк и столбцов матрицы предпочтений.* Алгоритмы коллаборативной фильтрации должны измерять сходство строк и (или) столбцов

матрицы предпочтений. Расстояние Жаккара годится для этой цели, если матрица содержит только единицы и незаполненные элементы («не оценено»). Для матриц более общего вида применяется косинусное расстояние. Часто перед тем как вычислять косинусное расстояние, бывает полезно нормировать матрицу предпочтений путем вычитания среднего значения (по строке, по столбцу или комбинацию того и другого) из каждого элемента.

- *Кластеризация пользователей и объектов*. Поскольку большинство элементов матрицы предпочтений обычно не заполнено, данных для сравнения строк или столбцов с помощью расстояния Жаккара или косинусного расстояния слишком мало. В этом случае может помочь выполнение одного или нескольких предварительных шагов, на которых сходство используется для кластеризации пользователей и (или) объектов в небольшие группы. Впоследствии для сравнения строк и столбцов берутся именно эти группы похожих элементов.
- *UV-декомпозиция*. Один из способов предсказания отсутствующих значений в матрице предпочтений заключается в том, чтобы найти две длинных и узких матрицы *U* и *V*, произведение которых аппроксимирует заданную матрицу. Поскольку в произведении *UV* заполнены все элементы, мы можем использовать ее для предсказания значения элемента матрицы предпочтений, соответствующего любой паре пользователь-объект. Этот метод имеет смысл, потому что существует относительно немного характеристик (длина «узкой» стороны матриц *U* и *V*), определяющих, нравится объект пользователю или нет.
- *Среднеквадратичная ошибка*. Неплохой мерой близости произведения *UV* к заданной матрице предпочтений является СКО (среднеквадратичная ошибка). Для вычисления СКО сумма квадратов разностей элементов *UV* и матрицы предпочтений для всех непустых элементов последней делится на число таких элементов. Квадратный корень из этой величины и есть СКО.
- *Вычисление U и V*. Один из способов найти подходящие матрицы *U* и *V* для *UV*-декомпозиции заключается в том, чтобы начать с произвольных матриц. Затем на каждом шаге корректируется один из элементов *U* или *V*, так чтобы минимизировать СКО между произведением *UV* и заданной матрицей предпочтений. Этот процесс сходится к локальному минимуму, а чтобы с высокой вероятностью найти глобальный минимум, мы должны либо повторить его для многих начальных матриц, либо производить поиск при одних и тех же начальных матрицах многими различными способами.
- *Задача NetFlix*. Важным стимулом для исследований в области рекомендательных систем стала задача, поставленная компанией NetFlix. Претендент, предложивший алгоритм, хотя бы на 10 % лучший собственного алгоритма NetFlix для предсказания оценок фильмов, поставленных пользователями, получал миллион долларов. Приз был вручен в сентябре 2009 года.

# 9.7. Список литературы

В статье [1] приведен обзор рекомендательных систем по состоянию на 2005 год. Аргументация в пользу важности длинных хвостов в онлайновых системах, взята из работы [2], на основе которой впоследствии была написана книга [3].

В работе [8] обсуждается использование компьютерных игр для разметки объектов.

В работе [5] содержится обсуждение сходства объектов и структуры алгоритма коллаборативной фильтрации, применяемого Amazon для выработки рекомендаций товаров.

В трех статьях – [4], [6] и [7] – описываются алгоритмы, комбинация которых стала победителем в конкурсе, объявленном NetFlix.

1. G. Adomavicius, A. Tuzhilin «Towards the next generation of recommender systems: a survey of the state-of-the-art and possible extensions», IEEE Trans. on Data and Knowledge Engineering 17:6, pp. 734–749, 2005.
2. C. Anderson `http://www.wired.com/wired/archive/12.10/tail.html`
3. C. Anderson «The Long Tail: Why the Future of Business is Selling Less of More», Hyperion Books, New York, 2006.
4. Y. Koren «The BellKor solution to the Netflix grand prize», `www.netflixprize.com/assets/GrandPrize2009 BPC BellKor.pdf`, 2009.
5. G. Linden, B. Smith, J. York «Amazon.com recommendations: item-to-item collaborative filtering», Internet Computing 7:1, pp. 76–80, 2003.
6. M. Piotte, M. Chabbert «The Pragmatic Theory solution to the Netflix grand prize» `www.netflixprize.com/assets/GrandPrize2009 BPC PragmaticTheory.pdf`, 2009.
7. A. Toscher, M. Jahrer, R. Bell «The BigChaos solution to the Netflix grand prize» `www.netflixprize.com/assets/GrandPrize2009 BPC BigChaos.pdf`, 2009.
8. L. von Ahn «Games with a purpose», IEEE Computer Magazine, pp. 96–98, June 2006.

# ГЛАВА 10. Анализ графов социальных сетей

Много информации можно собрать, анализируя данные, полученные из социальных сетей. Самый известный пример социальной сети – отношение «быть другом» на сайтах типа Facebook. Однако, как мы увидим, есть еще немало источников данных, связывающих людей или другие сущности.

В этой главе мы будем изучать методы анализа таких сетей. Один из важных вопросов применительно к социальной сети – как найти «сообщества», т. е. подмножества вершин (представляющих людей или иные сущности) с необычно большим числом связей. Некоторые методы поиска сообществ похожи на алгоритмы кластеризации, которые обсуждались в главе 7. Однако сообщества почти никогда не разбивают вершины графа сети на непересекающиеся множества, а обычно перекрываются. Например, вы сами можете быть членом нескольких сообществ друзей или одноклассников. Люди из одного сообщества, как правило, знают друг друга, тогда как члены разных сообществ обычно незнакомы. Вряд ли вы согласитесь быть членом только одного сообщества, и вряд ли имеет смысл группировать всех людей из всех сообществ, которым вы принадлежите, в один кластер.

Также в этой главе мы изучим эффективные алгоритмы выявления других свойств графов. Мы познакомимся с индексом SimRank, описывающим сходство вершин графа и исследуем метод подсчета треугольников как способ измерения связности сообщества. Мы приведем эффективные алгоритмы точного и приближенного вычисления размера окрестностей вершин графа. Наконец, мы расскажем об эффективных алгоритмах вычисления транзитивного замыкания.

## 10.1. Социальные сети как графы

Начнем обсуждение социальных сетей с описания графовой модели. Не каждый граф подходит для представления того, что мы интуитивно называем социальной сетью. Поэтому мы обсудим понятие «локальности» – свойство социальной сети, означающее, что вершины и ребра графа имеют тенденцию образовывать сообщества. Здесь же мы рассмотрим некоторые виды социальных сетей, встречающиеся на практике.

## 10.1.1. Что такое социальная сеть?

Говоря о социальной сети, мы подразумеваем Facebook, Twitter, Google+ или еще какой-нибудь сайт, который называют «социальной сетью», и действительно это представители более широкого класса сетей, называемых «социальными». Перечислим отличительные особенности социальной сети.

1. Существует множество сущностей, составляющих сеть. Как правило, сущностями являются люди, но вообще это совершенно необязательно. Примеры другого рода мы обсудим в разделе 10.1.3.
2. Существует по меньшей мере одна связь между сущностями сети. На сайте Facebook и ему подобных такая связь называется *друзьями*. Иногда связь бинарна – она либо есть, либо ее нет: люди либо являются друзьями, либо не являются. Но в других социальных сетях у связи может быть степень. Степень может быть дискретной, например: друзья, семья, знакомые или отсутствует – как в Google+. Она может быть и вещественным числом, например, часть среднего дня, которую два человека тратят на разговоры друг с другом.
3. Имеет место гипотеза о неслучайности или локальности. Это условие труднее всего формализовать, но интуитивно она означает, что связи образуют кластеры. То есть, если сущность $A$ связана с $B$ и $C$, то вероятность того, что $B$ и $C$ связаны, выше средней.

## 10.1.2. Социальные сети как графы

Социальные сети естественно моделируются в виде графов, которые иногда называют *социальными графами*. Сущностям соответствуют вершины, и две вершины соединены ребром, если между ними есть связь, характеризующая сеть. Степень, ассоциированная со связью, моделируется в виде метки ребра. Часто социальные графы не ориентированы, как, например, граф друзей в Facebook. Но могут быть и ориентированными, как графы читателей (follower) в Twitter или Google+.

**Пример 10.1.** На рис. 10.1 приведен пример крохотной социальной сети. Сущностями являются вершины от $A$ до $G$. Связь, которую можно условно назвать «друзья», представлена ребрами. Например, $B$ дружит с $A$, $C$ и $D$.

Является ли этот граф типичной социальной сетью, т. е. присутствует ли в нем локальность связей? Прежде всего, отметим, что в графе имеется девять ребер из $\binom{7}{2} = 21$ возможных. Предположим, что $X$, $Y$, $Z$ – вершины графа на рис. 10.1 такие, что существуют ребра, соединяющие $X$ и $Y$, а также $X$ и $Z$. Что можно сказать о вероятности существования ребра между $Y$ и $Z$? Если бы граф был большим, то эта вероятность была бы очень близка к доле пар вершин, соединенных ребрами, в нашем случае 9/21 = 0.429. Но поскольку граф маленький, то велико различие между истинной вероятностью и отношением числа ребер к числу пар вершин. Поскольку мы уже знаем, что ребра $(X,Y)$ и $(X,Z)$ существуют, остается всего семь ребер. Эти семь ребер

могли бы соединить любые из 19 оставшихся пар вершин. Поэтому вероятность существования ребра (*Y*,*Z*) равна 7/19 = .368.

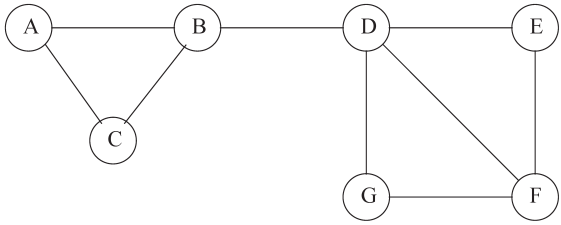



**Рис. 10.1.** Пример небольшой социальной сети

Теперь нужно вычислить вероятность существования ребра (*Y*,*Z*) при условии, что ребра (*X*,*Y*) и (*X*,*Z*) существуют. На самом деле нам нужно подсчитать пары вершин, которые могли бы совпадать с *Y* и *Z*, неважно в каком порядке. Если *X* – это *A*, то *Y* и *Z* должны совпадать с *B* и *C* в каком-то порядке. Поскольку ребро (*B*,*C*) существует, значит, *A* дает один положительный пример (когда ребро существует) и ни одного отрицательного (когда ребро отсутствует). Случаи, когда *X* – одна из вершин *C*, *E* или *G*, ничем не отличаются. В каждом случае у *X* только два соседа, между которыми существует ребро. Таким образом, пока мы видели четыре положительных примера и ни одного отрицательного.

Теперь рассмотрим случай *X* = *F*. У вершины *F* три соседа – *D*, *E* и *G*. Между двумя из трех пар соседей имеются ребра, но между *G* и *E* ребра нет. Следовательно, мы обнаружили еще два положительных примера и наконец-то один отрицательный. Если *X* = *B*, то соседей снова три, но только два из них, *A* и *C*, соединены ребром. Стало быть, мы имеем еще два отрицательных примера и один положительный, а всего получилось семь положительных и три отрицательных. Наконец, если *X* = *D*, то соседа четыре. И лишь две из шести пар соединены ребрами.

Таким образом, всего в этом графе девять положительных примеров и семь отрицательных. Следовательно, третье ребро существует в 9 случаях из 16, т. е. вероятность равна 9/16 = 0.563. Это значительно больше, чем ожидаемое значение 0.368, а, значит, граф на рис. 10.1 действительно обладает свойством локальности, требуемым от социальной сети.

## *10.1.3. Разновидности социальных сетей*

Есть немало примеров социальных сетей, отличных от сети «друзей». Упомянем несколько сетей, также обладающих свойством локальности связей.

### Телефонные сети

Здесь вершинами являются телефонные номера, по существу представляющие собой отдельных людей. Две вершины соединены ребром, если в течение некоторого промежутка времени, скажем месяца или «не определенного», между этими

номерами имел место хотя бы один вызов. Ребру можно приписать вес – количество вызовов за период. Сообщества в телефонной сети образуют группы часто общающихся людей, например: друзей, членов одного клуба или работников одной компании.

### Почтовые сети

Вершинами являются адреса электронной почты, которые также соответствуют отдельным людям. Наличие ребра означает, что с одного адреса на другой было отправлено хотя бы одно письмо. Или можно соединять вершины ребром, если имеется хотя бы по одному письму в обоих направлениях. В таком случае мы не будем рассматривать спамера как «друга» своей жертвы. Другой подход – помечать ребра как *слабые* или *сильные*. Сильное ребро представляет коммуникацию в обоих направлениях, слабое – только в одном. Сообщества в почтовой сети образуются так же, как в телефонной. Похожую сеть составляют люди, посылающие другу другу текстовые сообщения по мобильному телефону.

### Созидательная сеть

Вершинами являются люди, публикующие научные статьи. Между двумя вершинами существует ребро, если соответствующие люди опубликовали одну или несколько статей совместно. Можно также пометить ребра количеством совместных публикаций. Сообщества в такой сети образованы авторами, работающими в одной узкой области.

Альтернативным представлением тех же данных является граф, в котором вершинами являются статьи. Две статьи соединены ребром, если у них есть хотя бы один общий автор. Теперь сообщества образованы статьями на одну тему.

Существуют и другие виды данных, образующие подобные двойственные сети. Например, можно рассмотреть людей, редактирующих статьи в википедии, и статьи, которые они редактируют. Два редактора соединены ребром, если они совместно редактировали одну статью. Сообществами являются группы редакторов, интересующихся одним и тем же предметом. Наоборот, можно построить сеть статей, соединив две статьи ребром, если они редактировались одним человеком. В этом случае сообществами будут статьи на схожие темы.

На самом деле, часто можно считать, что данные для коллаборативной фильтрации, обсуждавшейся в главе 9, образуют две сети: покупателей и товаров. Покупатели, приобретающие товары одного типа, например научно-фантастические книги, образуют сообщества и, наоборот, товары, приобретаемые одними и теми же покупателями, также образуют сообщества, например все научно-фантастические книги.

### Другие примеры социальных графов

Многие явления порождают графы, похожие на социальные сети, особенно в части локальности, например: информационные сети (документы, веб-графы, патенты), инфраструктурные сети (дороги, авиамаршруты, водопроводы, элек-

тросети), биологические сети (гены, белки, пищевые цепочки животных, питающихся друг другом) и другие, в т. ч. сети совместной покупки товаров (например, Groupon).

## 10.1.4. Графы с вершинами нескольких типов

Существует еще одно социальное явление, в котором участвуют сущности разных типов. Выше, в подразделе «Созидательные сети» мы рассмотрели несколько видов графов, образованных вершинами двух видов. В сетях авторства встречаются вершины-авторы и вершины-статьи. Строя две социальные сети, мы исключали вершины одного типа, но вообще-то нас к этому никто не принуждает. Можно размышлять и о структуре в целом.

Вот более сложный пример: пользователи сайта типа deli.cio.us ассоциируют метки с веб-страницами. То есть налицо три типа сущностей: пользователи, метки и страницы. Можно считать, что пользователи как-то связаны, если часто используют одни и те же метки или если помечают одни и те же страницы. Аналогично метки можно считать родственными, если они ассоциированы с одними и теми же страницами или поставлены одними и теми же пользователями, а страницы считать похожими, если у них много общих одинаковых меток или они помечались одними и теми же пользователями.

Естественный способ представления такой информации дают $k$-дольные графы при некотором $k > 1$. В разделе 8.3 нам уже встречались двудольные графы (случай $k = 2$). В общем случае $k$-дольный граф состоит из $k$ непересекающихся множеств вершин таких, что никакие две вершины, принадлежащие одному множеству, не соединены ребром.

**Пример 10.2.** На рис. 10.2 изображен трехдольный граф. Мы видим три множества вершин, которые можно интерпретировать как пользователей $\{U_1, U_2\}$, метки $\{T_1, T_2, T_3, T_4\}$ и веб-страницы $\{W_1, W_2, W_3\}$. Обратите внимание, что ребрами соединены только вершины, принадлежащие разным множествам. Можно считать, что этот граф представляет информацию о трех видах сущностей. Например, наличие ребра $(U_1, T_2)$ означает, что пользователь $U_1$ пометил меткой $T_2$ по меньшей мере одну страницу. Заметим, что граф ничего не говорит о детали, которая может оказаться важной: кто пометил какой меткой какую страницу? Для включения такой информации понадобилась бы более сложная структура, например отношение в базе данных с тремя столбцами, соответствующими пользователям, меткам и страницам.

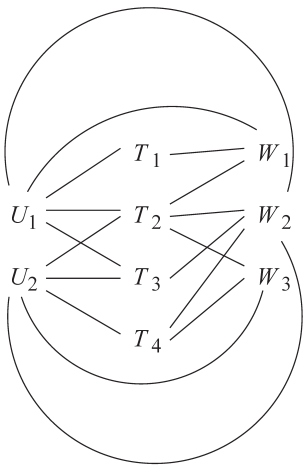



**Рис. 10.2.** Трехдольный граф, представляющий пользователей, метки и веб-страницы

### 10.1.5. Упражнения к разделу 10.1

**Упражнение 10.1.1.** Ребра одного графа $G$ можно представлять себе как вершины другого $G'$. *Двойственный граф* $G'$ строится из $G$ следующим образом:

1. Если $(X, Y)$ – ребро $G$, то символ $XY$, представляющий неупорядоченное множество элементов $X$ и $Y$, является вершиной $G'$. Заметим, что $XY$ и $YX$ представляют одну и ту же, а не разные вершины $G'$.
2. Если $(X, Y)$ и $(X, Z)$ – ребра $G$, то в $G'$ существует ребро между $XY$ и $XZ$. Иначе говоря, вершины $G'$ соединены ребром, если ребра $G$, представленные этими вершинами, имеют общую вершину.

(*а*) Если применить описанную процедуру к графу друзей, то как можно интерпретировать ребра двойственного графа?

(*б*) Постройте двойственный граф для графа на рис. 10.1.

! (*в*) Как степень вершины $XY$ в $G'$ соотносится со степенями вершин $X$ и $Y$ в $G$?

!! (*г*) Существует формула, связывающая число ребер $G'$ со степенями вершин $G$. Выведите ее.

! (*д*) То, что мы назвали двойственностью, на самом деле двойственностью не является, потому что граф $G'$ необязательно изоморфен $G$. Приведите пример графа $G$, для которого двойственный граф $G'$ изоморфен $G$, и пример, когда графы $G$ и $G'$ не изоморфны.

## 10.2. Кластеризация графа социальной сети

Важная характеристика социальной сети – наличие сообществ сущностей, соединенных большим числом ребер. Типичные примеры сообществ: группы друзей в школе или группы ученых, работающих в одной области. В этом разделе мы рассмотрим кластеризацию графа как способ нахождения сообществ. Как выясняется, методы, изученные в главе 7, в общем случае не годятся для решения задачи о кластеризации графа социальной сети.

### 10.2.1. Метрики для графов социальных сетей

Если мы собираемся применить стандартные методы кластеризации к графу социальной сети, то, прежде всего, должны определить метрику. Если ребра графа снабжены метками, то иногда метки можно использовать в качестве полезной метрики – все зависит от того, что они обозначают. Но если ребра не помечены, как в графе «друзей», то возможностей для определения расстояния не так много.

Первое, что приходит в голову, – считать вершины близкими, если между ними имеется ребро, и далекими – в противном случае. То есть положить $d(x, y)$ равным 0, если существует ребро $(x, y)$, и 1, если такого ребра нет. Можно было бы использовать любые два значения, например 1 и $\infty$, лишь бы расстояние было меньше, когда ребро существует.

Но никакой выбор двух значений – 0 и 1 или 1 и ∞ – не дает настоящей метрики, поскольку для трех узлов, между которыми есть только два ребра, нарушается неравенство треугольника. Действительно, если существуют ребра (*A*,*B*) и (*B*,*C*), но нет ребра (*A*,*C*), то расстояние от *A* до *C* больше суммы расстояний от *A* до *B* и от *B* до *C*. Это можно было бы исправить, приняв, к примеру, что при наличии ребра расстояние равно 1, а при отсутствии – 1.5. Но, как мы увидим в следующем разделе, проблемы двузначных метрик не ограничиваются одним лишь неравенством треугольника.

## 10.2.2. Применение стандартных методов кластеризации

Напомним (см. раздел 7.1.2), что есть два общих подхода к кластеризации: иерархический (агломеративный) и на основе отнесения точек. Рассмотрим, как они работают для графов социальных сетей. Начнем с иерархических методов, описанных в разделе 7.2. Предположим, что межкластерное расстояние определено как минимальное расстояние между парами вершин по одной из каждого кластера.

Иерархическая кластеризация графа социальной сети начинается с объединения двух вершин, соединенных ребром. Затем на каждом шаге мы случайно выбираем ребро, не соединяющее вершины из одного кластера, и объединяем кластеры, которым принадлежат его вершины.

**Пример 10.3.** Рассмотрим граф, изображенный на рис. 10.1 и повторенный на рис. 10.3. Сначала договоримся, что считать сообществами. На верхнем уровне можно выделить два сообщества: {*A*,*B*,*C*} и {*D*,*E*,*F*,*G*}. Но можно также считать, что {*D*,*E*,*F*} и {*D*,*F*,*G*} – два подсообщества {*D*,*E*,*F*,*G*}; у них есть общие члены, поэтому никакой алгоритм чистой кластеризации их никогда не найдет. Наконец, можно было бы принять, что любые две вершины, соединенные ребром, являются сообществом размера 2, хотя такие сообщества не интересны.

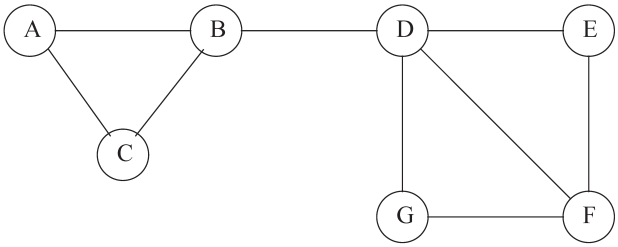



**Рис. 10.3.** Воспроизведение графа, изображенного на рис. 10.1

Проблема иерархической кластеризации графа наподобие изображенного на рис. 10.3 состоит в том, что рано или поздно мы с большой вероятностью решим объединить вершины *B* и *D*, хотя они, безусловно, принадлежат разным кластерам. Дело в том, что *D* и любой содержащий ее кластер находятся так же близко к *B* и любому содержащему ее кластеру, как вершины

*A* и *C*. Более того, с вероятностью 1/9 мы на первом же шаге объединим *B* и *D* в один кластер.

Чтобы уменьшить вероятность ошибки, мы можем сделать две вещи. Можно выполнить иерархическую кластеризацию несколько раз и выбрать прогон, на котором получились самые компактные кластеры. Или использовать один из более изощренных способов измерения расстояния между кластерами, обсуждавшихся в разделе 7.2.3. Но что бы мы ни делали, в большом графе, где много сообществ, велика вероятность, что на начальных шагах мы выберем ребра между вершинами, не принадлежащими одному и тому же крупному сообществу.

Теперь рассмотрим кластеризацию социальных сетей методом отнесения точек. И снова тот факт, что все ребра находятся на одном и том же расстоянии, вносит ряд случайных факторов, из-за которых некоторые вершины будут отнесены не к тому кластеру, что нужно. Проиллюстрируем на примере.

**Пример 10.4.** Допустим, что для кластеризации графа на рис. 10.3 применяется метод *k* средних. Поскольку нам нужно два кластера, положим $k = 2$. Две начальные вершины, выбранные случайным образом, могут оказаться в одном кластере. Если, как предлагалось в разделе 7.3.2, первую вершину выбрать случайно, а вторую – на максимальном удалении от первой, то ничего лучшего мы не получим. Поэтому можно просто взять любую пару узлов, не соединенных ребром, например *E* и *G*.

Предположим, однако, что мы все же выбрали две хороших начальных вершины, скажем *B* и *F*. Тогда мы отнесем *A* и *C* к кластеру *B*, а *E* и *G* – к кластеру *F*. Но *D* находится от *B* на таком же расстоянии, как от *F*, поэтому она может попасть в любой кластер, хотя «очевидно», что *D* и *F* должны быть вместе.

Если отложить решение об отнесении *D* до тех пор, пока не будут отнесены к кластерам какие-то другие вершины, то вполне может статься, что мы примем верное решение. Например, если относить вершину к кластеру с минимальным средним расстоянием до всех принадлежащих ему вершин, то *D* следует отнести к кластеру *F*, если только мы не пытаемся куда-то поместить *D* до того, как распределены все остальные вершины. Но в случае большого графа мы наверняка допустим ошибки на каких-то вершинах, относимых в начале.

## 10.2.3. Промежуточность

Поскольку стандартные методы кластеризации не дают решения, было разработано несколько специальных методов для нахождения сообществ в социальных сетях. В этом разделе мы рассмотрим один из самых простых, основанный на поиске ребер, которые с наименьшей вероятностью находятся внутри сообщества.

Определим *промежуточность* (betweenness) ребра $(a, b)$ как число пар вершин $x$ и $y$ таких, что ребро $(a, b)$ лежит на кратчайшем пути между $x$ и $y$. Точнее, по-

скольку может существовать несколько кратчайших путей между $x$ и $y$, в промежуточность ребра $(a, b)$ включается лишь часть кратчайших путей, содержащих $(a, b)$. Как и в гольфе, чем больше счет, тем хуже. Высокая промежуточность означает, что ребро $(a, b)$ соединяет два разных сообщества, т. е. $a$ и $b$ не принадлежат одному сообществу.

**Пример 10.5.** На рис. 10.3 самая высокая промежуточность у ребра (*B*,*D*), что и не удивительно. Действительно, это ребро лежит на каждом кратчайшем пути между любой из вершин *A*, *B*, *C* и любой из вершин *D*, *E*, *F*, *G*. Поэтому его промежуточность равна $3 \times 4 = 12$. С другой стороны, ребро (*D*,*F*) лежит только на четырех кратчайших путях: от *A*, *B*, *C* и *D* к *F*.

## 10.2.4. Алгоритм Гирвана-Ньюмана

Чтобы воспользоваться промежуточностью ребер, мы должны вычислить число кратчайших путей, проходящих по каждому ребру. Мы опишем алгоритм Гирвана-Ньюмана (GN), который посещает каждую вершину $X$ один раз и вычисляет число кратчайших путей из $X$ во все остальные вершины, проходящих по каждому ребру. Сначала выполняется поиск в ширину в графе, начиная с вершины $X$. Отметим, что уровень каждой вершины при поиске в ширину равен длине кратчайшего пути от $X$ до этой вершины. Следовательно, ребра, соединяющие вершины одного и того же уровня, не могут лежать ни на каком кратчайшем пути от $X$.

Ребра, соединяющие вершины разных уровней, называются DAG-ребрами (DAG означает «ориентированный ациклический граф»). Каждое DAG-ребро лежит хотя бы на одном кратчайшем пути от корня $X$. Если существует DAG-ребро $(Y,Z)$, где $Y$ находится на уровне выше $Z$ (т. е. ближе к корню), то будем называть $Y$ родителем $Z$, а $Z$ – потомком $Y$, хотя, в отличие от дерева, родитель в DAG-графе может быть не единственным.

**Пример 10.6.** На рис. 10.4 показан порядок обхода в ширину графа на рис. 10.3, начиная с вершины *E*. Сплошными линиями показаны DAG-ребра, а штриховые соединяют вершины одного уровня.

На втором шаге алгоритма GN каждая вершина помечается количеством кратчайших путей до нее из корня. Сначала пометим корень числом 1. Затем будем спускаться вниз и помечать каждую вершину $Y$ суммой меток ее родителей.

**Пример 10.7.** На рис. 10.4 показаны метки всех вершин. Корень *E* помечен

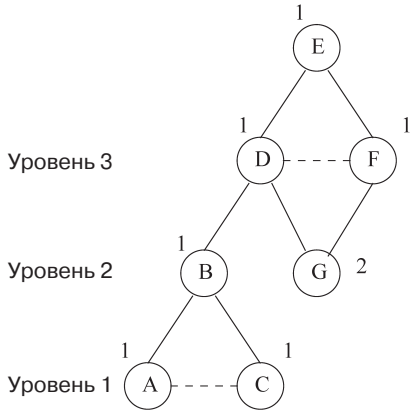



**Рис. 10.4.** Шаг 1 алгоритма Гирвана-Ньюмана

числом 1. На уровне 1 расположены вершины *D* и *F*. У каждой есть только один родитель, *E*, поэтому их метки тоже равны 1. Вершины *B* и *G* расположены на уровне 2. У *B* есть только родитель *D*, поэтому метка *B* равна метке *D*, т. е. 1. Но у *G* два родителя, *D* и *F*, поэтому метка *G* равна сумме двух меток, т. е. 2. Наконец, на уровне 3 вершины *A* и *C* имеют только одного родителя, *B*, так что их метки тоже равны 1.

На третьем и последнем шаге мы вычисляем для каждого ребра *e* сумму по всем вершинам *Y* доли кратчайших путей от *X* до *Y*, проходящих через *e*. При этом вычисление начинается снизу и производится как для вершин, так и для ребер. Каждой вершине, кроме корня, назначается *кредит* 1, представляющий кратчайший путь к этой вершине. Этот кредит может быть распределен между вершинами и ребрами, находящимися выше данной, поскольку может существовать несколько кратчайших путей к вершине. Правила вычисления таковы:

1. Каждая листовая вершина в DAG-графе (*листовой* называется вершина, из которой не исходит ни одного DAG-ребра в вершины на следующем уровне) получает кредит 1.
2. Каждая нелистовая вершина получает кредит 1 плюс сумма кредитов DAG-ребер, ведущих из данной вершины на следующий уровень.
3. DAG-ребру *e*, ведущему в вершину *Z* с предыдущего уровня, выделяется часть кредита *Z*, пропорциональная доле кратчайших путей из корня в *Z*, которые проходят через *e*. Формально: пусть $Y_1, Y_2, ..., Y_k$ – родители *Z*. Обозначим $p_i$ число кратчайших путей из корня в $Y_i$; это число было вычислено на шаге 2 и представлено метками на рис. 10.4. Тогда кредит ребра $(Y_i,Z)$ равен кредиту *Z*, умноженному на $p_i$ и поделенному на $\sum_{i=1}^{k} p_j$.

После вычисления кредитов при выборе каждой вершины в качестве корня мы суммируем кредиты ребер. Поскольку при этом каждый кратчайший путь будет учтен дважды – по разу для выбора каждого его конца в качестве корня, – необходимо разделить кредиты всех ребер на 2.

**Пример 10.8.** Вычислим кредиты для обхода в ширину графа, показанного на рис. 10.4. Начинаем с уровня 3 и поднимаемся вверх. Сначала листовые вершины *A* и *C* получают кредит 1. У каждой из них по одному родителю, поэтому их кредит передается ребрам (*B*,*A*) и (*B*,*C*) соответственно.

На уровне 2 вершина *G* листовая, поэтому она получает кредит 1. Вершина *B* нелистовая, поэтому ее кредит равен 1 плюс сумма кредитов DAG-ребер, входящих в нее снизу. Поскольку оба эти ребра имеют кредит 1, то кредит *B* равен 3. Интуитивно значение 3 означает, что все кратчайшие пути из *E* в *A*, *B* и *C* проходят через *B*. На рис. 10.5 показаны значения кредитов к этому моменту.

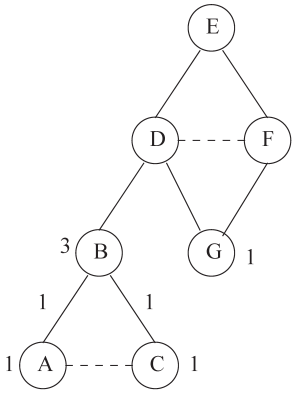



**Рис. 10.5.** Последний шаг алгоритма Гирвана-Ньюмана – уровни 3 и 2

Переходим к уровню 1. У *B* только один родитель, *D*, поэтому ребро (*D*,*B*) получает весь кредит *B*, равный 3. Однако у *G* два родителя, *D* и *F*. Поэтому мы должны распределить кредит 1, которым располагает *G*, между ребрами (*D*,*G*) и (*F*,*G*). В какой пропорции? На рис. 10.4 вершины *D* и *F* имеют метку 1, и это означает, что в каждую из них ведет только один кратчайший путь из *E*. Поэтому мы делим кредит *G* поровну между этими ребрами, т. е. кредит каждой равен 1/(1 + 1) = 0.5. Если бы у вершин *D* и *F* на рис. 10.4 были метки 5 и 3, т. е. существовало бы пять кратчайших путей в *D* и только три в *F*, то ребро (*D*,*G*) получило бы кредит 5/8, а ребро (*F*,*G*) – кредит 3/8.

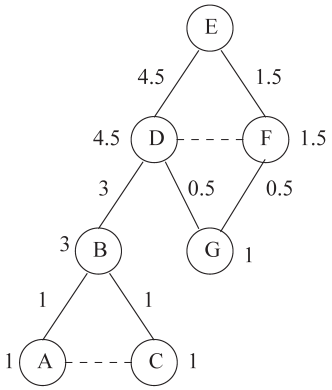



**Рис. 10.6.** Последний шаг алгоритма Гирвана-Ньюмана – завершение вычисления кредитов

Теперь можно присваивать кредиты на уровне 1. Вершина *D* получает 1 плюс сумму кредитов ребер, входящих в нее снизу, – 3 и 0.5. Следовательно, кредит *D* равен 4.5. Кредит *F* равен 1 плюс кредит ребра (*F*,*G*), т. е. 1.5. Наконец, ребра (*E*,*D*) и (*E*,*F*) получают кредиты *D* и *F* соответственно, потому что у каждой из этих вершин только по одному родителю. Все эти кредиты показаны на рис. 10.6. Кредит каждого из ребер на рис. 10.6 составляет вклад в промежуточность этого ребра, обусловленный кратчайшими путями из *E*. Так, вклад ребра (*E*,*D*) равен 4.5.

Чтобы завершить вычисление промежуточности, мы должны повторить всю описанную выше процедуру, выбирая в качестве корня каждую из оставшихся вершин, а затем суммировать вклады. И напоследок разделить все результаты пополам, потому что каждый кратчайший путь учтен дважды – по разу для каждого конца.

## 10.2.5. Использование промежуточности для нахождения сообществ

Величина промежуточности ребер ведет себя в чем-то похоже на метрику, определенную на вершинах графа. Это не вполне метрика, потому что она не определена для пар вершин, не соединенных ребром, а даже если бы была определена, то не обязательно удовлетворяла бы неравенству треугольника. Однако мы можем произвести кластеризацию, если будем брать ребра в порядке возрастания промежуточности и добавлять их в граф по одному. На каждом шаге связные компоненты графа образуют некоторые кластеры. Чем большую промежуточность мы допускаем, тем больше получаем ребер и тем крупнее становятся кластеры.

Чаще эта идея реализуется как процесс удаления ребер. Из исходного графа мы удаляем ребра с максимальной промежуточностью, пока граф не распадется на подходящее число связных компонент.

### Ускорение вычисления промежуточностей

Сложность описанного в разделе 10.2.4 метода вычисления промежуточностей ребер составляет $O(ne)$, где $n$ – число вершин, а $e$ – число ребер графа. Точнее, обход графа в ширину, начиная с одной вершины, требует времени $O(e)$, как и оба шага вычисления меток. И это необходимо сделать для каждой из $n$ вершин.

Для большого графа – а когда сложность алгоритма равна $O(ne)$, даже граф с миллионом вершин следует считать большим – мы не можем позволить себе такие временные затраты. Но если выбрать случайное подмножество вершин и только для них выполнять поиск в ширину, то мы получим аппроксимацию промежуточности каждого ребра, достаточную для большинства приложений.

**Пример 10.9.** Рассмотрим граф на рис. 10.1. На рис. 10.7 показаны промежуточности всех его ребер. Их вычисление оставляем читателю в качестве упражнения. Нужно только не забыть, что между вершинами $E$ и $G$ существует два кратчайших пути: один проходит через $D$, другой – через $F$. Поэтому каждому из ребер $(D,E)$, $(E,F)$, $(D,G)$ и $(G,F)$ достается половина кратчайшего пути в качестве кредита.

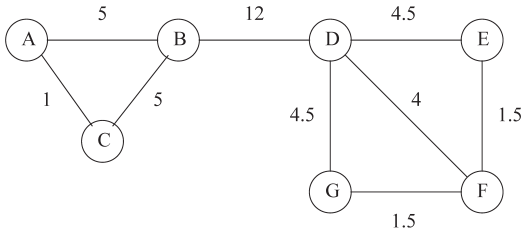



**Рис. 10.7.** Промежуточности для графа, изображенного на рис. 10.1

Ясно, что максимальная промежуточность у ребра $(B,D)$, поэтому оно удаляется первым. После этого остаются именно те сообщества, которые мы сочли разумными в самом начале: $\{A,B,C\}$ и $\{D,E,F,G\}$. Но мы можем продолжить удаление ребер. Следующими нужно удалить ребра $(A,B)$ и $(B,C)$, имеющие промежуточность 5, за ними – ребра $(D,E)$ и $(D,G)$ с промежуточностью 4.5. Затем граф покинет ребро $(D,F)$ с промежуточностью 4. На рис. 10.8 показан получившийся граф.

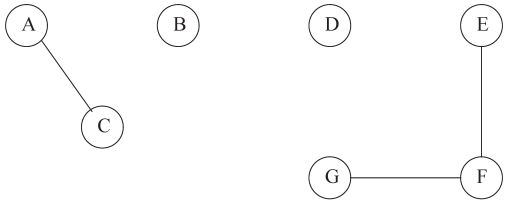



**Рис. 10.8.** Удалены все ребра с промежуточностью 4 и выше

«Сообщества» на рис. 10.8 выглядят странно. Можно сделать вывод, что *A* и *C* связаны между собой теснее, чем с *B*. В каком-то смысле *B* является по отношению к сообществу {*A*,*B*,*C*} «предателем», потому что имеет друга *D* вне этого сообщества. Аналогично *D* можно считать «предателем» по отношению к группе {*D*,*E*,*F*,*G*}, именно поэтому на рис. 10.8 остались соединены только вершины *E*, *F* и *G*.

### 10.2.6. Упражнения к разделу 10.2

**Упражнение 10.2.1.** На рис. 10.9 приведен пример графа социальной сети. С помощью алгоритма Гирвана-Ньюмана найдите, сколько существует кратчайших путей, проходящих по всем ребрам и начинающихся в вершине (*а*) *A*, (*б*) *B*.

**Упражнение 10.2.2.** Благодаря симметрии вычисления, проделанные в упражнении 10.2.1, – все, что необходимо для вычисления промежуточности каждого ребра. Выполните это вычисление.

**Упражнение 10.2.3.** Имея результаты упражнения 10.2.2, определите кандидатов на роль сообществ в графе на рис. 10.9, удалив ребра, промежуточность которых выше пороговой.

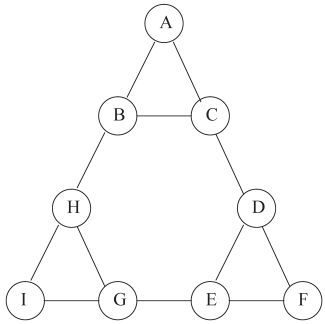



**Рис. 10.9.** Граф, используемый в упражнениях

## 10.3. Прямое нахождение сообществ

В предыдущем разделе мы искали сообщества путем разделения на группы всех участников социальной сети. И хотя этот подход относительно эффективен, у него есть ограничения. Никакого участника нельзя отнести к двум разным сообществам, и каждый участник оказывается отнесен к какому-то сообществу. В этом разделе мы рассмотрим прямой способ нахождения сообществ посредством поиска подмножеств вершин, между которыми относительно много ребер. Интересно, что этот метод поиска в большом графе включает нахождение больших частых предметных наборов – тема главы 6.

### 10.3.1. Нахождение клик

Если требуется найти множества вершин, соединенных большим числом ребер, то первая мысль – начать с поиска большой *клики* (множества вершин, каждые две из которых соединены ребром). Однако это отнюдь не просто. Мало того что поиск максимальных клик является NP-полной задачей, так это еще одна из самых трудных NP-полных задач в том смысле, что даже ее приближенное решение найти сложно. Кроме того, может случиться, что существует множество вершин, между которыми проведены почти все возможные ребра, и, тем не менее, число клик относительно мало.

**Пример 10.10.** Пусть вершины графа пронумерованы 1, 2, ..., $n$ и между вершинами $i$ и $j$ существует ребро, если только $i$ и $j$ не дают одинаковый остаток при делении на $k$. Тогда доля имеющихся ребер от числа потенциально возможных составляет приблизительно $(k - 1)/k$. Существует много клик размера $k$, например $\{1, 2, ..., k\}$.

Но вместе с тем нет ни одной клики размера больше $k$. Чтобы доказать это, заметим, что в любом множестве из $k + 1$ вершин есть хотя бы две, дающие одинаковый остаток при делении на $k$. Это простое следствие принципа Дирихле. Поскольку существует всего $k$ разных остатков, остатки от деления всех $k + 1$ вершин не могут быть различны. Поэтому никакое множество из $k + 1$ вершин этого графа не может быть кликой.

## 10.3.2. Полные двудольные графы

Вспомним обсуждение двудольных графов в разделе 8.3. *Полный двудольный граф* содержит $s$ вершин с одной стороны и $t$ вершин с другой стороны, причем существуют все $st$ возможных ребер между теми и другими вершинами. Обозначим такой граф $K_{s,t}$. Можно провести аналогию между полными двудольными графами как подграфами двудольных графов общего вида и кликами как подграфами графов общего вида. На самом деле, клика $s$ вершин часто называется *полным графом* и обозначается $K_s$, а полный двудольный граф иногда называют *бикликой*.

Как показывает пример 10.10, не всегда можно гарантировать, что в графе с большим числом ребер имеется крупная клика, однако есть гарантия, что в двудольном графе с большим числом ребер имеется большой полный двудольный подграф[1]. Мы можем рассматривать полный двудольный подграф (или клику, если таковая обнаружится) как ядро сообщества и добавлять в него вершины, имеющие много связей (ребер) с членами этого сообщества. Если сам граф $k$-дольный (см. раздел 10.1.4), то мы можем взять вершины каких-то двух типов и ребра между ними и образовать из них двудольный граф. В этом двудольном графе можно поискать полные двудольные графы, являющиеся ядрами сообществ. Например, в примере 10.2 мы могли бы ограничиться только вершинами графа на рис. 10.2, описывающими метки и веб-страницы, и попытаться найти сообщества меток и страниц. Такое сообщество состояло бы из родственных меток и родственных страниц, получивших все или большинство таких меток.

Однако полные двудольные графы можно использовать и для нахождения сообществ в обычных графах, когда все узлы однотипны. Случайным образом разобьем все вершины на две равные группы. Если сообщество существует, то можно ожидать, что в каждую группу попадет половина его вершин. Поэтому велики шансы найти большой полный двудольный граф в этом сообществе. К найденному ядру будем добавлять вершины из обеих групп, если между ними и уже идентифицированными членами сообщества существует много ребер.

[1] Важно понимать, что мы не имеем в виду *порожденный* подграф – такой, который образуется путем выборки некоторых вершин и включения всех имеющихся между ними ребер. В данном контексте мы требуем только, чтобы между любой парой вершин, по одной с каждой стороны, существовало ребро. Допускается также, что некоторые вершины по одну сторону соединены ребрами.

## 10.3.3. Нахождение полных двудольных подграфов

Пусть дан большой двудольный граф $G$, и мы хотим найти в нем экземпляры $K_{s,t}$. Эту задачу можно рассматривать как поиск частых предметных наборов. Будем считать «предметами» вершины в одной доле $G$, которую назовем для определенности *левой*. Мы предполагаем, что искомый экземпляр $K_{s,t}$ имеет $t$ вершин в левой доле, причем ради эффективности будем считать, что $t \leq s$. «Корзинам» соответствуют вершины в другой доле $G$ (*правой*). Элементами корзины вершины $v$ являются те вершины в левой доле, с которыми $v$ соединена. Наконец, пороговую поддержку положим равной $s$, числу узлов $K_{s,t}$ в правой доле.

Теперь задачу о нахождении экземпляров $K_{s,t}$ можно сформулировать как задачу о нахождении частых предметных наборов $F$ размера $t$. Если множество $t$ вершин в левой доле частое, то все они встречаются вместе по крайней мере в $s$ корзинах. Но корзины – это вершины в правой доле. Каждая корзина соответствует вершине, соединенной со всеми $t$ вершинами в $F$. Таким образом, частый предметный набор размера $t$ и $s$ корзин, в которых все эти предметы встречаются, образуют экземпляр $K_{s,t}$.

**Пример 10.11.** Рассмотрим еще раз двудольный граф, изображенный на рис. 8.1, который мы для удобства повторили на рис. 10.10. Левая доля состоит из вершин {1, 2, 3, 4}, а правая – из вершин {*a*, *b*, *c*, *d*}. Последние являются корзинами, так что корзина *a* включает «предметы» 1 и 4, т. е. $a = \{1, 4\}$. Аналогично $b = \{2, 3\}$, $c = \{1\}$ и $d = \{3\}$.

Если $s = 2$ и $t = 1$, то мы должны найти предметные наборы размера 1, встречающиеся по меньшей мере в двух корзинах. Один такой набор – {1}, другой – {3}. Но в этом крохотном примере нет предметных наборов для бóльших сколько-нибудь интересных значений $s$ и $t$, например $s = t = 2$.

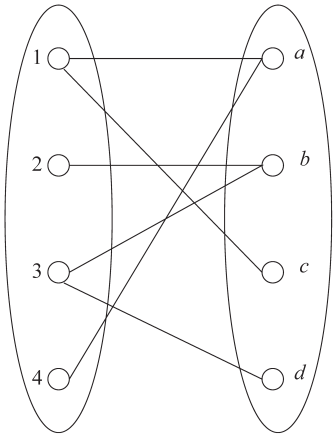



**Рис. 10.10.** Воспроизведение двудольного графа, изображенного на рис. 10.1

## 10.3.4. Почему должны существовать полные двудольные графы

Теперь докажем, что в любом двудольном графе с достаточно большим числом ребер обязательно существует экземпляр $K_{s,t}$. Далее будем предполагать, что граф $G$ содержит $n$ вершин слева и $n$ вершин справа. Предположение о равном числе вершин в обеих долях упрощает вычисления, но наше рассуждение обобщается на доли любого размера. Наконец, пусть $d$ – средняя степень вершин.

Подсчитаем количество частых предметных наборов размера $t$, в которые вносит вклад корзина, содержащая $d$ предметов. Просуммировав эту величину по всем вершинам в правой доле, мы найдем полную частоту всех поднаборов размера $t$ слева. Разделив на $\binom{n}{t}$, получим среднюю частоту всех наборов размера $t$. По крайней мере один набор должен иметь частоту, не меньшую средней, поэтому если среднее окажется не меньше $s$, то мы будем знать, что экземпляр $K_{s,t}$ существует.

Теперь опишем вычисление детально. Обозначим $d_i$ степень $i$-й вершины в правой доле, т. е. $d_i$ – размер $i$-й корзины. Эта корзина вносит вклад в $\binom{d_i}{t}$ предметных наборов размера $t$. Полный вклад всех $n$ вершин в правой доле равен $\sum_i \binom{d_i}{t}$. Конечно, значение этой суммы зависит от $d_i$. Но мы знаем, что среднее значение $d_i$ равно $d$. Известно, что эта сумма достигает минимума, когда все $d_i$ равны $d$. Мы не станем доказывать это утверждение, но простое соображение иллюстрирует ход рассуждений: поскольку $\binom{d_i}{t}$ растет, примерно как $t$-я степень $d_i$, вычитание 1 из большего $d_i$ и прибавление ее к меньшему $d_j$ уменьшает сумму $\binom{d_i}{t} + \binom{d_j}{t}$.

**Пример 10.12.** Предположим, что есть всего две вершины, т. е. $t = 2$, а средняя степень вершин равна 4. Тогда $d_1 + d_2 = 8$, и интересующая нас сумма равна $\binom{d_1}{2} + \binom{d_2}{2}$. Если $d_1 = d_2 = 4$, то эта сумма равна $\binom{4}{2} + \binom{4}{2} = 6 + 6 = 12$. Но если $d_1 = 5$ и $d_2 = 3$, то сумма равна $\binom{5}{2} + \binom{3}{2} = 10 + 3 = 13$. А если $d_1 = 6$ и $d_2 = 2$, то сумма равна $\binom{6}{2} + \binom{2}{2} = 15 + 1 = 16$.

Таким образом, в дальнейшем мы будем предполагать, что все вершины имеют среднюю степень $d$. При этом достигает минимума полный вклад в счетчики предметных наборов, а, значит, и вероятность существования частого набора (с поддержкой не меньше $s$) размера $t$. Сделаем следующие наблюдения:

- полный вклад $n$ вершин из правой доли в счетчики предметных наборов размера $t$ равен $n\binom{d}{t}$;
- количество предметных наборов размера $t$ равно $\binom{n}{t}$;
- следовательно, средний счетчик предметного набора размера $t$ равен $n\binom{d}{t}/\binom{n}{t}$; для доказательства существования экземпляра $K_{s,t}$ это выражение должно быть не меньше $s$.

Раскрывая биномиальные коэффициенты, находим

$$n\binom{d}{t}/\binom{n}{t} = nd!(n-t)!t! \,/\, ((d-t)!t!n!) =$$

$$n(d)(d-1)\ldots(d-t+1) \,/\, (n(n-1)\ldots(n-t+1))$$

Для упрощения этого выражения предположим, что $n$ много больше $d$, а $d$ много больше $t$. Тогда $d(d-1)\ldots(d-t+1)$ приближенно равно $d^t$ и $n(n-1)\ldots(n-t+1)$ приближенно равно $n^t$. Потребуем поэтому, чтобы

$$n(d/n)^t \geq s$$

Значит, если существует сообщество с $n$ вершинами в каждой доле, средняя степень вершин равна $d$ и $n(d/n)^t \geq s$, то гарантируется, что сообщество содержит пол-

ный двудольный подграф $K_{s,t}$. Более того, мы можем эффективно найти экземпляр $K_{s,t}$, применяя методы из главы 6, даже если это маленькое сообщество находится внутри гораздо большего графа. То есть мы можем рассматривать все вершины графа как корзины и предметы и, выполнив для всего графа алгоритм Apriori или один из его улучшенных вариантов, найти наборы $t$ предметов с поддержкой $s$.

**Пример 10.13.** Предположим, что существует сообщество со 100 вершинами в каждой доле, а средняя степень вершин равна 50, т. е. существует половина всего возможного числа ребер. В этом сообществе найдется экземпляр $K_{s,t}$ при условии, что $100(1/2)^t \geq s$. Например, если $t = 2$, то $s$ может быть не больше 25. Если $t = 3$, то $s$ может быть не больше 11, а если $t = 4$, то не больше 6.

К сожалению, наша аппроксимация несколько завышает верхнюю границу $s$. Вернувшись к исходной формуле $n\binom{d}{t}/\binom{n}{t} \geq s$, мы увидим, что в случае $t = 4$ необходимо, чтобы выполнялось неравенство $100\binom{50}{4}/\binom{100}{4} \geq s$, или

$$\frac{100 \times 50 \times 49 \times 48 \times 47}{100 \times 99 \times 98 \times 97} \geq s \,.$$

Выражение в левой части равно не 6, а всего 5.87. Но если средняя поддержка для предметного набора размера 4 равна 5.87, то не может быть, чтобы все такие наборы имели поддержку 5 или меньше. Следовательно, мы можем быть уверены, что по крайней мере у одного набора размера 4 поддержка не меньше 6 и, значит, в сообществе существует экземпляр $K_{6,4}$.

## 10.3.5. Упражнения к разделу 10.3

**Упражнение 10.3.1.** В примере социальной сети, изображенной на рис. 10.1, сколько экземпляров $K_{s,t}$ существует при:

(*а*) $s = 1$ и $t = 3$;
(*б*) $s = 2$ и $t = 2$;
(*в*) $s = 2$ и $t = 3$.

**Упражнение 10.3.2.** Пусть имеется сообщество с $2n$ вершинами. Разобьем его случайным образом на две группы по $n$ членов и образуем из них двудольный граф. Обозначим среднюю степень вершин этого графа $d$. Найдите множество максимальных пар $(t, s)$, где $t \leq s$, при которых гарантированно существует экземпляр $K_{s,t}$, для следующих комбинаций $n$ и $d$:

(*а*) $n = 20$ и $d = 5$;
(*б*) $n = 200$ и $d = 150$;
(*в*) $n = 1000$ и $d = 400$.

Говоря «максимальная», мы имеем в виду, не существует другой пары $(s', t')$, для которой одновременно справедливы неравенства $s' \geq s$ и $t' \geq t$.

# 10.4. Разрезание графов

В этом разделе мы рассмотрим другой подход к организации графов социальных сетей. Воспользуемся некоторыми важными инструментами из теории матриц (спектральными методами) для постановки задачи о разбиении графов таким образом, чтобы минимизировать число ребер, соединяющих различные компоненты. Но сначала нужно ясно понять, в чем состоит цель минимизации размера «разреза». Например, если вы только что зарегистрировались в Facebook, то еще не обзавелись друзьями. Мы не хотим разрезать граф друзей, так чтобы вы оказались в одной части, а весь остальной мир – в другой, хотя в этом случае вообще не оказалось бы ребер, соединяющих вершины из разных частей. Однако такой разрез нежелателен, потому что размеры компонент слишком сильно различаются.

## 10.4.1. Какое разрезание считать хорошим?

Мы хотели бы разбить множество вершин графа на такие два подмножества, чтобы *разрез*, т. е. множество ребер, соединяющих вершины из разных подмножеств, был минимален. Но при этом мы хотели бы наложить ограничение: разрез должен быть таким, чтобы размеры обоих подмножеств были примерно одинаковы. В примере ниже иллюстрируется эта мысль.

**Пример 10.14.** Вернемся к графу на рис. 10.1. Очевидно, что при наилучшем разрезании вершины {*A*,*B*,*C*} помещаются в одно множество, а вершины {*D*,*E*,*F*,*G*} – в другое. Разрез состоит из единственного ребра (*B*,*D*), т. е. его размер равен 1. Нетривиального разреза меньшего размера просто не может быть.

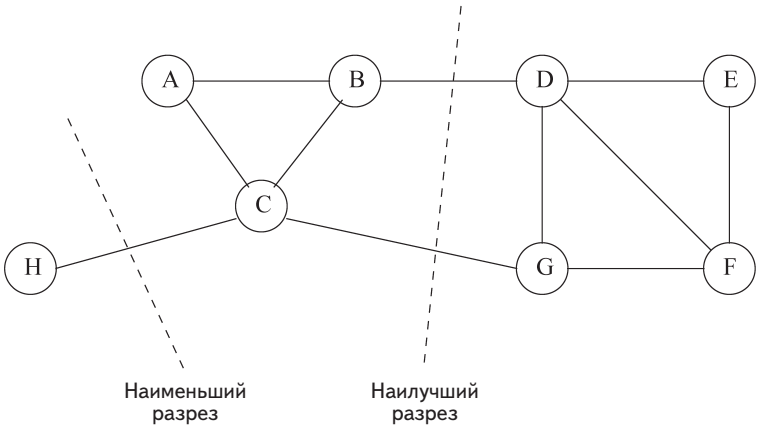



**Рис. 10.11.** Наименьший разрез может не быть наилучшим

На рис. 10.11 показан вариант этого примера, где мы добавили вершину *H* и два ребра (*H*,*C*) и (*C*,*G*). Если бы целью была только минимизация размера разреза, то лучше всего было бы поместить *H* в одно множество, а все остальные вершины – в другое. Но видно, что если мы отвергаем раз-

резания, в которых одно множество слишком мало, то лучшее, что можно сделать, – взять разрез из двух ребер ($B$,$D$) и ($C$,$G$), который делит граф на две части одинакового размера: {$A$,$B$,$C$,$H$} и {$D$,$E$,$F$,$G$}.

## 10.4.2. Нормализованные разрезы

Правильное определение «хорошего» разреза должно соблюдать баланс между размером самого разреза и разницей в размерах множеств, на которые разрез делит вершины графа. Одно из таких определений – «нормализованный разрез». Сначала определим *объем* множества вершин $S$, обозначаемый $Vol(S)$, как число таких ребер, что по крайней мере один конец ребра принадлежит $S$.

Допустим, что множество вершин графа разбито на два непересекающихся подмножества $S$ и $T$. Обозначим $Cut(S,T)$ число ребер, один конец которых принадлежит $S$, а другой – $T$. Тогда величина *нормализованного разреза* для $S$ и $T$ равна

$$\frac{Cut(S,T)}{Vol(S)} + \frac{Cut(S,T)}{Vol(T)}.$$

**Пример 10.15.** Снова рассмотрим граф на рис. 10.11. Если взять $S$ = {$H$}, $T$ = {$A$,$B$,$C$,$D$,$E$,$F$,$G$}, то $Cut(S,T)$ = 1. $Vol(S)$ = 1, потому что существует только одно ребро с вершиной $H$. С другой стороны, $Vol(T)$ = 11, потому что по крайней мере один конец любого ребра принадлежит $T$. Следовательно, нормализованный разрез при таком разрезании равен 1/1 + 1/11 = 1.09.

Теперь рассмотрим предпочтительный разрез того же графа, состоящий из ребер ($B$,$D$) и ($C$,$G$). Тогда $S$ = {$A$,$B$,$C$,$H$} и $T$ = {$D$,$E$,$F$,$G$}. $Cut(S,T)$ = 2, $Vol(S)$ = 6, $Vol(T)$ = 7. В этом случае нормализованный разрез равен только 2/6 + 2/7 = 0.62.

## 10.4.3. Некоторые матрицы, описывающие графы

Чтобы понять, как матричная алгебра помогает находить хорошие разрезания графов, нам нужно сначала определить три матрицы, описывающие различные аспекты графа. Первая вам, наверное, знакома; это *матрица смежности*, в которой на пересечении строки $i$ и столбца $j$ находится 1, если существует ребро, соединяющее вершины $i$ и $j$, а все остальные элементы равны 0.

**Пример 10.16.** На рис. 10.12 воспроизведен наш любимый пример. Его матрица смежности показана на рис. 10.13. Отметим, что строки и столбцы соответствуют вершинам $A$,$B$, ..., $G$ именно в таком порядке. О наличии ребра ($B$,$D$) мы знаем, потому что элемент на пересечении строки 2 и столбца 4 равен 1, как и элемент на пересечении строки 4 и столбца 2.

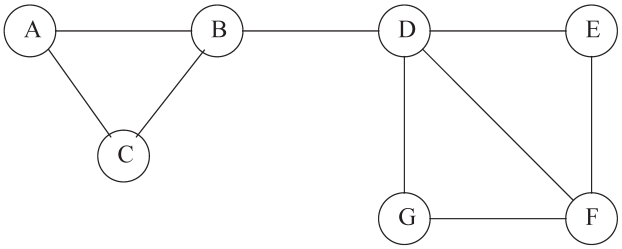



**Рис. 10.12.** Воспроизведение графа, изображенного на рис. 10.1

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

**Рис. 10.13.** Матрица смежности графа на рис. 10.12

Вторая матрица называется *степенной матрицей* графа. В ней ненулевые элементы находятся только на диагонали. Элемент на пересечении $i$-ой строки и $i$-го столбца равен степени вершины $i$.

**Пример 10.17.** Степенная матрица графа на рис. 10.12 показана на рис. 10.14. Используется тот же порядок вершин, что в примере 10.16. Так, элемент на пересечении строки 4 и столбца 4 равен 4, потому что вершина $D$ соединена ребрами с четырьмя вершинами. Элемент на пересечении строки 4 и столбца 5 равен 0, потому что он не находится на главной диагонали.

$$\begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

**Рис. 10.14.** Степенная матрица графа на рис. 10.12

Пусть $A$ обозначает матрицу смежности графа, а $D$ – его степенную матрицу. *Матрицей Лапласа* $L = D - A$ называется разность между степенной матрицей и матрицей смежности. Это означает, что на диагонали матрицы Лапласа находятся те же элементы, что на диагонали $D$, а элемент на пересечении строки $i$ и столбца $j$, где $i \neq j$, равен $-1$, если вершины $i$ и $j$ соединены ребром, и 0 в противном случае.

**Пример 10.18.** Матрица Лапласа графа на рис. 10.12 показана на рис. 10.15. Отметим, что суммы элементов в любой строке и в любом столбце равны 0, и это справедливо для любой матрицы Лапласа.

$$\begin{bmatrix} 2 & -1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 3 & -1 & -1 & 0 & 0 & 0 \\ -1 & -1 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 4 & -1 & -1 & -1 \\ 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & -1 & 3 & -1 \\ 0 & 0 & 0 & -1 & 0 & -1 & 2 \end{bmatrix}$$

**Рис. 10.15.** Матрица Лапласа графа на рис. 10.12

## 10.4.4. Собственные значения матрицы Лапласа

Составить представление о наилучшем разрезании графа можно, зная собственные значения и собственные векторы матрицы Лапласа. В разделе 5.1.2 мы замети-

ли, что главный собственный вектор (с наибольшим собственным значением) матрицы переходов в вебе сообщает полезную информацию о важности веб-страниц. Более того, в простых случаях (без телепортации) главный собственный вектор и есть вектор PageRank. Но в случае матрицы Лапласа нужную нам информацию дают наименьшие собственные значения и их собственные векторы.

Наименьшее собственное значение любой матрицы Лапласа равно 0, а его собственный вектор равен [1, 1, ..., 1]. Чтобы доказать это, обозначим $L$ матрицу Лапласа графа с $n$ вершинами, и пусть $\mathbf{1}$ обозначает вектор-столбец размерности $n$, все элементы которого равны 1. Мы утверждаем, что произведение $L\mathbf{1}$ – вектор-столбец, все элементы которого равны 0. Действительно, рассмотрим строку $i$ матрицы $L$. На пересечении ее с диагональю находится степень $d$ вершины $i$. Кроме того, в $i$-й строке $d$ раз встречается $-1$, а все остальные элементы равны 0. Умножение строки $i$ на вектор-столбец $\mathbf{1}$ сводится к суммированию элементов строки, и понятно, что эта сумма равна $d + (-1)d = 0$. Следовательно, $L\mathbf{1} = 0\mathbf{1}$, т. е. 0 – собственное значение, а $\mathbf{1}$ – соответствующий ему собственный вектор.

Существует простой способ найти второе снизу по величине собственное значение любой *симметричной* матрицы (в которой элемент на пересечении строки $i$ и столбца $j$ равен элемент на пересечении строки $j$ и столбца $i$), в том числе матрицы Лапласа. Доказывать это утверждение мы не будем, а просто сообщим, что второе по величине собственное значение $L$ равно минимуму $\mathbf{x}^{\mathrm{T}}L\mathbf{x}$, где $\mathbf{x} = [x_1, x_2, ..., x_n]$ – $n$-мерный вектор-столбец, а минимум вычисляется при следующих условиях:

1. Длина $\mathbf{x}$ равна 1, т. е. $\sum_{i=1}^{n} x_i^2 = 1$.
2. $\mathbf{x}$ ортогонален собственному вектору с наименьшим собственным значением.

Кроме того, значение $\mathbf{x}$, доставляющее минимум, и есть второй собственный вектор.

Если $L$ – матрица Лапласа графа с $n$ вершинами, то можно сказать больше. Собственный вектор, соответствующий наименьшему собственному значению, равен $\mathbf{1}$. Следовательно, раз $\mathbf{x}$ ортогонален $\mathbf{1}$, то должно быть

$$\mathrm{x}^{\mathrm{T}}1 = \sum_{i=1}^{n} x_1 = 0 .$$

Кроме того, для матрицы Лапласа выражение $\mathbf{x}^{\mathrm{T}}L\mathbf{x}$ может быть записано в полезной эквивалентной форме. Напомним, что $L = D - A$, где $D$ – степенная матрица, а $A$ – матрица смежности графа. Следовательно, $\mathbf{x}^{\mathrm{T}}L\mathbf{x} = \mathbf{x}^{\mathrm{T}}D\mathbf{x} - \mathbf{x}^{\mathrm{T}}A\mathbf{x}$. Вычислим оба члена по отдельности. Здесь $D\mathbf{x}$ – вектор-столбец $[d_1x_1, d_2x_2, ..., d_nx_n]$, где $d_i$ – степень $i$-ой вершины графа. Значит, $\mathbf{x}^{\mathrm{T}}D\mathbf{x} = \sum_{i=1}^{n} d_i x_i^2$.

Теперь обратимся к $\mathbf{x}^{\mathrm{T}}A\mathbf{x}$. $i$-й элемент вектора-столбца $A\mathbf{x}$ равен сумме $x_j$ по всем таким $j$, что в графе существует ребро $(i, j)$. Следовательно, $-\mathbf{x}^{\mathrm{T}}A\mathbf{x}$ – это сумма $-2x_ix_j$ по всем парам вершин $\{i, j\}$ таким, что между ними существует ребро. Множитель 2 включен, потому что каждому ребру $\{i, j\}$ соответствуют два члена: $-x_ix_j$ и $-x_jx_i$.

Мы можем сгруппировать члены $\mathbf{x}^{\mathrm{T}}L\mathbf{x}$, распределив их между всеми парами $\{i, j\}$. Из $-\mathbf{x}^{\mathrm{T}}A\mathbf{x}$ возьмем член $-2x_ix_j$, а член $d_ix^2{}_i$ из $\mathbf{x}^{\mathrm{T}}D\mathbf{x}$ распределим между $d_i$ парами, содержащими вершину $i$. В результате с каждой парой вершин $\{i, j\}$, соединенных ребром, будет ассоциировано выражение $x^2{}_i - 2x_ix_j + x^2{}_j$, равное $(x_i - x_j)^2$. Таким образом, мы доказали, что $\mathbf{x}^{\mathrm{T}}L\mathbf{x}$ равно сумме $(x_i - x_j)^2$ по всем ребрам графа.

Напомним, что второе по величине собственное значение равно минимуму этого выражения при ограничении $\sum_{i=1}^{n} x^2{}_i = 1$. Интуитивно понятно, что для его минимизации нужно сблизить $x_i$ и $x_j$, когда между вершинами $i$ и $j$ в графе существует ребро. Можно было бы подумать, что достаточно взять $x_i = 1/\sqrt{n}$ для всех $i$ и тем самым обратить эту сумму в 0. Напомним, однако, что мы ограничены условием ортогональности $\mathbf{x}$ и $\mathbf{1}$, в силу которого сумма всех $x_i$ должна быть равна 0. Кроме того, требуется, чтобы $\sum_{i=1}^{n} x^2{}_i = 1$, поэтому все элементы не могут быть равны 0. Следовательно, среди элементов вектора $\mathbf{x}$ должны быть как положительные, так и отрицательные.

Для разрезания графа мы можем взять в качестве одного множества вершины $i$, которым соответствует положительный элемент вектора $x_i$, а в качестве другого – вершины, которым соответствует отрицательный элемент. Такой выбор не гарантирует равенство размеров обоих множеств, но с большой вероятностью они будут близки. Мы полагаем, что разрез будет содержать не слишком много ребер, потому что $(x_i - x_j)^2$ скорее окажется меньше, если знаки $x_i$ и $x_j$ одинаковы, чем если они различны. Поэтому можно ожидать, что в векторе $\mathbf{x}$, минимизирующем $\mathbf{x}^{\mathrm{T}}L\mathbf{x}$ при заданных ограничениях, элементы $x_i$ и $x_j$ скорее будут иметь одинаковый знак, если существует ребро $(i, j)$.

**Пример 10.19.** Применим описанный выше способ к графу на рис. 10.16. Его матрица Лапласа показана на рис. 10.17. Применяя стандартные методы или пакеты математических программ, можно найти все собственные значения и собственные векторы этой матрицы. Они сведены в таблицу на рис. 10.18, начиная с наименьшего собственного значения. Мы не масштабировали собственные векторы на длину 1, но при желании легко могли это сделать.

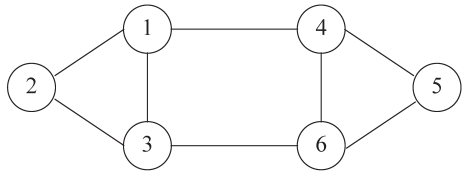



**Рис. 10.16.** Граф для иллюстрации разрезания методами спектрального анализа

$$\begin{bmatrix} 3 & -1 & -1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ -1 & -1 & 3 & 0 & 0 & -1 \\ -1 & 0 & 0 & 3 & -1 & -1 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & -1 & -1 & 3 \end{bmatrix}$$

**Рис. 10.17.** Матрица Лапласа для графа на рис. 10.16

| **Собственное значение** | 0 | 1 | 3 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Собственный вектор** | 1 | 1 | -5 | -1 | -1 | -1 |
| | 1 | 2 | 4 | -2 | 1 | 0 |
| | 1 | 1 | 1 | 3 | -1 | 1 |
| | 1 | -1 | -5 | -1 | 1 | 1 |
| | 1 | -1 | 4 | -2 | -1 | 0 |
| | 1 | -1 | 1 | 3 | 1 | -1 |

**Рис. 10.18.** Собственные значения и собственные векторы матрицы на рис. 10.17

У второго собственного вектора три положительных и три отрицательных элемента. Отсюда вытекает рекомендация объединить в одну группу вершины {1, 2, 3}, соответствующие положительным элементам, а в другую – вершины {4, 5, 6}.

## 10.4.5. Другие методы разрезания

Метод из раздела 10.4.4 дает хорошее разрезание графа на две части с небольшим разрезом. Есть несколько способов использовать те же самые собственные векторы для получения других разрезаний. Прежде всего, мы не обязаны помещать все вершины, соответствующие положительным элементам собственного вектора, в одну группу, а отрицательным – в другую. Можно было бы взять порог, отличный от 0.

Например, модифицируем пример 10.19, выбрав в качестве порога не 0, а –1.5. Тогда обе вершины 4 и 6, соответствующие элементам –1 второго собственного вектора на рис. 10.18, присоединятся к вершинам 1, 2, 3, так что в одной группе будет пять вершин, а в другой – только вершина 5. В этом случае размер разреза равен 2, как и при выборе порога 0, но размеры компонент сильно отличаются, так что лучше бы предпочесть первоначальный выбор. Но бывает и так, что при нулевом пороге получаются неравные компоненты, например, если бы мы воспользовались третьим собственным вектором на рис. 10.18.

Иногда требуется разрезать граф на большее число частей. Один из возможных подходов – воспользоваться описанным выше методом для разрезания на две части, а затем применять его же к получившимся частям столько раз, сколько необходимо. Другой подход – использовать для разрезания несколько собственных векторов, а не только второй. Если задействовать $m$ собственных векторов, задав для каждого свой порог, то мы разрежем граф на $2^m$ частей, каждая из которых будет содержать вершины, большие или меньшие пороговых значений, для каждой комбинации собственных векторов.

Стоит отметить, что каждый собственный вектор $\mathbf{x}$, кроме первого, минимизирует $\mathbf{x}^{\mathrm{T}}L\mathbf{x}$ при соблюдении ограничения ортогональности со всеми предыдущими собственными векторами. Это естественное обобщение ограничения на второй собственный вектор. В результате получается, что хотя каждый собственный вектор стремится получить разрез минимального размера, из-за наличия все большего числа ограничений на каждом последующем шаге разрезы оказываются хуже и хуже.

**Пример 10.20.** Снова рассмотрим граф на рис. 10.16, собственные векторы его матрицы Лапласа сведены в таблицу на рис. 10.18. Если взять третий собственный вектор и порог 0, то вершины 1 и 4 попадут в одну группу, а остальные четыре вершины – в другую. Это не плохое разрезание, но размер его разреза равен 4, а не 2, как при выборе второго собственного вектора.

Если взять второй и третий собственные векторы, то вершины 2 и 3 попадут в одну группу, потому что соответствующие им элементы положительны в обоих собственных векторах. Вершины 5 и 6 попадут в другую группу, по-

тому что соответствующие им элементы второго собственного вектора положительны, а третьего – отрицательны. Вершина 1 оказывается в отдельной группе, потому что соответствующий ей элемент второго собственного вектора положителен, а третьего – отрицателен. И наконец, вершина 4 тоже выделена в отдельную группу, потому что соответствующие ей элементы обоих собственных векторов отрицательны. Это разрезание графа с шестью вершинами на четыре группы слишком мелкое для практических целей. Но, по крайней мере, в каждой группе размера 2 имеется ребро между входящими в нее вершинами, так что никакое разрезание на группы такого размера не могло бы быть лучше.

### 10.4.6. Упражнения к разделу 10.4

**Упражнение 10.4.1.** Для графа на рис. 10.9 постройте:

(*а*) матрицу смежности;
(*б*) степенную матрицу;
(*в*) матрицу Лапласа.

**! Упражнение 10.4.2.** Для матрицы Лапласа, построенной в упражнении 10.4.1 (*в*), найдите второе по величине собственное значение и его собственный вектор. Какое им соответствует разрезание?

**!! Упражнение 10.4.3**. Для матрицы Лапласа, построенной в упражнении 10.4.1 (*в*), найдите третье и последующие собственные значения и их собственные векторы.

## 10.5. Нахождение пересекающихся сообществ

До сих пор мы занимались кластеризацией социального графа для нахождения сообществ. Но на практике сообщества редко бывают непересекающимися. В этом разделе мы опишем метод нахождения для социального графа модели, которая наилучшим образом объясняет, как он получился. Для этого будем предполагать, что вероятность существования ребра между двумя вершинами («друзьями») тем больше, чем больше число сообществ, в которые они входят одновременно. Важным инструментом нашего анализа будет «оценка максимального правдоподобия», которую мы рассмотрим до того, как переходить к вопросу о поиске пересекающихся сообществ.

### 10.5.1. Природа сообществ

Для начала подумаем, как должны были бы выглядеть перекрывающиеся сообщества. Данными в нашем случае является социальный граф, вершины которого представляют людей, а между двумя вершинами существует ребро, если люди являются «друзьями». Представим, что граф описывает учеников школы, в которой

есть два клуба: шахматный и испанский. Разумно предположить, что вокруг обоих клубов формируются сообщества, как в любой школе. Также разумно предположить, что два члена шахматного клуба скорее будут друзьями в графе, потому что знают друга по клубу. И точно так же два члена испанского клуба, наверное, знакомы и потому будут друзьями в графе.

А если два человека состоят членами обоих клубов? Теперь у них две причины быть знакомыми, поэтому вероятность оказаться друзьями в социальном графе еще выше. Таким образом, мы заключаем, что плотность ребер высока в любом сообществе, но еще выше в пересечении двух сообществ. И так далее. Эту мысль иллюстрирует рис. 10.19.

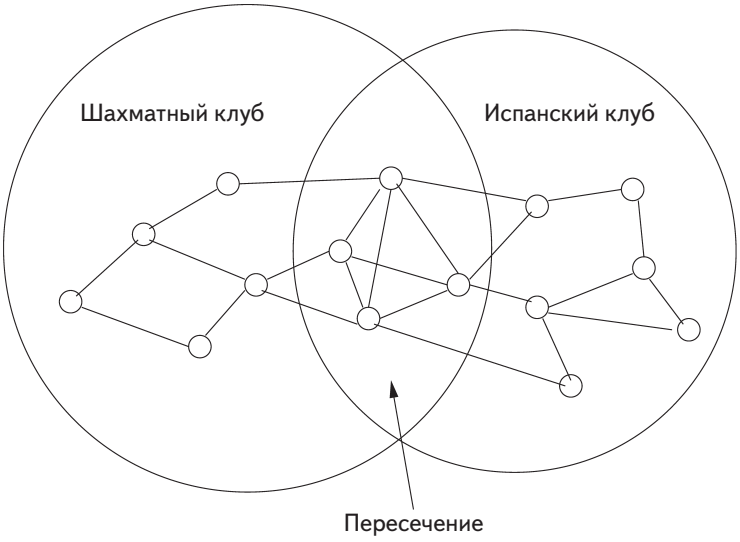



**Рис. 10.19.** Плотность ребер в пересечении двух сообществ выше, чем в их непересекающихся частях

## 10.5.2. Оценка максимального правдоподобия

Прежде чем излагать алгоритм нахождения сообществ, перекрывающихся, как описано в разделе 10.5.1, немного отвлечемся и поговорим о полезном средстве моделирования – *оценке максимального правдоподобия* (ОМП). Идея ОМП заключается в выдвижении гипотезы о порождающем процессе (модели), создавшем экземпляры некоторого явления, например, «графы друзей». У модели имеются параметры, определяющие вероятность порождения любого конкретного экземпляра; эта вероятность называется *правдоподобием* данных значений параметров. Предполагается, что модель, параметры которой дают максимальное значение правдоподобия, и является правильной моделью наблюдаемого явления.

Поясним идею ОМП на примере. Допустимы, мы генерируем случайные графы. Пусть вероятность наличия любого ребра равна $p$, а отсутствия – $1 - p$, причем события наличия и отсутствия ребра независимы. Единственный настраиваемый

параметр – величина $p$. Для любого значения $p$ существует небольшая, но ненулевая вероятность, что будет порожден именно тот граф, который мы видим. Согласно принципу ОМП, мы должны объявить, что истинным значением $p$ является то, для которого вероятность порождения наблюдаемого графа максимальна.

**Пример 10.21.** Рассмотрим граф на рис. 10.19. В нем 15 вершин и 23 ребра. Существует $\binom{15}{2} = 105$ пар из 15 вершин, поэтому, если каждое ребро выбирается с вероятностью $p$, то вероятность (правдоподобие) порождения именно того графа, который изображен на рис. 10.19, определяется функцией $p^{23}(1-p)^{82}$. При любом значении $p$ от 0 до 1 это число очень мало. Но у функции все же есть максимум, который можно найти, приравняв производную нулю:

$$23p^{22}(1-p)^{82} - 82p^{23}(1-p)^{81} = 0.$$

Перегруппируем члены

$$p^{22}(1-p)^{81}(23(1-p) - 82p) = 0.$$

Правая часть может обратиться в 0 только тогда, когда $p$ равно 0 или 1 либо последний сомножитель

$$(23(1-p) - 82p)$$

равен 0. При $p$, равном 0 или 1, функция правдоподобия $p^{23}(1-p)^{82}$ достигает минимума, а не максимума, поэтому в 0 должен обращаться последний сомножитель. Таким образом, правдоподобие порождения графа на рис. 10.19 оказывается максимальным, когда

$$23 - 23p - 82p = 0$$

или $p = 23/105$.

Этот результат не должен вызывать удивления. Он означает лишь, что наиболее правдоподобное значение $p$ совпадает с тем, какую долю от общего числа возможных ребер составляют ребра, присутствующие в наблюдаемом графе. Однако если используется более сложный механизм порождения графов или других объектов, значения параметров, при которых с максимальным правдоподобием порождается наблюдаемый объект, отнюдь не очевидны.

### Априорные вероятности

В ходе анализа ОМП мы обычно предполагаем, что параметры могут принимать любое значение из своей области определения, и никакого смещения в пользу определенных значений нет. Если же это не так, то можно умножить выражение для вероятности порождения наблюдаемого объекта, записанное в виде функции от значений параметров, на функцию, которая представляет относительную вероятность того, что эти значения параметров являются истинными. В упражнениях будут приведены примеры ОМП с предположениями об априорном распределении параметров.

### 10.5.3. Модель графа принадлежности

Мы опишем разумный механизм порождения социальных графов из сообществ, который называется *моделью графа принадлежности* (affiliation-graph model). Поняв, как параметры модели влияют на правдоподобие наблюдения данного графа, мы сможем подойти к вопросу о нахождении значений параметров, при которых правдоподобие достигает максимума. Введем следующие предположения.

1. Имеется заданное число сообществ и заданное число лиц (вершин графа).
2. Членами каждого сообщества может быть произвольное число лиц. Таким образом, членство в сообществах – параметры модели.
3. С каждым сообществом $C$ ассоциирована вероятность $p_C$ существования ребра между двумя лицами вследствие их принадлежности к сообществу $C$. Эти вероятности также являются параметрами модели.
4. Если две вершины входят в два или более сообществ, то между ними существует ребро, если какое-либо сообщество, в которое они оба входят, оправдывает наличие такого ребра согласно правилу (3).

Мы должны вычислить правдоподобие того, что заданный граф порожден описанным механизмом. Ключевое наблюдение касается того, как вычисляются вероятности ребер при заданном распределении лиц по сообществам и значениях $p_C$. Рассмотрим ребро $(u, v)$ между вершинами $u$ и $v$. Предположим, что $u$ и $v$ – члены сообществ $C$ и $D$ и ни в какие другие сообщества не входят. Тогда вероятность отсутствия ребра между $u$ и $v$ равна произведению вероятности отсутствия ребра, обусловленного вхождением в $C$, и вероятности отсутствия ребра, обусловленного вхождением в $D$. То есть ребра $(u, v)$ нет в графе с вероятностью $(1 - p_C)(1 - p_D)$, а вероятность, что оно есть в графе, – единице минус это значение.

В общем случае, если $u$ и $v$ – члены непустого множества сообществ $M$ и не входят ни в какие другие сообщества, то вероятность $p_{uv}$ существования ребра между $u$ и $v$ описывается формулой:

$$p_{uv} = 1 - \prod_{C \in M} (1 - p_C).$$

Имеется важный особый случай: если $u$ и $v$ не входят совместно ни в одно сообщество, то полагаем $p_{uv}$ равным $\varepsilon$, некоторому очень маленькому числу. Мы обязаны считать эту вероятность ненулевой, иначе не сможем приписать ненулевое правдоподобие никакому множеству сообществ, в которых нет общих членов. Но поскольку она очень мала, мы предпочитаем решения, в которых каждое наблюдаемое ребро объясняется совместным членством в некотором сообществе.

Если мы знаем, какие вершины к каким сообществам относятся, то можем вычислить правдоподобие данного графа при заданных вероятностях ребер с помощью несложного обобщения примера 10.21. Обозначим $M_{uv}$ множество сообществ, в которые одновременно входят $u$ и $v$. Тогда правдоподобие того, что $E$ является в точности множеством ребер наблюдаемого графа, равно

$$\prod_{(u,v) \in E} p_{uv} \quad \prod_{(u,v) \notin E} (1 - p_{uv})$$

**Пример 10.22.** Рассмотрим крохотный социальный граф на рис. 10.20. Предположим, что имеется два сообщества $C$ и $D$, с которыми ассоциированы вероятности $p_C$ и $p_D$. Предположим также, что мы определили (или выдвинули в качестве временной гипотезы), что $C = \{w, x, y\}$ и $D = \{w, y, z\}$. Для начала рассмотрим пару вершин $w$ и $x$. $M_{wx} = \{C\}$, т. е. эта пара входит в сообщество $C$, но не входит в сообщество $D$. Поэтому $p_{wx} = 1 - (1 - p_C) = p_C$.

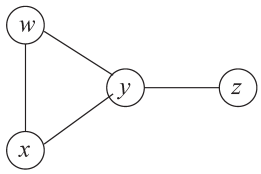



**Рис. 10.20.** Социальный граф

Аналогично $x$ и $y$ входят вместе только в $C$, а $y$ и $z$, равно как $w$ и $z$ – только в $D$. Поэтому $p_{xy} = p_C$ и $p_{yz} = p_{wz} = p_D$. Но пара $w$ и $y$ входит вместе в оба сообщества, так что $p_{wy} = 1 - (1 - p_C)(1 - p_D) = p_C + p_D - p_C p_D$. Наконец, $x$ и $z$ не встречаются вместе ни в каком сообществе, поэтому $p_{xz} = \varepsilon$.

Теперь мы можем вычислить правдоподобие графа на рис. 10.20 при заданных предположениях относительно членства в двух сообществах. Оно равно произведению вероятностей, ассоциированных с каждой из четырех пар вершин, для которых соединяющие их ребра присутствуют в графе, на единицу минус вероятность каждой из двух пар, чьи ребра отсутствуют:

$$p_{wx} p_{wy} p_{xy} p_{yz} (1 - p_{wz})(1 - p_{xz}).$$

Подставляя полученные выше выражения для каждой вероятности, получаем

$$(p_C)^2 p_D (p_C + p_D - p_C p_D)(1 - p_D)(1 - \varepsilon).$$

Заметим, что $\varepsilon$ очень мало, поэтому последний сомножитель близок к 1, и его можно опустить. Мы должны найти такие значения $p_C$ и $p_D$, при которых достигается максимум этого выражения. Заметим, прежде всего, что все сомножители либо не зависят от $p_C$, либо растут вместе с $p_C$. Единственный нетривиальный момент в рассуждении – вспомнить, что $p_D \leq 1$, поэтому выражение

$$p_C + p_D - p_C p_D$$

должно монотонно возрастать при росте $p_C$. Отсюда следует, что правдоподобие достигает максимума при максимально большом $p_C$, т. е. при $p_C = 1$.

Далее мы должны найти значение $p_D$, доставляющее максимум этому выражению при условии, что $p_C = 1$. Тогда все выражение приобретает вид $p_D(1 - p_D)$, и, как легко видеть, его максимум достигается при $p_D = 0.5$. Таким образом, принимая во внимание, что $C = \{w, x, y\}$ и $D = \{w, y, z\}$, максимальное правдоподобие для графа на рис. 10.20 достигается, когда между членами $C$ наверняка имеются ребра и с вероятностью 50 % совместное членство в $D$ приводит к появлению ребра между членами.

Однако пример 10.22 – лишь часть решения. Нам еще нужно найти такое распределение членов по сообществам, при котором максимальное правдоподобие, найденное для этого распределения, оказывается наибольшим для любого распределения. Зафиксировав распределение, мы можем найти вероятности $p_C$, ассоциированные с каждым сообществом, даже для очень больших графов с большим количеством сообществ. Общий метод решения таких задач называет «градиентным

спуском», мы познакомились с ним в разделе 9.4.5 и продолжим его обсуждение в разделе 12.3.4.

К сожалению, не очевидно, как применить к множеству членов каждого сообщества метод градиентного спуска, потому что изменения в составе сообществ дискретны, а не описываются какой-то непрерывной функцией. Единственный практически осуществимый способ поиска в пространстве возможных распределений членов по сообществам – начать с некоторого распределения и производить небольшие изменения, скажем вставку или удаление одного члена. Для каждого такого распределения мы можем найти оптимальные вероятности $p_C$ методом градиентного спуска. Однако понять, какие изменения членства ведут в правильном направлении, нелегко, и нет гарантии, что мы вообще дойдем до оптимального распределения, постепенно изменяя начальное.

**Логарифмическое правдоподобие**

Обычно вычисляют логарифм функции правдоподобия (*логарифмическое правдоподобие*), а не саму функцию. Это удобнее по нескольким причинам. Произведения заменяются суммами, что часто упрощает выражения. Кроме того, суммирование большого количества чисел устойчивее к ошибкам округления, чем умножение очень малых чисел.

## 10.5.4. Как избежать дискретных изменений членства

Существует решение описанной в разделе 10.5.3 проблемы, связанной с дискретностью членства в сообществах: человек либо является членом сообщества, либо нет. Можно ввести понятие «силы членства». Интуитивно представляется, что чем сильнее членство двух лиц в одном сообществе, тем вероятнее, что это сообщество приведет к появлению ребра между ними. При такой модели силу членства лица в сообществе можно изменять непрерывно – точно так же, как ассоциированную с сообществом вероятность в модели графа принадлежности. Это позволяет применять стандартные методы, например градиентного спуска, для максимизации функции правдоподобия. В усовершенствованной модели мы имеем:

1. Фиксированные множества сообществ и лиц, как и раньше.
2. Для каждого сообщества $C$ и лица $x$ определен параметр силы членства $F_{xC}$. Эти параметры могут принимать произвольное неотрицательное значение, причем 0 означает, что лицо заведомо не входит в сообщество.
3. Вероятность того, что членство в сообществе $C$ станет причиной появления ребра между вершинами $u$ и $v$, задается функцией

$$p_C(u, v) = 1 - e^{-F_{uC}F_{vC}}$$

Как и раньше, вероятность существования ребра между вершинами $u$ и $v$ равна 1 минус вероятность того, что ни одно сообщество не привело к существованию ребра между ними. То есть каждое сообщество независимо влияет на наличие ребра, и ребро между двумя вершинами существует, если это вызвано хотя бы одним сообществом. Формально вероятность $p_{uv}$ существования ребра между ребрами $u$ и $v$ вычисляется по формуле

$$p_{uv} = 1 - \prod_C (1 - p_C(u,v)) .$$

Подставляя сюда выражение для $p_C(u, v)$, предполагаемое в модели, получаем

$$p_C(u, v) = 1 - e^{-\Sigma_C F_{uC} F_{vC}}.$$

Наконец, пусть $E$ – множество ребер в наблюдаемом графе. Как и раньше, можно выписать выражение для правдоподобия наблюдаемого графа в виде произведения $p_{uv}$ для каждого ребра $(u, v)$, встречающегося в $E$, умноженного на произведение $1 - p_{uv}$ для каждого ребра $(u, v)$, не встречающегося в $E$. Таким образом, в новой модели формула правдоподобия графа с ребрами $E$ выглядит так:

$$\prod_{(u,v)\in E} (1 - e^{-\sum_C F_{uC} F_{vC}}) \quad \prod_{(u,v)\notin E} e^{-\sum_C F_{uC} F_{vC}} .$$

Напомним, что максимизация функции равносильна максимизации ее логарифма. Поэтому выражение можно немного упростить, взяв натуральный логарифм для замены произведений суммами. Упрощение достигается еще и потому, что $\log(e^x) = x$.

$$\sum_{(u,v)\in E} \log(1 - e^{-\sum_C F_{uC} F_{vC}}) - \sum_{(u,v)\notin E} \sum_C F_{uC} F_{vC} \qquad (10.1)$$

Теперь можно найти значения $F_{xC}$, доставляющие максимум выражению (10.1). Один из вариантов – воспользоваться градиентным спуском, как в разделе 9.4.5. То есть выбрать одну вершину $x$ и корректировать все значения $F_{xC}$ в том направлении, в котором значение (10.1) становится лучше всего. Отметим, что на изменения $F_{xC}$ реагируют только сомножители, для которых одна из вершин $u$ и $v$ совпадает с $x$, а другая – с узлом, смежным с $x$. Степень узла обычно много меньше количества ребер в графе, поэтому на каждом шаге мы можем не рассматривать большинство членов в (10.1).

## 10.5.5. Упражнения к разделу 10.5

**Упражнение 10.5.1.** Пусть графы порождаются случайным образом, как в примере 10.21: каждое ребро существует с вероятностью $p$ независимо от остальных. При каком значении $p$ правдоподобие графа на рис. 10.20 максимально? С какой вероятностью будет порожден такой граф?

**Упражнение 10.5.2.** Вычислите ОМП для графа из примера 10.22 при следующих гипотезах о членстве в двух сообществах:

(*а*) $C = \{w, x\}$; $C = \{y, z\}$.

(*б*) $C = \{w, x, y, z\}$; $C = \{x, y, z\}$.

**Упражнение 10.5.3.** Пусть имеется монета, не обязательно правильная, и в результате нескольких бросаний выпало $h$ орлов и $t$ решек.

(*а*) Если вероятность выпадения орла равна $p$, то как в терминах $h$ и $t$ выражается ОМП $p$?

! (*б*) Пусть нам сообщили, что с вероятностью 90 % монета правильная (т. е. $p = 0.5$) и с вероятностью 10 % – $p = 0.1$. При каких значениях $h$ и $t$ более вероятно, что монета правильная?

!! (*в*) Предположим, что априорная вероятность конкретного значения $p$ пропорциональна $|p - 0.5|$. Это означает, что $p$ с большей вероятностью окажется близко к 0 или 1, чем к 1/2. Если наблюдается $h$ орлов и $t$ решек, то какова оценка максимального правдоподобия $p$?

# 10.6. Simrank

В этом разделе мы рассмотрим другой подход к анализу графов социальных сетей. Эта техника, именуемая «simrank», лучше всего подходит для графов с вершинами нескольких типов, хотя в принципе применима к любому графу. Цель состоит в том, чтобы измерить сходство между вершинами одного типа, а для этого мы смотрим, куда заведет случайное блуждание, начатое в произвольной вершине. Поскольку вычисление необходимо произвести для каждой вершины, размер графа, который можно до конца проанализировать этим методом, ограничен.

## 10.6.1. Случайные блуждания в социальном графе

Напомним, что в разделе 5.1 мы говорили, что PageRank отражает поведение пользователя, случайно блуждающего по графу веба. Точно так же можно представить себе «блуждание» в социальной сети. В общем случае граф социальной сети не ориентирован, тогда как граф веба ориентированный. Но это различие несущественно. Посетитель, находящийся в вершине $N$ неориентированного графа, с равной вероятностью переходит в любую *соседнюю* с $N$ вершину (соединенную с $N$ ребром).

Предположим, к примеру, что посетитель начал блуждание в вершине $T_1$ графа, изображенного на рис. 10.2 и повторенного на рис. 10.21. На первом шаге он перейдет в $U_1$ или $W_1$. Если в $W_1$, то дальше он перейдет либо об-

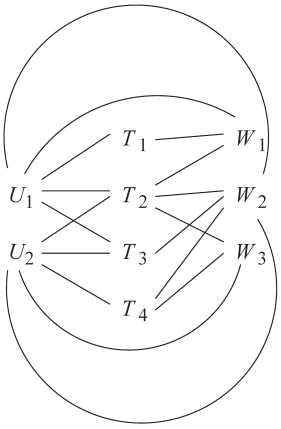



**Рис. 10.21.** Воспроизведение графа, изображенного на рис. 10.2

ратно в $T_1$, либо в $T_2$. Если же сначала посетитель перейдет в $U_1$, то затем он может оказаться в любой из вершин $T_1$, $T_2$ или $T_3$.

Мы делаем вывод, что, начав в $T_1$, *посетитель* имеет хорошие шансы посетить $T_2$, по крайней мере в начале, и что эти шансы выше, чем на посещение $T_3$ или $T_4$. Было бы интересно, если бы мы могли отсюда вывести, что метки $T_1$ и $T_2$ каким-то образом связаны или похожи. Факт тот, что обе они помечают одну и ту же веб-страницу $W_1$ и поставлены одним и тем же пользователем $U_1$.

Но если мы позволим посетителю и дальше случайно обходить граф, то вероятность попасть в любую конкретную вершину не зависит от начальной точки. Этот вывод следует из теории марковских процессов, упомянутой в разделе 5.1.2, хотя для удовлетворения условия независимости от начальной точки необходимы дополнительные условия, помимо связности. Впрочем, для графа на рис. 10.21 они выполняются.

## 10.6.2. Случайное блуждание с перезапуском

Из сказанного выше вытекает, что невозможно изменить сходство с конкретным узлом, изучая распределение вершин, посещенных в процессе случайного блуждания. Но в разделе 5.1.5 мы уже рассматривали включение небольшой вероятности остановки блуждания в случайной точке. Позже, в разделе 5.3.2 мы видели, что есть причины выбирать в качестве множества телепортации – страниц, на которые перемещается посетитель после случайной остановки блуждания, – только подмножество всех веб-страниц.

Здесь мы доведем эту идею до логического завершения. Поскольку нас интересует одна конкретная вершина $N$ графа социальной сети, и мы хотим знать, куда заведет случайного посетителя короткое блуждание из этой вершины, модифицируем матрицу вероятностей переходов, добавив небольшую вероятность перехода в $N$ из любой вершины. Формально обозначим $M$ *матрицу переходов* графа $G$. Элемент на пересечении строки $i$ и столбца $j$ матрицы $M$ равен $1/k$, если вершина $j$ имеет степень $k$ и является смежной с $i$. В противном случае этот элемент равен 0. Телепортацию мы обсудим чуть ниже, а пока рассмотрим простой пример матрицы переходов.

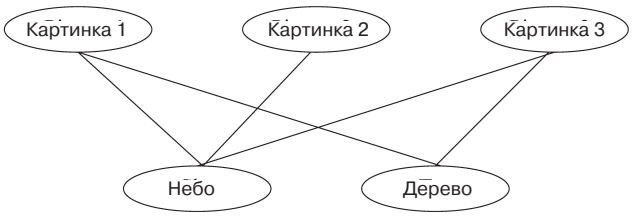



**Рис. 10.22.** Простой двудольный социальный граф

**Пример 10.23.** На рис. 10.22 приведен пример очень простой сети, содержащей три картинки и две ассоциированных с ними метки – «Небо» и «Дерево». Картинки 1 и 3 помечены обеими метками, а Картинка 2 – только

меткой «Дерево». Интуитивно кажется, что Картинка 3 больше похожа на Картинку 1, чем Картинка 2, и анализ с использованием случайного блуждания, перезапускаемого в Картинке 1, подкрепляет эту гипотезу.

Упорядочим вершины следующим образом: Картинка 1, Картинка 2, Картинка 3, Небо, Дерево. Тогда матрица переходов будет иметь вид:

$$\begin{bmatrix} 0 & 0 & 0 & 1/3 & 1/2 \\ 0 & 0 & 0 & 1/3 & 0 \\ 0 & 0 & 0 & 1/3 & 1/2 \\ 1/2 & 1 & 1/2 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 \end{bmatrix}.$$

Например, четвертый столбец соответствует вершине «Небо», и эта вершина соединена с каждой из трех вершин с картинками. Поэтому ее степень равна 3, так что все ненулевые элементы в ее столбце должны быть равны 1/3. Поскольку вершинам с картинками соответствуют первые три столбца и три строки, то элементы в первых трех строках четвертого столбца равны 1/3. Поскольку вершина «Небо» не соединена ребром ни сама с собой, ни с вершиной «Дерево», элементы в последних двух строках столбца 4 равны 0.

Как и раньше, обозначим $\beta$ вероятность того, что посетитель продолжает случайное блуждание, так что $1 - \beta$ – вероятность телепортации в начальную вершину $N$. Обозначим $\mathbf{e}_N$ – вектор-столбец, в котором элемент, соответствующий вершине $N$, равен 1, а все остальные 0. Тогда, если $\mathbf{v}$ – вектор-столбец, описывающий вероятность, что случайный посетитель окажется в каждой вершине на некотором шаге, а $\mathbf{v}'$ – вектор, описывающий вероятности оказаться в каждой вершине на следующем шаге, то $\mathbf{v}'$ связан с $\mathbf{v}$ таким соотношением:

$$\mathbf{v}' = \beta M\mathbf{v} + (1 - \beta)\mathbf{e}_N$$

**Пример 10.24.** Пусть $M$ – матрица из примера 10.23 и $\beta = 0.8$. Предположим также, что вершина $N$ – Картинка 1, т. е. мы хотим вычислить сходство Картинки 1 с другими картинками. Тогда рекуррентное уравнение для нахождения нового значения $\mathbf{v}'$ имеет вид:

$$v' = \begin{bmatrix} 0 & 0 & 0 & 4/15 & 2/5 \\ 0 & 0 & 0 & 4/15 & 0 \\ 0 & 0 & 0 & 4/15 & 2/5 \\ 2/5 & 4/5 & 2/5 & 0 & 0 \\ 2/5 & 0 & 2/5 & 0 & 0 \end{bmatrix} v + \begin{bmatrix} 1/5 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Поскольку граф на рис. 10.22 связный, исходная матрица $M$ стохастическая, и мы можем сделать вывод, что если сумма элементов начального вектора $\mathbf{v}$ равна 1, то и сумма элементов $\mathbf{v}'$ будет равна 1. Поэтому уравнение можно упростить, прибавив 1/5 к каждому элементу первой строки матрицы. Таким образом, итерация описывается умножением матрицы на вектор:

$$v' = \begin{bmatrix} 1/5 & 1/5 & 1/5 & 7/15 & 3/5 \\ 0 & 0 & 0 & 4/15 & 0 \\ 0 & 0 & 0 & 4/15 & 2/5 \\ 2/5 & 4/5 & 2/5 & 0 & 0 \\ 2/5 & 0 & 2/5 & 0 & 0 \end{bmatrix} v .$$

Если начать с $\mathbf{v} = \mathbf{e}_N$, то последовательность оценок распределения случайного блуждания будет такой:

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1/5 \\ 0 \\ 0 \\ 2/5 \\ 2/5 \end{bmatrix}, \begin{bmatrix} 35/75 \\ 8/75 \\ 20/75 \\ 6/75 \\ 6/75 \end{bmatrix}, \begin{bmatrix} 95/375 \\ 8/375 \\ 20/375 \\ 142/375 \\ 110/375 \end{bmatrix}, \begin{bmatrix} 2353/5625 \\ 568/5625 \\ 1228/5625 \\ 786/5625 \\ 690/5625 \end{bmatrix}, \cdots, \begin{bmatrix} .345 \\ .066 \\ .145 \\ .249 \\ .196 \end{bmatrix}.$$

Мы замечаем, что в пределе вероятность оказаться в Картинке 3 более чем в два раза превышает вероятность оказаться в Картинке 2. Следовательно, подтверждается наше интуитивное предположение о том, что Картинка 3 больше похожа на Картинку 1, чем Картинка 2.

Из примера 10.24 можно сделать еще несколько выводов. Во-первых, отметим, что проведенный анализ относится только в Картинке 1. Если бы мы захотели узнать, какие картинки больше всего похожи на какую-нибудь другую, то должны были бы заново провести анализ для другой картинки. Аналогично, если бы мы захотели узнать, какие метки теснее всего связаны с меткой «Небо» (в таком маленьком примере, где всего две метки, этот вопрос не представляет интереса), то должны были бы организовать телепортацию только в вершину «Небо».

Во-вторых, сходимость занимает время из-за начальных колебаний. То есть поначалу весь вес сосредоточен в картинках, а на втором шаге большая часть веса переходит к меткам. На третьем шаге значительная часть веса возвращается к картинкам, а на четвертом вновь переходит к меткам. Однако в пределе процесс все же сходится, так что 5/9 веса приходится на картинки, а 4/9 – на метки. В общем случае процесс сходится для любого связного $k$-дольного графа.

## 10.6.3. Упражнения к разделу 10.6

**Упражнение 10.6.1.** Если на рис. 10.22 начать блуждание с Картинки 2, то каким окажется сходство этой картинки с двумя другими? Какая картинка, на ваш взгляд, будет больше похожа на Картинку 2?

**Упражнение 10.6.2.** Если на рис. 10.22 начать блуждание с Картинки 3, то какого сходства этой картинки с двумя другими можно ожидать?

**! Упражнение 10.6.3.** Повторите анализ, проведенный в примере 10.24, и вычислите сходство Картинки 1 с другими, подвергнув граф на рис. 10.22 следующим модификациям:

(*а*) добавить метку «Дерево» к Картинке 2;
(*б*) добавить к Картинке 3 третью метку «Вода»;
(*в*) добавить третью метку «Вода» к Картинкам 1 и 2.
*Примечание*: изменения в каждом пункте независимы, они не объединяются.

# 10.7. Подсчет треугольников

Одна из самых полезных операций над графами социальных сетей – подсчет треугольников и других простых подграфов. В этом разделе мы опишем методы получения оценки или точного числа треугольников в очень большом графе. Начнем с объяснения, зачем такой подсчет нужен, а затем познакомимся с методами эффективного решения этой задачи.

## 10.7.1. Зачем подсчитывать треугольники?

Если начать с $n$ вершин и случайным образом добавить $m$ ребер, то в графе образуется какое-то число треугольников. Подсчитать его математическое ожидание не составляет труда. Существует $\binom{n}{3}$ сочетаний трех узлов, т. е приблизительно $n^3/6$ троек узлов, которые могли бы составить треугольник. Вероятность существования ребра между любыми двумя заданными вершинами равна $m/\binom{n}{2}$, или приблизительно $2m/n^2$. Вероятность, что в произвольно взятой тройке узлов каждый два будут соединены ребрам, если ребра добавлялись независимо, приближенно равна $(2m/n^2)^3 = 8m^3/n^6$. Таким образом, ожидаемое число треугольников в графе с $n$ вершинами и $m$ случайно выбранными ребрами приближенно равно $(8m^3/n^6)(n^3/6) = 4/3\ (m/n)^3$.

В графе социальной сети с $n$ участниками и $m$ парами «друзей» ожидаемое количество треугольников должно быть значительно больше, чем в случайном графе. Дело в том, что если $A$ и $B$ – друзья, и $A$ при этом является еще и другом $C$, то вероятность, что $B$ и $C$ – также друзья, гораздо выше средней. Следовательно, подсчет треугольников позволит измерить похожесть заданного графа на граф социальной сети.

Мы можем также проанализировать сообщества в социальной сети. Было показано, что время существования сообщества связано с плотностью треугольников. То есть, когда группа только формируется, люди приглашают в нее своих единомышленников, но количество треугольников сравнительно мало. Если $A$ приглашает своих друзей $B$ и $C$, вполне может статься, что $B$ и $C$ незнакомы. По мере развития сообщества $B$ и $C$ могут познакомиться в силу членства в одном сообществе. Поэтому велики шансы, что со временем треугольник $\{A,B,C\}$ будет достроен.

## 10.7.2. Алгоритм нахождения треугольников

Начнем наше исследование с алгоритма, имеющего минимально возможное время работы на одном процессоре. Пусть имеется граф с $n$ вершинами и $m \geq n$ ребрами. Для удобства предположим, что вершинами являются целые числа 1, 2, ..., $n$.

Назовем вершину *влиятельной*, если ее степень не меньше $\sqrt{m}$. *Влиятельным треугольником* называется треугольник, все вершины которого влиятельные. Для

подсчета влиятельных и прочих треугольников применяются разные алгоритмы. Заметим, что число влиятельных вершин не превышает $2\sqrt{m}$, потому что иначе сумма степеней влиятельных вершин была бы больше $2m$. Поскольку каждое ребро дает вклад только в степени двух вершин, то в этом случае число ребер превышало бы $m$.

Предполагая, что граф представлен своими ребрами, выполним его предварительную обработку.

1. Вычислить степени всех вершин. Для этого требуется только перебрать все ребра и прибавить 1 к степеням обоих концов. Общее время $O(m)$.
2. Создать индекс ребер, в котором ключом является пара вершин ребра. Такой индекс позволяет для каждой пары вершин узнать, существует ли ребро между ними. Для этой цели вполне достаточно хэш-таблицы, которую можно построить за время $O(m)$, а ожидаемое время ответа на запрос о существовании ребра постоянно, по крайней мере, в типичном случае[2].
3. Создать еще один индекс ребер, ключом которого является одиночная вершина. С его помощью мы сможем, зная вершину $v$, найти все смежные с ней вершины за время, пропорциональное числу таких вершин. И снова достаточно хэш-таблицы с постоянным временем ответа в типичном случае.

Введем на множестве вершин отношение порядка. Сначала упорядочим вершины по степени. Если степени $v$ и $u$ одинаковы, то, пользуясь тем, что $v$ и $u$ – целые числа, упорядочим их по числовому значению. Таким образом, мы говорим, что $v \prec u$, если:

($i$) степень $v$ меньше степени $u$ или
($ii$)степени $u$ и $v$ одинаковы и $v < u$.

*Влиятельные треугольники.* Существует всего $O(\sqrt{m})$ влиятельных вершин, поэтому мы можем рассмотреть все тройки таких вершин. Потенциально может существовать $O(m^{3/2})$ влиятельных треугольников, и с помощью индекса над ребрами мы можем проверить наличие всех трех ребер за время $O(1)$. Поэтому для нахождения всех влиятельных треугольников понадобится время $O(m^{3/2})$.

*Прочие треугольники.* Прочие треугольники мы будем искать по-другому. Рассматриваем каждое ребро $(v_1, v_2)$. Если и $v_1$, и $v_2$ – влиятельные вершины, пропускаем это ребро. Пусть теперь вершина $v_1$ не влиятельная и $v_1 \prec v_2$. Пусть $u_1, u_2, ..., u_k$ – вершины, смежные с $v_1$. Заметим, что $k < \sqrt{m}$. Эти вершины можно найти, пользуясь индексом по вершинам за время $O(k)$, заведомо не большее $O(\sqrt{m})$. Для каждой вершины $u_i$ можно воспользоваться первым индексом и за время $O(1)$ проверить существование ребра $(u_i, v_1)$. Мы также можем за время $O(1)$ найти степень вершины $u_i$, потому что заранее подсчитали степени всех вершин. Треугольник $\{v_1, v_2, u_i\}$ учитывается тогда и только тогда, когда существует ребро $(u_i, v_2)$ и $v_1 \prec u_i$. При таком подсчете каждый треугольник учитывается только один раз – когда $v_1$

[2] Строго говоря, наш алгоритм оптимален только в смысле ожидаемого времени работы, а не времени работы в худшем случае. Но при хэшировании большого числа объектов вероятность ожидаемого поведения очень высока, а если мы все-таки выбрали хэш-функцию, при которой некоторые ячейки оказались слишком велики, то можно повторить хэширование с другой функцией, пока не найдется подходящая.

является вершиной, предшествующей двум другим его вершинам относительно порядка $\prec$. Следовательно, время обработки всех вершин, смежных с $v_1$, имеет порядок $O(\sqrt{m})$. Поскольку всего существует $m$ ребер, общее время, потраченное на подсчет прочих треугольников, равно $O(m^{3/2})$.

Итак, предварительная обработка занимает время $O(m)$, а время нахождения влиятельных треугольников, как, впрочем, и всех остальных, – $O(m^{3/2})$. Следовательно, общее время работы алгоритма составляется $O(m^{3/2})$.

## 10.7.3. Оптимальность алгоритма нахождения треугольников

Оказывается, что алгоритм, описанный в разделе 10.7.2, по порядку величины является наилучшим из возможных. Чтобы понять, почему, рассмотрим полный граф с $n$ вершинами. В этом графе $m = \binom{n}{2}$ ребер и $\binom{n}{3}$ треугольников. Поскольку невозможно пересчитать треугольники за время, меньшее их количества, на таком графе любой алгоритм подсчет требует времени $\Omega(n^3)$. Но поскольку $m = O(n^2)$, время работы любого алгоритма для этого графа составляет $O(m^{3/2})$.

Но, быть может, существует более быстрый алгоритм для не столь плотных графов? Однако мы можем добавить к полному графу цепочку вершин произвольной длины не более $n^2$. При этом новых треугольников не появится. Количество ребер увеличится не более чем вдвое, а количество вершин можно сделать каким угодно, следовательно, мы можем сделать отношение числа ребер к числу вершин сколь угодно близким к 1. Поскольку треугольников по-прежнему $\Omega(m^{3/2})$, получается, что нижняя граница остается в силе при любом возможном отношении $m/n$.

## 10.7.4. Нахождение треугольников с помощью MapReduce

Если граф очень велик, то хотелось бы ускорить вычисления за счет распараллеливания. Задачу нахождения треугольников можно сформулировать как многопутевое соединение и воспользоваться описанной в разделе 2.5.3 техникой для оптимизации их подсчета с помощью одного задания. Оказывается, что это пример ситуации, когда многопутевое соединение в общем случае гораздо эффективнее двух двухпутевых. Более того, общее время выполнения параллельного алгоритма по существу такое же, как время выполнения алгоритма из раздела 10.7.2 на одном процессоре.

Для начала предположим, что вершины графа пронумерованы числами 1, 2, ..., $n$. Для представления ребер воспользуемся отношением $E$. Чтобы не учитывать одно ребро дважды, предположим, что если $E(A,B)$ – кортеж этого отношения, то не только существует ребро между $A$ и $B$, и но и для соответствующих целочисленных номеров вершин справедливо неравенство $A < B$.[3] При таком тре-

[3] Не путайте это простое числовое упорядочение с отношением порядка $\prec$ из раздела 10.7.2, которое подразумевает также сравнение степеней вершин. Здесь степени вершин не вычисляются и несущественны.

бовании также устраняются петли (ребра, начинающиеся и заканчивающиеся в одной и той же вершине), которые вообще-то в графах социальных сетей и так отсутствуют, но могли бы стать причиной появления «треугольников», имеющих менее трех вершин.

Пользуясь этим отношением, мы можем описать множество треугольников в графе с множеством ребер $E$ с помощью естественного соединения

$$E(X,Y) \bowtie E(X,Z) \bowtie E(Y,Z) \qquad (10.2)$$

Чтобы разобраться в этом соединении, нужно понять, что атрибутам отношения $E$ даются различные имена в каждом из трех случаев использования $E$. То есть мы должны мысленно представить, что есть три копии $E$ с одними и теми же кортежами, но разными схемами. На языке SQL это соединение записывалось бы с использованием единственного отношения $E(A,B)$ следующим образом:

```
SELECT e1.A, e1.B, e2.B
FROM E e1, E e2, E e3
WHERE e1.A = e2.A AND e1.B = e3.A AND e2.B = e3.B
```

В этом запросе приравненные атрибуты $e1.A$ и $e2.A$ представлены атрибутом $X$ в соединении (10.2). Атрибуты $e1.B$ и $e3.A$ представлены атрибутом $Y$, а $e2.B$ и $e3.B$ – атрибутом $Z$.

Отметим, что каждый треугольник встречается в этом соединении ровно один раз. Треугольник, состоящий из вершин $v_1$, $v_2$ и $v_3$, порождается, когда $X$, $Y$ и $Z$ – вершины с числовым порядком $X < Y < Z$. Например, если в числовом порядке $v1 < v2 < v3$, то в качестве $X$ может выступать только $v_1$, в качестве $Y$ – только $v_2$, а в качестве $Z$ – только $v_3$.

Метод из раздела 2.5.3 можно применить для оптимизации соединения (10.2). Напомним идеи из примера 2.9, где мы рассматривали, сколькими способами следует хэшировать значение каждого атрибута. В данном случае этот вопрос решается очень просто. Все три вхождения отношения $E$, очевидно, одного размера, поэтому из соображений симметрии $X$, $Y$ и $Z$ будут хэшироваться в одно и то же число ячеек. В частности, если вершины хэшируются в $b$ ячеек, то будет $b^3$ операций редукции. Каждая задача-редуктор ассоциирована с последовательностью из трех номеров ячеек $(x, y, z)$, где $x$, $y$ и $z$ принадлежат диапазону от 1 до $b$.

Задачи-распределители разбивают отношение $E$ на столько частей, сколько имеется задач. Пусть на вход некоторому распределителю поступает кортеж $E(u, v)$ для отправки редукторам. Будем рассматривать $(u, v)$ как кортеж члена соединения $E(X, Y)$. С помощью хэширования $u$ и $v$ мы можем получить номера ячеек для $X$ и $Y$, но не знаем, в какую ячейку хэшируется $Z$. Следовательно, мы должны отправить $E(u, v)$ всем редукторам, которые соответствуют последовательности трех номеров ячеек $(h(u), h(v), z)$ для любой из $b$ возможных ячеек $z$.

Но тот же самый кортеж $E(u, v)$ необходимо рассмотреть и как кортеж для члена $E(X, Z)$. Поэтому мы посылаем $E(u, v)$ также всем редукторам, которые соответствуют тройке $(h(u), y, h(v))$ для любого $y$. Наконец, рассматриваем $E(u, v)$ как кортеж члена $E(Y, Z)$ и отправляем его всем редукторам, которые соответствуют

тройке $(x, h(u), h(v))$ для любого $x$. Таким образом, общий объем коммуникаций составляет $3b$ пар ключ-значение для каждого из $m$ кортежей реберного отношения $E$. Поэтому при использовании $b^3$ редукторов минимальная коммуникационная стоимость равна $O(mb)$.

Далее вычислим общую стоимость выполнения всеми редукторами. Предположим, что хэш-функция распределяет ребра достаточно случайно, так что все редукторы получают примерно одинаковое количество ребер. Поскольку общее число ребер, распределенных $b^3$ редукторам, составляет $O(mb)$, каждый получит $O(m/b^2)$ ребер. Если в каждом редукторе использовать алгоритм из раздела 10.7.2, то коммуникационная стоимость для него составит $O((m/b^2)^{3/2})$, или $O(m^{3/2}/b^3)$. Поскольку всего редукторов $b^3$, полная коммуникационная стоимость равна $O(m^{3/2})$, т. е. ровно столько, сколько для однопроцессорного алгоритма из раздела 10.7.2.

## 10.7.5. Использование меньшего числа редукторов

Посредством продуманного упорядочения вершин мы можем уменьшить количество редукторов примерно в 6 раз. Будем считать «именем» вершины $i$ пару $(h(i), i)$, где $h$ – хэш-функция, которую мы использовали в разделе 10.7.4 для хэширования вершин по $b$ ячейкам. Упорядочим вершины по имени, рассматривая только первую компоненту (т. е. номер ячейки, в которую хэшируется вершина) и используя вторую лишь в том случае, когда две вершины хэшируются в одну и ту же ячейку.

При таком упорядочении вершин редуктор, соответствующий списку ячеек $(i, j, k)$, понадобится, только если $i \le j \le k$. При большом $b$ этим неравенствам удовлетворяет приблизительно $1/6$ всех $b^3$ троек целых чисел в диапазоне от 1 до $b$. Для любого $b$ количество таких троек равно $\binom{b+2}{3}$ (см. упражнение 10.7.4). Следовательно, точное отношение равно $(b + 2)(b + 1)/(6b^2)$.

Благодаря уменьшению числа редукторов значительно уменьшается количество пар ключ-значение, подлежащих передаче по сети. Каждое из $m$ ребер теперь нужно отправить не $3b$, а только $b$ редукторам. Точнее, рассмотрим ребро $e$, концы которого были хэшированы в ячейки $i$ и $j$, одинаковые или различные. Для каждого из $b$ значений $k$ между 1 и $b$ рассмотрим тройку, образованную числами $i, j, k$ в указанном выше порядке сортировки. Тогда ребро $e$ необходимо редуктору, который соответствует этой тройке, и никакому другому.

Для сравнения коммуникационной стоимости метода из этого раздела с методом из раздела 10.7.4 зафиксируем количество редукторов $k$. Тогда метод из раздела 10.7.4 хэширует вершины в $\sqrt[3]{k}$ ячеек и, значит, будет передавать $3m\sqrt[3]{k}$ пар ключ-значение. С другой стороны, метод из этого раздела хэширует вершины приблизительно в $\sqrt[3]{6k}$ ячеек, т. е. требует $m\sqrt[3]{6}\sqrt[3]{k}$ передач по сети. Следовательно, отношение коммуникационной стоимости двух методов составляет $3/\sqrt[3]{6} = 1.65$.

**Пример 10.25.** Рассмотрим прямолинейный алгоритм из раздела 10.7.4 при $b = 6$. В нем участвует $b^3 = 216$ редукторов, а коммуникационная стоимость равна $3mb = 18m$. Метод, описанный в этом разделе, не позволяет использовать ровно 216 редукторов, но можно подойти очень близко к этому значению, если положить $b = 10$. В таком случае число редукторов составит $\binom{12}{3} = 220$, а коммуникационная стоимость будет равна $mb = 10m$, т. е. 5/9 от стоимости прямолинейного метода.

## 10.7.6. Упражнения к разделу 10.7

**Упражнение 10.7.1.** Подсчитайте количество треугольников в следующих графах:

(*а*) на рис. 10.1;
(*б*) на рис. 10.9;
! (*в*) на рис. 10.2.

**Упражнение 10.7.2.** Для каждого графа из упражнения 10.7.1 определите:

(*i*) при какой минимальной степени вершину следует считать влиятельной;
(*ii*) какие вершины являются влиятельными;
(*iii*) какие треугольники являются влиятельными.

! **Упражнение 10.7.3.** В этом упражнении рассматривается задача о нахождении квадратов в графе. Мы хотим найти четверки вершин $a, b, c, d$ такие, что в графе существуют все четыре ребра $(a, b)$, $(b, c)$, $(c, d)$, $(a, d)$. Предположим, что граф представлен отношением $E$, как в разделе 10.7.4. Невозможно написать единственное соединение четырех экземпляров $E$, которое описывало бы все возможные квадраты в графе, но можно написать три таких соединения. Кроме того, в некоторых случаях соединение должно сопровождаться последующей выборкой, чтобы исключить «квадраты», в которых противоположные углы соответствуют одной и той же вершине. Мы можем предполагать, что вершина $a$ в числовом порядке меньше своих соседей $b$ и $d$, но для $c$ нужно рассмотреть три случая:

(*i*) $c$ меньше $b$ и $d$;
(*ii*) $c$ находится между $b$ и $d$;
(*iii*) $c$ больше $b$ и $d$.

(*а*) Выпишите естественные соединения, порождающие квадраты, которые удовлетворяют каждому из трех перечисленных условий. Разрешается использовать четыре различных атрибута $W, X, Y$ и $Z$ и предполагать, что имеется четыре копии отношения $E$ с разными схемами, так что все соединения можно выразить как естественные.

(*б*) Для каких соединений необходима последующая выборка, гарантирующая, что противоположные углы действительно представляют разные вершины?

!! (*в*) Допустим, что мы планируем использовать $k$ редукторов. Для каждого соединения из пункта (*а*) ответьте, в какое число ячеек нужно будет хэширо-

вать каждый атрибут $W$, $X$, $Y$, $Z$, чтобы минимизировать коммуникационную стоимость?

(*г*) В отличие от треугольников не гарантируется, что каждый квадрат порождается ровно один раз, хотя есть гарантия, что каждый квадрат порождается только одним из трех соединений. Например, квадрат, две противоположные вершины которого численно меньше обеих оставшихся вершин, порождается только соединением (*i*). Для каждого из трех соединений ответьте, сколько раз оно порождает свои квадраты.

! **Упражнение 10.7.4.** Покажите, что число последовательностей целых чисел, удовлетворяющих условиям $1 \le i \le j \le k \le b$, равно $\binom{b+2}{3}$. *Подсказка:* покажите, что можно установить взаимно однозначное соответствие между такими последовательностями и двоичными строками длины $b+2$, содержащими ровно три единицы.

# 10.8. Окрестности в графах

Некоторые важные свойства графов связаны с количеством вершин, достижимых из данной вершины по короткому пути. В этом разделе мы рассмотрим алгоритмы решения задач о путях и окрестностях в очень больших графах. В некоторых случаях точное решение невозможно получить для графов с миллионами вершин. Поэтому нас будут интересовать не только точные, но и приближенные алгоритмы.

## 10.8.1. Ориентированные графы и окрестности

В этом разделе мы будем моделировать сеть ориентированным графом. *Ориентированным графом*, или *орграфом* называется множество вершин и множество направленных ребер, или *дуг*; дугой называется упорядоченная пара вершин, записываемая в виде $u \to v$. Будем называть $u$ *начальной вершиной*, а $v$ – *конечной вершиной* дуги. Говорят, что дуга *направлена от u к v*.

Многие задачи моделируются орграфами. Главным примером служит веб, где дугой $u \to v$ представляется ссылка со страницы $u$ на страницу $v$. Также дуга $u \to v$ может означать, что абонент телефонной сети $u$ звонил абоненту $v$ в течение прошедшего месяца. Другой пример: дуга может означать, что пользователь Twitter $u$ читает сообщения пользователя $v$. Или же что научная статья $u$ ссылается на статью $v$.

Любой неориентированный граф можно представить в виде ориентированного, если вместо ребра $(u, v)$ использовать две дуги $u \to v$ и $v \to u$. Поэтому материал этого раздела относится также к графам, которые по сути своей не ориентированы, например, к графу друзей в социальной сети.

*Путем* в орграфе называется такая последовательность вершин $v_0, v_1, ..., v_k$, что существуют дуги $v_i \to v_{i+1}$ для всех $i = 0, 1, ..., k - 1$. *Длиной* этого пути называется число составляющих его дуг $k$. Отметим, что число дуг в пути длины $k$ равно $k + 1$, а отдельная вершина считается путем длины 0.

*Окрестностью радиуса d* вершины *v* называется множество вершин *u* таких, что существует путь из *v* в *u* длины не больше *d*. Будем обозначать такой путь $N(v, d)$. Например, $N(v, 0)$ всегда равно $\{v\}$, а $N(v, 1)$ – это *v* плюс множество вершин, для которых существует дуга, начинающаяся в *v*. В общем случае, если *V* – некоторое множество вершин, то $N(V, d)$ – множество таких вершин *u*, что существует путь длины *d* или меньше хотя бы из одной вершины, входящей в *V*.

*Профилем окрестностей* вершины *v* называется последовательность размеров ее окрестностей $|N(v, 1)|, |N(v, 2)|, \ldots$ . Мы не включаем окрестность радиуса 0, потому что ее размер всегда равен 1.

**Пример 10.26.** Рассмотрим неориентированный граф на рис. 10.1, который для удобства воспроизведен на рис. 10.23. Чтобы превратить его в ориентированный, будем рассматривать каждое ребро как пару дуг, по одной в каждом направлении. Например, ребро (*A*,*B*) преобразуется в дуги $A \rightarrow B$ и $B \rightarrow A$. Сначала рассмотрим окрестности вершины *A*. Мы знаем, что $N(A, 0) = \{A\}$. Далее, $N(A, 1) = \{A,B,C\}$, поскольку существуют дуги только из *A* в *B* и *C*. Аналогично $N(A, 2) = \{A,B,C,D\}$ и $N(A, 3) = \{A,B,C,D,E,F,G\}$. Все окрестности большего радиуса совпадают с $N(A, 3)$.

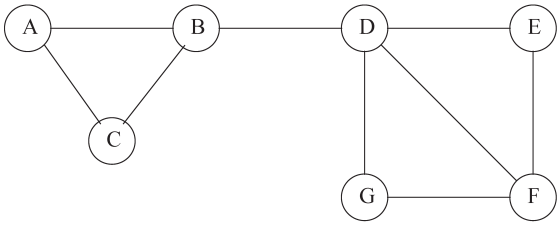



**Рис. 10.23.** Наша миниатюрная социальная сеть. Представляйте ее как ориентированный граф

С другой стороны, рассмотрим вершину *B*. Для нее $N(B, 0) = \{B\}$, $N(B, 1) = \{A,B,C,D\}$, $N(B, 2) = \{A,B,C,D,E,F,G\}$. Мы знаем, что *B* более центральна, чем *A*, и этот факт отражен в профилях окрестностей обеих вершин. Для вершины *A* профиль имеет вид 3, 4, 7, 7, ..., а для *B* – 4, 7, 7, ... . Очевидно, *B* более центральна, чем *A*, потому что на любом расстоянии размер ее окрестности не меньше, чем для *A*. На самом деле, *D* даже более центральна, чем *B*, потому что профиль ее окрестностей – 5, 7, 7, ... доминирует над профилем любой другой вершины.

## 10.8.2. Диаметр графа

*Диаметром* орграфа называется наименьшее целое число *d* такое, что для любых двух вершин *u* и *v* существует путь длины не больше *d* из *u* в *v*. Это определение имеет смысл, только если орграф сильно связный, т. е. существует путь из любой вершины в любую другую. Напомним (см. раздел 5.1.3), что в графе веба существует сильно связное подмножество в «центре», но веб в целом связным не явля-

ется. В нем существуют страницы, с которых не ведет ни одна ссылка, и страницы, до которых невозможно добраться, следуя по ссылкам.

Для неориентированного графа определение диаметра такое же, как для ориентированного, но путь может проходить по ребрам в любом направлении. То есть неориентированное ребро рассматривается как пара дуг, по одной в каждом направлении. И в этом случае понятие диаметра имеет смысл только для связных графов.

> **Пример 10.27.** Диаметр графа на рис. 10.23 равен 3. Существуют пары узлов, например *A* и *E*, для которых нет ни одного пути длины меньше 3. Но для любой пары узлов имеется соединяющий путь длины не больше 3.

Вычислить диаметр графа можно, вычисляя размеры его окрестностей возрастающего радиуса до тех пор, пока для некоторого радиуса не окажется больше ни одной новой вершины. То есть для каждой вершины $v$ находим наименьшее $d$ такое, что $|N(v, d)| = |N(v, d + 1)|$. Это значение $d$ является точной верхней гранью длины кратчайшего пути из $v$ в любую другую достижимую вершину. Назовем его $d(v)$. Так, в примере 10.26 мы видели, что $d(A) = 3$ и $d(B) = 2$. Если существует такая вершина $v$, что $|N(v, d(v))|$ не равно числу вершин во всем графе, то граф не является сильно связным, и никакое конечное целое число не может быть его диаметром. Но если граф сильно связный, что его диаметр равен $\max_v(d(v))$.

Это рассуждение справедливо, потому что $N(v, d+1)$ можно представить как объединение $N(v, d)$ и множества всех вершин $w$ таких, что для некоторой вершины $u$ из $N(v, d)$ существует дуга $u \rightarrow w$. То есть нужно начать с окрестности $N(v, d)$ и добавлять в нее конечные вершины всех дуг с начальной вершиной в $N(v, d)$. Если все дуги, начинающиеся в $N(v, d)$, уже принадлежат $N(v, d)$, то не только $N(v, d + 1)$, но и все последующие окрестности $N(v, d + 2)$, $N(v, d + 3)$, ... совпадают с $N(v, d)$. Наконец, заметим, что поскольку $N(v, d) \subseteq N(v, d+1)$, размер $|N(v, d)|$ может оказаться равным $|N(v, d+1)|$, только если $N(v, d)$ и $N(v, d + 1)$ – одно и то же множество. Таким образом, если $d$ – наименьшее целое число такое, что $|N(v, d)| = |N(v, d + 1)|$, то каждая вершина $v$ достижима с помощью пути длины, не большей $d$.

### Шесть шагов до знакомства

Существует знаменитая игра «шесть шагов до Кевина Бэйкона». Ее цель – найти пути длины не более 6 в графе, вершинами которого являются кинозвезды, а ребрами соединены актеры, игравшие в одном фильме. Утверждается, что в этом графе все актеры находятся от Кевина Бэйкона на расстоянии, не большем шести, т. е. его диаметр равен 6. При небольшом диаметре вычисление окрестностей более эффективно, поэтому было бы хорошо, если бы у всех графов социальных сетей диаметр был невелик. На самом деле, фраза «шесть шагов до знакомства» означает, что диаметр сети, охватывающей всех людей в мире, где наличие ребра означает знакомство, равен шести. К сожалению, как мы выясним в разделе 10.8.3, не все важные графы обладают столь тесными связями.

## 10.8.3. Транзитивное замыкание и достижимость

*Транзитивным замыканием* графа называется множество пар вершин $(u, v)$ таких, что существует путь из $u$ в $v$ длины 0 или более. Иногда мы будем записывать это утверждение в виде $Path(u, v)$[4]. С этим понятием тесно связано понятие *достижимости*. Говорят, что вершина $v$ достижима из $u$, если $Path(u, v)$ истинно. Задача о вычислении транзитивного замыкания состоит в том, чтобы найти в графе все пары вершин $u$ и $v$, для которых $Path(u, v)$ истинно. Задача о достижимости заключается в том, чтобы для данной вершины $u$ найти все такие вершины $v$, что $Path(u, v)$ истинно.

Оба эти понятия связаны с введенным выше понятием окрестности. На самом деле, $Path(u, v)$ истинно тогда и только тогда, когда $v$ принадлежит окрестности $N(u, \infty)$, определенной как $\cup_{i \geq 0} N(u, i)$. Таким образом, задача о достижимости эквивалентна вычислению объединения всех окрестностей заданной вершины $u$. В разделе 10.8.2 было показано, что для вычисления достижимого множества вершины $u$ нужно вычислить ее окрестности вплоть до наименьшего радиуса $d$, при котором $N(u, d) = N(u, d + 1)$.

Обе задачи – о транзитивном замыкании и о достижимости – связаны, но существует много примеров графов, для которых задача о достижимости практически разрешима, а задача о транзитивном замыкании – нет. Взять, к примеру, граф веба, насчитывающий миллиард вершин. Найти страницы (вершины), достижимые с заданной страницы, мы сможем даже на одной машине с большим объемом оперативной памяти. Но в транзитивном замыкании этого графа может быть до $10^{18}$ пар вершин, а это слишком много даже для большого вычислительного кластера[5].

## 10.8.4. Вычисление транзитивного замыкания с помощью MapReduce

Если говорить о параллельной реализации, то задача о транзитивном замыкании легче поддается распараллеливанию, чем задача о достижимости. Если мы хотим вычислить $N(v, \infty)$, множество вершин, достижимых из $v$, не вычисляя все транзитивное замыкание, то единственный способ – вычислить последовательность окрестностей, а это, по сути дела, обход графа в ширину, начиная с вершины $v$. В терминах реляционной теории обозначим $Arc(X, Y)$ отношение, содержащее такие пары вершин $(x, y)$, что существует дуга $x \rightarrow y$. Мы хотим итеративно вычислить отношение $Reach(X)$ – множество вершин, достижимых из $v$. После $i$ раундов $Reach(X)$ будет содержать все вершины, принадлежащие $N(v, i)$.

Первоначально $Reach(X)$ содержит только $v$. Предположим, что после некоторого раунда MapReduce оно содержит все вершины, принадлежащие $N(v, i)$. Для

[4] Строго говоря, здесь определено рефлексивное и транзитивное замыкание графа, потому что $Path(v, v)$ всегда истинно, даже если не существует цикла, содержащего $v$.

[5] Хотя полностью вычислить транзитивное замыкание мы не можем, о структуре графа все же можно узнать довольно много при условии, что в нем есть большие сильно связные компоненты.

построения $N(v, i + 1)$ мы должны соединить *Reach* с отношением *Arc*, а затем спроецировать результат на вторую компоненту и выполнить объединение результата с предыдущим значением *Reach*. В обозначениях SQL эта операция записывается в виде

```
SELECT DISTINCT Arc.Y
FROM Reach, Arc
WHERE Arc.X = Reach.X;
```

Этот запрос описывает естественное соединение *Reach*(*X*) и *Arc*(*X, Y*), которое можно произвести с помощью MapReduce, как объяснялось в разделе 2.3.7. Затем нужно сгруппировать результат по *Y* и устранить дубликаты, для чего достаточно еще одного задания MapReduce (см. раздел 2.3.8).

Число раундов этого процесса зависит от того, насколько далеко от *v* находится самая далекая достижимая из *v* вершина. Диаметр многих графов социальных сетей мал, как было сказано во врезке «Шесть шагов до знакомства». А раз так, то параллельное вычисление достижимости с помощью технологии MapReduce или ей подобной практически осуществимо. Потребуется всего несколько раундов вычислений, а требования к памяти ограничены объемом, необходимым для представления графа.

Но существуют графы, для которых количество раундов оказывается серьезным препятствием. Например, показано, что в типичной части веба путь к большинству страниц, достижимых из данной, имеет длину от 10 до 15. Но существуют такие пары страницы, что вторая достижима из первой, но только по пути длиной порядка нескольких сотен. В частности, блоги иногда устроены так, что ответ достижим только через комментарий, к которому он относится. Длинные споры порождают длинные пути, которые никак невозможно «срезать». Другой пример: учебное пособие в вебе, состоящее из 50 глав, может быть структурировано так, что перейти на главу $i$ можно только со страницы главы $i - 1$.

Интересно, что параллельно вычислить транзитивное замыкание можно гораздо быстрее, чем строгую достижимость. Применив технику рекурсивного удвоения, мы можем удвоить длину известных путей в одном раунде. То есть для графа диаметром $d$ понадобится только $\log_2 d$, а не $d$ раундов. При $d = 6$ это различие несущественно, но если $d = 1000$, то $\log_2 d$ приблизительно равно 10, так что количество раундов уменьшается в сто раз. Проблема, как уже было сказано, заключается в том, что нам приходится вычислять гораздо больше фактов, чем при вычислении достижимости в том же графе, а, значит, при вычислении транзитивного замыкания требуется гораздо больше памяти. Иными словами, если требуется получить только множество *Reach*($v$), то можно вычислить транзитивное замыкание всего графа и затем отбросить все пары, кроме тех, в которых первым элементом является $v$. Но мы не можем ничего отбросить, пока все пары не будут вычислены. В ходе вычисления транзитивного замыкания приходится вычислять много фактов *Path*($x$, $y$), в которых ни $x$, ни $y$ не достижимы из $v$, а даже если и достижимы, нам не всегда нужно знать, что $y$ достижима из $x$.

Но и в предположении, что граф достаточно мал, так что есть возможность вычислить транзитивное замыкание целиком, все равно нужно тщательно продумать, как это сделать с помощью MapReduce или другого подхода к распараллеливанию. Простейший подход на основе рекурсивного удвоения состоит в том, чтобы в начале положить отношение *Path*(*X*, *Y*) равным отношению *Arc*(*X*, *Y*). Предположим, что после $i$ раундов *Path*(*X*, *Y*) содержит все такие пары $(x, y)$, что существует путь из $x$ в $y$ длины не более $2^i$. Тогда, если на следующем раунде мы соединим *Path* с собой, то найдем все такие пары $(x, y)$, что существует путь из $x$ в $y$ длины не более $2 \times 2^i = 2^{i+1}$. На языке SQL запрос, необходимый для рекурсивного удвоения, записывается в виде

```
SELECT DISTINCT p1.X, p2.Y
FROM Path p1, Path p2
WHERE p1.Y = p2.X;
```

После выполнения этого запроса мы найдем все пары вершин, соединенных путем длины от 2 до $2^{i+1}$, в предположении, что *Path* содержит пары вершин, соединенных путем длины от 1 до $2^i$. Если взять объединение результата этого запроса с самим отношением *Arc*, то мы получим все пути длины от 1 до $2^{i+1}$ и сможем использовать это объединение в качестве отношения *Path* на следующем раунде рекурсивного удвоения. Сам запрос можно реализовать двумя заданиями MapReduce: одно для вычисления соединения, другое для вычисления объединения и устранения дубликатов. Как уже было отмечено при обсуждении параллельного вычисления достижимости, достаточно методов, описанных в разделах 2.3.7 и 2.3.8. Способ вычисления объединения из раздела 2.3.6 не нуждается в отдельном задании MapReduce; его можно скомбинировать с устранением дубликатов.

Если диаметр графа равен $d$, то после $\log_2 d$ раундов описанного выше алгоритма отношение *Path* будет содержать все пары вершин $(x, y)$, соединенных путем длины не больше $d$, т. е. все пары, принадлежащие транзитивному замыканию. Если $d$ заранее неизвестен, то необходим дополнительный раунд для проверки того, что больше никаких пар не существует, но при больших $d$ этот процесс может потребовать гораздо меньше раундов, чем обход графа в ширину, применяемый при вычислении достижимости.

Однако в методе рекурсивного удвоения делается очень много лишней работы. Поясним эту мысль на примере.

**Пример 10.28.** Предположим, что кратчайший путь из $x_0$ в $x_{17}$ имеет длину 17, т. е. существует путь $x_0 \to x_1 \to \ldots \to x_{17}$. Истинность *Path*($x_0$, $x_{17}$) будет установлена на пятом раунде, когда *Path* содержит все пары вершин, соединенные путями длины не более 16. Один и тот же путь из $x_0$ в $x_{17}$ будет обнаружен 16 раз при соединении *Path* с собой. То есть мы можем взять факт *Path*($x_0$, $x_{16}$) и, скомбинировав его с *Path*($x_{16}$, $x_{17}$), получить факт *Path*($x_0$, $x_{17}$). Но для установления того же факта можно скомбинировать *Path*($x_0$, $x_{15}$) с *Path*($x_{15}$, $x_{17}$) и т. д.

### 10.8.5. Интеллектуальное транзитивное замыкание

Вариант рекурсивного удвоения, при котором каждый путь обнаруживается не более одного раза, называется *интеллектуальным* транзитивным замыканием (smart transitive closure). Всякий путь длины больше 1 можно разбить на *голову*, длина которой равна степени двойки, и *хвост*, длина которого не превышает длину головы.

> **Пример 10.29.** Путь длины 13 состоит из головы, содержащей первые 8 дуг, и хвоста, содержащего оставшиеся 5 дуг. Путь длины 2 состоит из головы длины 1 и хвоста длины 1. Отметим, что 1 – это степень двойки (с показателем степени 0), и что длина хвоста совпадает с длиной головы, когда длина пути уже является степенью двойки.

Для реализации интеллектуального транзитивного замыкания на SQL, введем отношение $Q(X, Y)$, которое после $i$-ого раунда должно содержать все пары вершин $(x, y)$ такие, что длина кратчайшего пути из $x$ в $y$ составляет в точности $2^i$. Кроме того, после $i$-ого раунда утверждение $Path(x, y)$ будет истинным, если длина кратчайшего из $x$ в $y$ не превышает $2^{i+1} - 1$. Заметим, что такая интерпретация *Path* несколько отличается от интерпретации *Path* в простом методе рекурсивного удвоения, описанном в разделе 10.8.4.

Первоначально $Q$ и *Path* являются копиями отношения *Arc*. Предположим, что после $i$-ого раунда содержимое $Q$ и *Path* такое, как описано в предыдущем абзаце. Заметим, что на раунде $i = 1$ начальные значения $Q$ и *Path* удовлетворяют условиям для $i = 0$. На $(i + 1)$-ом раунде мы выполняем следующие действия:

1. Вычислить новое значение $Q$, соединив его с собой таким SQL-запросом:

```
SELECT DISTINCT q1.X, q2.Y
FROM Q q1, Q q2
WHERE q1.Y = q2.X;
```

2. Вычесть *Path* из отношения $Q$, вычисленного на шаге 1. Отметим, что на шаге (1) будут найдены все пути длины $2^{i+1}$. Но для некоторых пар, соединенных такими путями, могут существовать и более короткие пути. В результате шага (2) в $Q$ остаются те и только те пары $(u, v)$, для которых длина кратчайшего пути из $u$ в $v$ составляет в точности $2^{i+1}$.
3. Соединить *Path* с новым значением $Q$, вычисленным на шаге (2), выполнив такой SQL-запрос:

```
SELECT DISTINCT Q.X, Path.Y
FROM Q, Path
WHERE Q.Y = Path.X
```

В начале раунда *Path* содержит все пары $(y, z)$ такие, что длина кратчайшего пути из $y$ в $z$ не превышает $2^{i+1} - 1$, а новое значение $Q$ содержит все

пары $(x, y)$ такие, что длина кратчайшего пути из $x$ в $y$ равна $2^{i+1}$. Тогда результатом выполнения этого запроса будет множество пар $(x, y)$ таких, что длина кратчайшего пути из $x$ в $y$ находится в диапазоне от $2^{i+1} + 1$ до $2^{i+2} - 1$ включительно.

4. Присвоить *Path* новое значение, равное объединению отношения, вычисленного на шаге (3), нового значения $Q$, вычисленного на шаге (1), и старого значения *Path*. Эти три члена дают все пары $(x, y)$, для которых длина кратчайшего пути находится в диапазоне от $2^{i+1} + 1$ до $2^{i+2} - 1$, в точности равна $2^{i+1}$ и находится в диапазоне от 1 до $2^{i+1} - 1$. Следовательно, объединение содержит все кратчайшие пути длины не более $2^{i+2} - 1$, что соответствует предположению индукции о том, какое утверждение должно быть истинным после каждого раунда.

На каждом раунде алгоритма интеллектуального транзитивного замыкания выполняются операции соединения, агрегирования (устранения дубликатов) и объединения. Следовательно, раунд можно реализовать в виде короткой последовательности заданий MapReduce. К тому же, изрядную часть работы можно устранить за счет комбинирования этих операций, скажем, путем применения более общих видов коммуникации, допустимых системой потоков работ (см. раздел 2.4.1).

**Факты о путях и пути**

Следует четко различать путь, т. е. последовательность дуг, и факт о пути, т. е. утверждение о существовании пути из вершины $x$ в вершину $y$. Факт о пути обычно записывается в виде *Path*($x$, $y$). Алгоритм интеллектуального транзитивного замыкания находит каждый путь ровно один раз, но факт о пути может быть установлен несколько раз. Причина в том, что в графе часто бывает много путей из $x$ в $y$, в том числе и одинаковой длины.

Не все пути обнаруживаются алгоритмом интеллектуального транзитивного замыкания независимо. Например, если существуют дуги $w \to x \to y \to z$, а также дуги $x \to u \to z$, то факт о пути *Path*($w$, $z$) будет установлен дважды: один раз в результате комбинирования *Path*($w$, $y$) с *Path*($y$, $z$) и второй – в результате комбинирования *Path*($w$, $u$) с *Path*($u$, $z$). С другой стороны, если бы существовали ребра $w \to x \to y \to z$ и $w \to v \to y$, то факт *Path*($w$, $z$) был бы установлен только один раз – в результате комбинирования *Path*($w$, $y$) с *Path*($y$, $z$).

## 10.8.6. Транзитивное замыкание посредством сокращения графа

Типичный ориентированный граф, например граф веба, содержит много сильно связных компонент (ССК). С точки зрения транзитивного замыкания, ССК можно свернуть в одну вершину, поскольку из всех вершин ССК достижимы в точ-

ности одни и те же вершины. Существует элегантный алгоритм Дж. Э. Хопкрофта и Р. А. Тарьяна нахождения ССК графа за время, линейно зависящее от размера графа. Однако этот алгоритм принципиально последовательный, основанный на обходе графа в глубину, и для параллельной реализации в больших графах плохо приспособлен.

Мы можем найти большую часть ССК в графе с помощью случайного выбора вершин и двух обходов в ширину. Отметим, что чем больше размер ССК, тем вероятнее, что она свернется рано, поэтому размер графа быстро уменьшается. Опишем алгоритм сворачивания ССК в одиночные вершины. Пусть $G$ – исходный граф, а $G'$ получен из $G$ изменением направления всех дуг на противоположное.

1. Выбрать случайную вершину $v$ из $G$.
2. Найти множество $N_G(v, \infty)$ вершин $G$, достижимых из $v$.
3. Найти множество $N_{G'}(v, \infty)$ вершин графа $G'$, достижимых из $v$. Это то же самое, что нахождение вершин $G$, из которых достижима $v$.
4. Построить ССК $S$, содержащую $v$ в виде $N_G(v, \infty) \cap N_{G'}(v, \infty)$. То есть $v$ и $u$ принадлежат одной ССК графа $G$ тогда и только тогда, $v$ достижима из $u$ и $u$ достижима из $v$.
5. Заменить ССК $S$ единственной вершиной $s$ в $G$. Для этого удалить из $G$ все вершины, вошедшие в $S$, и добавить $s$ в множество вершин $G$. Удалить из $G$ все дуги, хотя бы один конец которых принадлежит $S$. Затем добавить в множество дуг $G$ дугу $s \to x$, если в $G$ существовала дуга, ведущая в $x$ хотя бы из одной вершины, принадлежащей $S$. Наконец, добавить дугу $x \to s$, если существовала дуга, соединяющая $x$ хотя бы с одним элементом $S$.

Мы можем повторить описанные выше шаги фиксированное число раз. Или же повторять их до тех пор, пока граф не станет достаточно малым. Или перебирать все вершины $v$ по очереди и не останавливаться, пока каждая вершина не станет единственным элементом содержащей ее ССК, т. е.

$$N_G(v,\infty) \cap N_{G'}(v,\infty) = \{v\}$$

для всех оставшихся вершин $v$. В последнем случае результирующий граф называется *транзитивным сокращением* исходного графа $G$. Транзитивное сокращение всегда является ациклическим графом, потому что если бы в нем был цикл, то осталась бы ССК, содержащая более одной вершины. Однако необязательно сокращать граф до ациклического, достаточно, если количество оставшихся вершин будет достаточно мало, чтобы можно было вычислить полное транзитивное замыкание. То есть настолько мало, что мы сможем управиться с результирующим графом, размер которого пропорционален квадрату количества оставшихся вершин.

Конечно, транзитивное замыкание сокращенного графа – не то же самое, что транзитивное замыкание исходного, но в сочетании с информацией о том, какой ССК принадлежит каждая вершина исходного графа, его достаточно, чтобы узнать все, что можно было бы почерпнуть из транзитивного замыкания исходного графа. Если мы захотим узнать, верно ли утверждение $Path(u, v)$ в исходном графе,

то нужно будет найти ССК, содержащую $u$ и $v$. Если одна или обе вершины не были свернуты в ССК, то рассматриваем соответствующую вершину как отдельную ССК. Если $u$ и $v$ принадлежат одной ССК, то, разумеется, $v$ достижима из $u$. Если они принадлежат разным ССК, $s$ и $t$ соответственно, то смотрим, достижима ли $t$ из $s$ в сокращенном графе. Если да, то $v$ достижима из $u$ в исходном графе, иначе недостижима.

**Пример 10.30.** Вернемся к изображению веба в виде «галстука-бабочки» в разделе 5.1.3. Количество вершин в исследованной части графа было больше 200 миллионов; обработать данные, объем которых пропорционален квадрату этого числа, вряд ли возможно. Существует большое множество вершин, названное «главной ССК», рассматриваемое как центр графа. Поскольку этой ССК принадлежит примерно одна вершина из четырех, она будет свернута в одну вершину, как только в результате случайного выбора будет взята какая-нибудь из ее вершин. Но в вебе есть много других ССК, хотя они и не показаны явно в «бабочке». Например, внутри входящей компоненты имеется много больших ССК. Вершины любой из таких ССК достижимы друг из друга и могут также достигать некоторые другие вершины входящей компоненты и, разумеется, все вершины центральной ССК. ССК, принадлежащие входящей и исходящей компоненте, трубкам и другим структурам, можно свернуть, получив в результате меньший граф.

## 10.8.7. Аппроксимация размеров окрестностей

В этом разделе мы займемся задачей о вычислении профиля окрестностей для каждой вершины большого графа. Разновидностью этой задачи, решаемой таким же способом, является задача о нахождении размера достижимого множества для каждой вершины $v$, т. е. множества, которое мы обозначили $N(v,\infty)$. Напомним, что для графа с миллиардом вершин практически невозможно вычислить окрестности всех вершин, сколь бы большим вычислительным кластером мы ни располагали. Но даже если мы хотим знать только количество вершин в каждой окрестности, то придется запоминать уже встречавшиеся вершины, иначе мы не будем знать, учитывать вновь встретившуюся вершину или нет.

С другой стороны, нетрудно получить аппроксимацию размеров всех окрестностей, применяя алгоритм Флажоле-Мартена, описанный в разделе 4.4.2. Напомним, что в этом алгоритме используется много хэш-функций; в данном случае хэш-функции применяются к вершинам графа. Важным свойством битовой строки, которую мы получаем, применяя хэш-функцию $h$ к вершине $v$, является «длина хвоста» – количество нулей в конце строки. Размер любого множества вершин оценивается величиной $2^R$, где $R$ – длина самого длинного хвоста для всех элементов множества. Таким образом, вместо хранения всех элементов множества мы можем хранить только значение $R$ для него. Правда, хэш-функций много, и придется хранить значения $R$ для каждой из них.

**Пример 10.31.** Если мы будем использовать хэш-функции, порождающие 64-разрядные строки, то для хранения одного значения $R$ необходимо всего шесть бит. Так, если в графе миллиард вершин и требуется оценить размер окрестности каждой, то при использовании 20 хэш-функций для хранения всех значений $R$ хватит 15 ГБ.

Если мы будем хранить длины хвостов для каждой окрестности, то сможем воспользоваться этой информацией для вычисления оценок бóльших окрестностей по оценкам, полученным для меньших окрестностей. Предположим, что уже вычислены оценки $|N(v, d)|$ для всех вершин $v$, и мы хотим вычислить оценки для окрестностей радиуса $d + 1$. Для каждой хэш-функции $h$ значение $R$ для $N(v, d + 1)$ является максимумом из:

1. Длины хвоста самой вершины $v$ и
2. Значений $R$, ассоциированных с $h$ и $N(u, d)$, где $v \to u$ – некоторая дуга графа.

Отметим, что неважно, достижима вершина только через одну вершину, следующую за $v$ в графе, или через несколько таких вершин. В обоих случаях получается одна и та же оценка. Этим полезным свойством мы уже пользовались в разделе 4.4.2, чтобы не думать о том, встречается элемент в потоке один или несколько раз.

Теперь опишем полностью алгоритм ANF (Approximate Neighborhood Function – приближенная функция окрестностей). Выберем $K$ хэш-функций $h_1, h_2, ..., h_k$. Для каждой вершины $v$ и радиуса $d$ обозначим $R_i(v, d)$ максимальную длину хвоста, вычисленную по всем вершинам из $N(v, d)$ с применением хэш-функции $h_i$. Будем считать, что $R_i(v, 0)$ – длина хвоста $h_i(v)$ для всех $i$ и $v$.

Для выполнения шага индукции предположим, что мы уже вычислили $R_i(v, d)$ для всех $i$ и $v$. Инициализируем значения $R_i(v, d + 1)$, положив их равными $R_i(v, d)$ для всех $i$ и $v$. Рассмотрим все присутствующие в графе дуги $x \to y$ в любом порядке. Для каждой дуги $x \to y$ положим $R_i(x, d + 1)$ равным максимуму из текущего значения и $R_i(y, d)$. Отметим, что возможность рассматривать дуги в любом порядке может существенно повысить скорость работы в случае, когда можно хранить $R_i$ в оперативной памяти, тогда как множество дуг настолько велико, что должно храниться на диске. Мы можем читать дисковые блоки, содержащие дуги, потоком по одному и, стало быть, просматривать каждый блок только один раз. Эта удобная возможность похожа на упомянутую в разделе 6.2.1, где мы заметили, что алгоритмы поиска частых предметных наборов типа Apriori могли бы читать данные о корзинах потоком, тогда каждый дисковый блок читался бы только один раз в каждом раунде.

Для оценки размера $N(v, d)$ скомбинируем значения $R_i(v, d)$ для $i = 1, 2, ..., k$, как было предложено в разделе 4.4.3. То есть объединяем значения $R$ в небольшие группы, вычисляем для них средние и берем медиану средних.

Еще одно усовершенствование алгоритма ANF возможно, если нас интересуют только оценки размеров достижимых множеств, $N(v, \infty)$. Тогда не нужно хранить

значения $R_i(v, d)$ для разных радиусов $d$. Можно запоминать одно значение $R_i(v)$ для каждой хэш-функции $h_i$ и каждой вершины $v$. Когда в каком-то раунде мы рассматриваем дугу $x \to y$, то просто присваиваем:

$$R_i(x) := \max(R_i(x), R_i(y))$$

Мы можем прекратить итерации, когда после очередного раунда ни одно значение $R_i(v)$ не изменилось. А если заранее известен диаметр $d$ графа, то можно остановиться после $d$ итераций.

## 10.8.8. Упражнения к разделу 10.8

**Упражнение 10.8.1.** Для графа, изображенного на рис. 10.9 и повторенного на рис. 10.24, ответьте на следующие вопросы:

(*а*) Если представить этот граф в виде ориентированного, то сколько в нем будет дуг?

(*б*) Каковы профили окрестностей вершин $A$ и $B$?

(*в*) Чему равен диаметр графа?

(*г*) Сколько пар вершин в транзитивном замыкании? *Подсказка:* не забудьте, что в этом графе существуют пути ненулевой длины из вершины в нее саму.

(*д*) Сколько раундов понадобится для вычисления транзитивного замыкания этого графа методом рекурсивного удваивания?

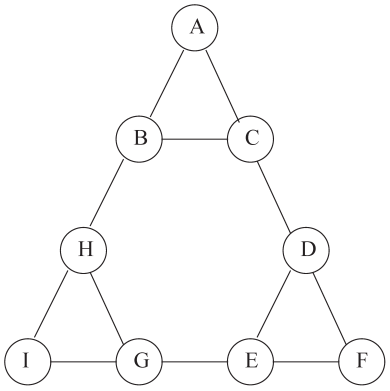



**Рис. 10.24.** Граф, используемый в упражнениях на тему окрестностей и транзитивного замыкания

**Упражнение 10.8.2.** В алгоритме интеллектуального транзитивного замыкания пути произвольной длины разбиваются на голову и хвост определенной длины. Какова длина головы и хвоста для путей длины 7, 8 и 9?

**Упражнение 10.8.3.** Рассмотрим пример социальной сети на рис. 10.23. Предположим, что используется одна хэш-функция, которая отображает вершину (заглавную букву) на ее код ASCII. Напомним, что код ASCII буквы $A$ – 01000001, а букв $B$, $C$, ... – 01000010, 01000011, ... .

(*а*) Применяя эту хэш-функцию, вычислите значения $R$ для каждой вершины и радиуса 1. Чему равны оценки размеров каждой окрестности? Насколько далеки они от реальности?

(*б*) Вычислите значения $R$ для каждой вершины и радиуса 2. Снова вычислите оценки и сравните их с реальностью.

(*в*) Диаметр этого графа равен 3. Для каждой вершины графа вычислите значение $R$ и оценку размера для множества достижимых из нее вершин.

(*г*) Возьмем еще одну хэш-функцию *g*, равную 1 плюс код ASCII буквы. Повторите упражнения (*а*) – (*в*) для этой хэш-функции. Возьмите в качестве оценки размера окрестности среднее из оценок, вычисленных для *h* и *g*. Насколько теперь оценки близки к реальности?

# 10.9. Резюме

- *Графы социальных сетей*. Графы, представляющие связи в социальной сети, не только очень велики, но и обладают свойством локальности, когда в небольших подмножествах вершин (сообществах) плотность ребер гораздо выше средней.
- *Сообщества и кластеры*. Хотя сообщества в некоторых отношениях и напоминают кластеры, между ними есть существенные различия. Люди (вершины) обычно входят в несколько сообществ, а обычные метрики не способны представить близость вершин сообщества. Поэтому стандартные алгоритмы поиска кластеров неприменимы для нахождения сообществ.
- *Промежуточность*. Один из способов выделить сообщества вершин состоит в том, чтобы измерить промежуточность ребер – сумму по всем парам вершин доли кратчайших путей между вершинами, проходящих через данное ребро. Для формирования сообществ нужно удалить ребра, промежуточность которых выше заданного порога.
- *Алгоритм Гирвана-Ньюмана*. Это эффективный алгоритм вычисления промежуточности ребер. Из каждой вершины производится обход графа в ширину, и в ходе последовательности шагов пометки вычисляется доля путей из корня во все остальные вершины, проходящих по каждому ребру. Доли, вычисленные для каждого ребра при обходе из каждой вершины (корня), суммируются для вычисления промежуточности.
- *Сообщества и полные двудольные графы*. В полном двудольном графе имеются две группы вершин, существуют все возможные ребра, соединяющие пары вершин по одной из каждой группы, и не существует ни одного ребра, соединяющего вершины из одной группы. В любом достаточно плотном сообществе (множестве вершин, между которыми проведено много ребер) имеется большой полный двудольный граф.
- *Нахождение полных двудольных графов*. Для поиска полных двудольных графов можно применить тот же метод, что для поиска частых предметных наборов. Вершины графа можно рассматривать как предметы и корзины. Корзиной, соответствующей вершине, является множество смежных с ней вершин, интерпретируемых как предметы. Поиск полного двудольного графа с группами вершин размером *t* и *s* можно рассматривать как поиск частых предметных наборов размера *t* с поддержкой *s*.
- *Разрезание графа*. Один из способов поиска сообществ заключается в многократном разрезании графа на части примерно одинакового размера.

Разрезом называется разбиение множества вершин графа на два подмножества, а его размером – количество ребер с концами в разных подмножествах. Объемом множества вершин называется число ребер, хотя бы один конец которых принадлежит данному множеству.

- *Нормализованные разрезы.* Мы можем нормализовать размер разреза, взяв его отношения к объемам каждого из двух образовавшихся в результате разрезания множеств. Сумма этих отношений и дает нормализованное значение разреза. Нормализованные разрезы с малой суммой хороши в том смысле, что делят множество вершин на две примерно равные части, а сам размер разреза относительно мал.
- *Матрица смежности.* Такая матрица описывает граф. Элемент на пересечении строки $i$ и столбца $j$ равен 1, если существует ребро, соединяющее вершины $i$ и $j$, и 0 в противном случае.
- *Степенная матрица.* В $i$-ом элементе на диагонали степенной матрицы графа находится значение $d$, равное степени $i$-ой вершины. Все элементы вне диагонали равны 0.
- *Матрица Лапласа.* Матрицей Лапласа графа называется разность между его степенной матрицей и матрицей смежности. Элемент на пересечении строки $i$ и столбца $i$ матрицы Лапласа равен степени $i$-ой вершины графа, а элемент на пересечении строки $i$ и столбца $j$ для $i \neq j$ равен $-1$, если существует ребро, соединяющее вершины $i$ и $j$, и 0 в противном случае.
- *Спектральный метод разрезания графов.* Наименьшее собственное значение любой матрицы Лапласа равно 0, а соответствующий ему собственный вектор состоит из всех единиц. Собственные векторы небольших собственных значений можно использовать для разрезания графа на две части близких размеров с небольшой величиной разреза. Например, можно поместить в одно множество вершины, соответствующие положительным элементам собственного вектора второго снизу по величине собственного значения, а в другое – вершины, соответствующие его отрицательным элементам.
- *Пересекающиеся сообщества.* Как правило, человек входит в несколько сообществ. В графах социальных сетей очень часто вероятность, что два человека являются друзьями, возрастает вместе с увеличением количества сообществ, в которые они входят одновременно.
- *Модель графа принадлежности.* Для моделирования членства в сообществах удобно предположить, что для каждого сообщества определена вероятность, что дружба между двумя людьми (наличие соединяющего их ребра в графе) обусловлена пребыванием в этом сообществе. Таким образом, вероятность, что две вершины соединены ребром, равна 1 минус произведение вероятностей того, что ни одно из сообществ, членами которых являются обе вершины, не обусловило наличие ребра между ними. Затем мы находим такое распределение вершин по сообществам и такие значения вероятностей, которые наилучшим образом описывают наблюдаемый граф.

- *Оценка максимального правдоподобия.* Важный метод моделирования, полезный, в частности, для моделирования сообществ. Заключается в вычислении вероятности порождения наблюдаемого объекта в виде функции от значений всех параметров, допустимых моделью. Правильными считаются значения параметров, для которых вероятность максимальна; они называются оценкой максимального правдоподобия (ОМП).
- *Метод градиентного спуска.* Если членство в сообществах известно, то ОМП можно найти методом градиентного спуска или другими методами. Но найти наилучшую оценку членства в сообществах методом градиентного спуска невозможно, потому что членство – величина дискретная, а не непрерывная.
- *Улучшенное моделирование сообществ с помощью сила членства.* Мы можем по-другому поставить задачу о нахождении ОМП сообществ в социальном графе, предположив, что с участником ассоциирована сила членства в каждом сообществе, равная 0, если он не является членом. Если определить вероятность существования ребра между двумя вершинами как функцию от силы членства этих вершин в общих сообществах, то задачу нахождения ОМП можно преобразовать в непрерывную и решить методом градиентного спуска.
- *Simrank.* Один из способов измерить сходство вершин графа, содержащего вершины нескольких типов, – начать случайное блуждание в некоторой вершине, задав фиксированную вероятность перезапуска с той же самой вершины. Распределение вершин, в которые может попасть случайный посетитель, – хорошая мера сходства вершин с той, откуда началось блуждание. Если мы хотим вычислить попарное сходство вершин, то этот процесс следует повторить для каждой вершины в роли начальной.
- *Треугольники в социальных сетях.* Среднее число треугольников на одну вершину – важная мера компактности сообщества, отражающая его зрелость. Подсчитать треугольники в графе, содержащем $m$ ребер, можно за время $O(m^{3/2})$, и в общем случае более эффективного алгоритма не существует.
- *Нахождение треугольников с помощью MapReduce.* Найти треугольники можно за один раунд MapReduce, рассматривая задачу как трехпутевое соединение. Количество редукторов, которым необходимо отправить каждое ребро, пропорционально кубическому корню из общего числа редукторов, а общее время вычисления, затраченное всеми редукторами, пропорционально времени работы последовательного алгоритма нахождения треугольников.
- *Окрестности.* Окрестностью радиуса $d$ вершины $v$ в ориентированном или неориентированном графе называется множество вершин, достижимых из $v$ по путям длины не больше $d$. Профилем окрестностей вершины называется последовательность размеров окрестностей всех радиусов, начиная с 1.

Диаметром связного графа называется наименьшее значение $d$, для которого окрестность радиуса $d$ любой вершины содержит весь граф целиком.

- *Транзитивное замыкание*. Вершина $u$ достижима из $v$, если $u$ принадлежит окрестности $v$ некоторого радиуса. Транзитивным замыканием графа называется множество пар вершин ($v$, $u$) таких, что $u$ достижима из $v$.
- *Вычисление транзитивного замыкания*. Поскольку число фактов в транзитивном замыкании может быть равно квадрату числа вершин графа, практически невозможно непосредственно вычислить транзитивное замыкание большого графа. Один из возможных подходов – найти сильно связные компоненты графа и свернуть каждую в одну вершину, перед тем как вычислять транзитивное замыкание.
- *Транзитивное замыкание и MapReduce*. Вычисление транзитивного замыкания можно рассматривать как итеративное соединение отношения путей (пар вершин $v$ и $u$ таких, что $u$ заведомо достижима из $v$) и отношения дуг графа. При таком подходе требуемое количество раундов MapReduce равно диаметру графа.
- *Вычисление транзитивного замыкания методом рекурсивного удвоения*. Существует другой подход, требующий меньшего числа раундов MapReduce, – соединять на каждом раунде отношение путей с собой. Тогда на каждом раунде удваивается длина путей, которые могут давать вклад в транзитивное замыкание. Поэтому число раундов равно лишь двоичному логарифму диаметра графа.
- *Интеллектуальное транзитивное замыкание*. При рекурсивном удвоении один и тот же путь может рассматриваться многократно, что увеличивает общее время вычисления (по сравнению с итеративным соединением путей с одиночными дугами). В то же время вариант этого алгоритма, называемый интеллектуальным транзитивным замыканием, позволяет избежать обнаружения одного пути несколько раз. Идея в том, чтобы длина первого из соединяемых путей была степенью двойки.
- *Аппроксимация размеров окрестностей*. Применяя метод Флажоле-Мартена аппроксимации числа различных элементов потока, мы можем найти приближенные размеры окрестностей разных радиусов. Для каждой вершины хранится множество длин хвостов. При увеличении радиуса на 1 мы рассматриваем каждое ребро ($u$, $v$) и для каждой длины хвоста $u$ устанавливаем ее равной длине соответствующего хвоста $v$, если вторая больше первой.

# 10.10. Список литературы

Simrank описан в работе [8]. В [11] применяется альтернативный подход, при котором сходство двух вершин рассматривается как вероятность, что два случайных блуждания, начатые из этих вершин, сойдутся в одной вершине. В [3] случайное

блуждание комбинируется с классификацией вершин для предсказания связей в графе социальной сети. В [16] изучается эффективность вычисления simrank в качестве персонализированного PageRank.

Алгоритм Гирвана-Ньюмана заимствован из [6]. Нахождение сообществ посредством поиска полных двудольных графов впервые упомянуто в [9].

Нормализованные разрезы в связи со спектральным анализом описаны в [13]. Работа [19] содержит обзор спектральных методов нахождения кластеров, а работа [5] – более общий обзор нахождения сообществ в графах. [10] содержит анализ сообществ во многих сетях, встречающихся на практике.

Поиск пересекающихся сообществ исследовался в работах [20], [21] и [22].

Подсчет треугольников с использованием MapReduce обсуждается в [15]. Описанный здесь метод взят из [1], где описана также техника, пригодная для любого подграфа. В работе [17] обсуждаются рандомизированные алгоритмы нахождения треугольников.

Алгоритм ANF впервые был изучен в [12]. В [4] описывается, как можно дополнительно ускорить первоначальный вариант ANF.

Алгоритм интеллектуального транзитивного замыкания был независимо открыт в работах [7] и [18]. Реализация транзитивного замыкания с помощью системы MapReduce и ей подобных обсуждается в [2].

Реализацию на C++ с открытым исходным кодом многих описанных в этой главе алгоритмов можно найти в библиотеке SNAP [14].

1. F. N. Afrati, D. Fotakis, J. D. Ullman «Enumerating subgraph instances by map-reduce», `http://ilpubs.stanford.edu:8090/1020`
2. F. N. Afrati, J. D. Ullman, «Transitive closure and recursive Datalog implemented on clusters», in Proc. EDBT (2012).
3. L. Backstrom, J. Leskovec «Supervised random walks: predicting and recommending links in social networks», Proc. Fourth ACM Intl. Conf. on Web Search and Data Mining (2011), pp. 635–644.
4. P. Boldi, M. Rosa, S. Vigna «HyperANF: approximating the neighbourhood function of very large graphs on a budget», Proc. WWW Conference (2011), pp. 625–634.
5. S. Fortunato «Community detection in graphs», Physics Reports 486:3–5 (2010), pp. 75–174.
6. M. Girvan, M.E.J. Newman «Community structure in social and biological networks», Proc. Natl. Acad. Sci. 99 (2002), pp. 7821–7826.
7. Y. E. Ioannidis «On the computation of the transitive closure of relational operators», Proc. 12th Intl. Conf. on Very Large Data Bases, pp. 403–411.
8. G. Jeh, J. Widom «Simrank: a measure of structural-context similarity», Proceedings of the eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2002), pp. 538–543.
9. R. Kumar, P. Raghavan, S. Rajagopalan, A. Tomkins «Trawling the Web for emerging cyber-communities, Computer Networks 31:11–16 (May, 1999), pp. 1481–1493.

10. J. Leskovec, K. J. Lang, A. Dasgupta, M. W. Mahoney «Community structure in large networks: natural cluster sizes and the absence of large well-defined clusters», http://arxiv.org/abs/0810.1355.
11. S. Melnik, H. Garcia-Molina, E. Rahm «Similarity flooding: a versatile graph matching algorithm and its application to schema matching», Proc. Intl. Conf. on Data Engineering (2002), pp. 117–128.
12. C.R. Palmer, P.B. Gibbons, C. Faloutsos «ANF: a fast and scalable tool for data mining in massive graphs», Proc. Eighth ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining (2002), pp. 81–90.
13. J. Shi, J. Malik «Normalized cuts and image segmentation», IEEE Trans. on Pattern Analysis and Machine Intelligence», 22:8 (2000), pp. 888–905.
14. Stanford Network Analysis Platform, http://snap.stanford.edu.
15. S. Suri and S. Vassilivitskii «Counting triangles and the curse of the last reducer», Proc. WWW Conference (2011).
16. H. Tong, C. Faloutsos, J.-Y. Pan «Fast random walk with restart and its applications», ICDM 2006, pp. 613–622.
17. C. E. Tsourakakis, U. Kang, G. L. Miller, C. Faloutsos «DOULION: counting triangles in massive graphs with a coin», Proc. Fifteenth ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining (2009).
18. P. Valduriez, H. Boral «Evaluation of recursive queries using join indices», Expert Database Conf. (1986), pp. 271–293.
19. U. von Luxburg «A tutorial on spectral clustering», Statistics and Computing bf17:4 (2007), 2007, pp. 395–416.
20. J. Yang, J. Leskovec «Overlapping community detection at scale: a nonnegative matrix factorization approach», ACM International Conference on Web Search and Data Mining, 2013.
21. J. Yang, J. McAuley, J. Leskovec «Detecting cohesive and 2-mode communities in directed and undirected networks», ACM International Conference on Web Search and Data Mining, 2014.
22. J. Yang, J. McAuley, J. Leskovec «Community detection in networks with node attributes», IEEE International Conference On Data Mining, 2013.

# ГЛАВА 11. Понижение размерности

Многие источники данных можно рассматривать как большую матрицу. В главе 5 мы видели, как представить веб в виде матрицы переходов. В главе 9 в центре нашего внимания была матрица предпочтений. А в главе 10 мы исследовали матрицы, представляющие социальные сети. Во многих подобных приложениях матрицу можно обобщить, подобрав «более узкие» матрицы, в каком-то смысле близкие к исходной. Но из-за того, что в новой матрице меньше строк или столбцов, работа с ней оказывается гораздо более эффективной, чем с исходной. Процесс поиска таких узких матриц называется *понижением размерности*.

Предварительный пример понижения размерности мы видели в разделе 9.4. Там обсуждалась UV-декомпозиция матрицы и был приведен простой алгоритм нахождения такой декомпозиции. Напомним, что требовалось разложить большую матрицу $M$ на две матрицы $U$ и $V$, произведение которых приближенно равно $M$. В матрице $U$ было мало столбцов, а в матрице $V$ мало строк, так что каждая была значительно меньше $M$, но вместе они содержали большую часть информации, присутствующей в $M$, и полезной для предсказания оценок, выставляемых пользователями.

В этой главе мы исследуем идею понижения размерности более детально. Начнем с обсуждения собственных значений и их использования в «методе главных компонент» (principal component analysis, PCA). Мы рассмотрим сингулярное разложение, обобщение UV-декомпозиции. И наконец, поскольку нас всюду интересуют данные максимального размера, допускающего обработку, мы рассмотрим еще одно разложение – CUR-декомпозицию – вариант сингулярного разложения, при котором матрицы-сомножители оказываются разреженными, если таковой была исходная матрица.

## 11.1. Собственные значения и собственные векторы

Предполагается, что читатель знаком с основами матричной алгебры: умножением, транспонированием, определителями и решением систем линейных уравнений. В этом разделе мы определим собственные значения и собственные векторы

симметричной матрицы и покажем, как их находить. Напомним, что матрица называется симметричной, если элемент на пересечении $i$-й строки и $j$-го столбца равен элементу на пересечении $j$-й строки и $i$-го столбца.

## 11.1.1. Определения

Пусть $M$ – квадратная матрица, а $\mathbf{e}$ – ненулевой вектор-столбец с таким же числом строк, как в $M$. Число $\lambda$ называется *собственным значением* $M$, а $\mathbf{e}$ соответствующим ему *собственным вектором*, если $M\mathbf{e} = \lambda\mathbf{e}$.

Если $\mathbf{e}$ – собственный вектор $M$, а $c$ – произвольная константа, то $c\mathbf{e}$ также является собственным вектором $M$ с тем же самым собственным значением. Умножение вектора на константу изменяет его длину, но сохраняет направление. Поэтому, чтобы избежать неоднозначности, потребуем, чтобы собственный вектор был *единичным*, т. е. сумма квадратов его элементов была равна 1. Но даже это еще не делает собственный вектор уникальным, потому что умножение на $-1$ не изменяет сумму квадратов элементов. Поэтому потребуем еще, чтобы первый ненулевой элемент собственного вектора был положителен.

**Пример 11.1.** Пусть $M$ – матрица

$$\begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix}.$$

Один из собственных векторов $M$ равен

$$\begin{bmatrix} 1/\sqrt{5} \\ 2/\sqrt{5} \end{bmatrix},$$

а соответствующее ему собственное значение равно 7. Это следует из равенства

$$\begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix}\begin{bmatrix} 1/\sqrt{5} \\ 2/\sqrt{5} \end{bmatrix} = 7\begin{bmatrix} 1/\sqrt{5} \\ 2/\sqrt{5} \end{bmatrix}$$

в котором обе части равны

$$\begin{bmatrix} 7/\sqrt{5} \\ 14/\sqrt{5} \end{bmatrix}.$$

Отметим еще, что этот собственный вектор единичный, потому что $(1/\sqrt{5})^2+(2/\sqrt{5})^2 = 1/5 + 4/5 = 1$.

## 11.1.2. Вычисление собственных значений и собственных векторов

С одним способом нахождения собственных значений и собственных векторов матрицы $M$ мы уже встречались в разделе 5.1: начать с произвольного единичного

вектора **v** подходящей размерности и итеративно вычислять $M^i\mathbf{v}$, пока процесс не сойдется.[1] Если $M$ – стохастическая матрица, то в пределе получится *главный* собственный вектор (с максимальным собственным значением), а соответствующее ему собственное значение равно 1.[2] Этот способ нахождения главного собственного вектора называется *степенным методом*. Он является вполне общим, но если главное собственное значение (соответствующее главному собственному вектору) не равно 1, то с ростом $i$ отношение $M^{i+1}\mathbf{v}$ к $M^i\mathbf{v}$ стремится к главному собственному значению, а $M^i\mathbf{v}$ – к вектору (возможно, не единичному), направленному так же, как главный собственный вектор.

В разделе 11.1.3 мы рассмотрим обобщение степенного метода для поиска всех собственных пар. Но существует метод точного вычисления всех собственных пар симметричной матрицы размерности $n \times n$, требующий времени порядка $O(n^3)$, и его мы рассмотрим сначала. Всегда существует $n$ собственных пар, хотя иногда некоторые собственные значения совпадают. Начнем с того, что перепишем определение собственной пары $M\mathbf{e} = \lambda\mathbf{e}$ в виде $(M - \lambda I)\mathbf{e} = \mathbf{0}$, где

1. $I$ – единичная матрица, в которой элементы главной диагонали равны 1, а все остальные 0.
2. $\mathbf{0}$ – вектор, все элементы которого равны 0.

Из линейной алгебры известно, что для выполнения равенства $(M - \lambda I)\mathbf{e} = 0$ при $\mathbf{e} \neq \mathbf{0}$ определитель матрицы $M - \lambda I$ должен быть равен 0. Заметим, что матрица $(M - \lambda I)$ отличается от $M$ только элементами главной диагонали: там, где в $M$ на диагонали находится значение $c$, в $(M - \lambda I)$ находится $c - \lambda$. Хотя определитель матрицы размерности $n \times n$ состоит из $n!$ слагаемых, его можно вычислить, и при том разными способами, за время $O(n^3)$; один из них называется методом «последовательной конденсации» (pivotal condensation).

Определитель матрицы $(M - \lambda I)$ представляет собой полином степени $n$ от $\lambda$, по которому мы можем найти $n$ значений $\lambda$, являющихся собственными значениями $M$. Для каждого такого значения $c$ мы можем затем решить уравнение $M\mathbf{e} = c\mathbf{e}$. Это система $n$ уравнений с $n$ неизвестными (элементы **e**), но поскольку ни в одном уравнении нет свободного члена, то найти из нее **e** можно только с точностью до постоянного множителя. Однако любое найденное решение можно нормировать, так чтобы сумма квадратов элементов вектора была равна 1, и таким образом получить собственный вектор, соответствующий собственному значению $c$.

**Пример 11.2.** Найдем собственные пары для матрицы $M$ размерности $2 \times 2$ из примера 11.1. Напомним, что $M$ равна

$$\begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix}.$$

[1] Напомним, что $M^i$ означает умножение матрицы $M$ на себя $i$ раз.

[2] Отметим, что стохастическая матрица, вообще говоря, не обязана быть симметричной. Симметричные и стохастические матрицы – два класса матриц, для которых собственные пары существуют и имеют полезные применения. В этой главе мы рассматриваем методы, относящиеся к симметричным матрицам.

Тогда $M - \lambda I$ равна

$$\begin{bmatrix} 3-\lambda & 2 \\ 2 & 6-\lambda \end{bmatrix}.$$

Определитель этой матрицы, равный $(3 - \lambda)(6 - \lambda) - 4$, нужно приравнять нулю и решить получающееся квадратное уравнение $\lambda^2 - 9\lambda + 14 = 0$ относительно $\lambda$. Это уравнение имеет корни $\lambda = 7$ и $\lambda = 2$; первый дает главное собственное значение, поскольку он больше. Пусть **e** – вектор неизвестных

$$\begin{bmatrix} x \\ y \end{bmatrix}.$$

Нам нужно решить уравнение

$$\begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = 7\begin{bmatrix} x \\ y \end{bmatrix}.$$

После умножения матрицы на вектор получаем два уравнения:

$$3x + 2y = 7x$$
$$2x + 6y = 7y$$

Отметим, что оба уравнения сводятся к одному и тому же: $y = 2x$. Поэтому один из возможных собственных векторов равен

$$\begin{bmatrix} x \\ y \end{bmatrix}.$$

Но это не единичный вектор, потому что сумма квадратов его элементов равна 5, а не 1. Чтобы получить единичный вектор того же направления, разделим каждый элемент на $\sqrt{5}$. Получаем главный собственный вектор

$$\begin{bmatrix} 1/\sqrt{5} \\ 2/\sqrt{5} \end{bmatrix}$$

и его собственное значение 7. Именно эта собственная пара и упоминалась в примере 11.1.

Чтобы найти вторую собственную пару, повторим описанные выше действия для собственного значения 2. Элементы вектора **e** связаны уравнением $x = -2y$, из которого находим второй собственный вектор

$$\begin{bmatrix} 2/\sqrt{5} \\ -1/\sqrt{5} \end{bmatrix}.$$

Разумеется, ему соответствует собственное значение 2.

## 11.1.3. Нахождение собственных пары степенным методом

Теперь рассмотрим обобщение алгоритма, использованного в разделе 5.1 для нахождения главного собственного вектора. Тогда этим собственным вектором был

вектор PageRank – из всех собственных векторов стохастической матрицы веба только он нас и интересовал. Мы начнем с вычисления главного собственного вектора, немного обобщив примененный ранее подход. Затем модифицируем матрицу таким образом, чтобы исключить главный собственный вектор, и в результате получим новую матрицу, главный собственный вектор которой равен второму собственному вектору (со вторым по величине собственным значением) исходной матрицы. Эта процедура повторяется: исключение каждого найденного собственного вектора и последующее применение степенного метода для нахождения главного собственного вектора оставшейся матрицы.

Пусть $M$ – матрица, для которой мы ищем собственные пары. Начав с произвольного ненулевого вектора $\mathbf{x}_0$, будем вычислять последовательные приближения:

$$\mathbf{x}_{k+1} := \frac{M_{\mathbf{x}_k}}{\left\| M_{\mathbf{x}_k} \right\|},$$

где $\|N\|$ обозначает норму Фробениуса матрицы или вектора $N$, т. е. квадратный корень из суммы квадратов элементов $N$. Мы умножаем текущий вектор $\mathbf{x}_k$ на матрицу $M$, пока не будет выполнено условие сходимости ($\|x_k - x_{k+1}\|$ меньше малой наперед заданной константы). Обозначим $\mathbf{x}$ вектор $\mathbf{x}_k$ для того значения $k$, при котором была достигнута сходимость. Тогда $\mathbf{x}$ приближенно равен главному собственному вектору $M$. Для получения соответствующего собственного значения нужно лишь вычислить $\lambda_1 = \mathbf{x}^{\mathrm{T}} M\mathbf{x}$; это и есть решение уравнения $M\mathbf{x} = \lambda\mathbf{x}$ относительно $\lambda$, поскольку $\mathbf{x}$ – единичный вектор.

**Пример 11.3.** Возьмем матрицу из примера 11.2

$$M = \begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix}$$

и начнем с вектора $\mathbf{x}_0$, в котором оба элемента равны 1. Для вычисления $\mathbf{x}_1$ умножаем $M$ на $\mathbf{x}_0$ и получаем

$$\begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 5 \\ 8 \end{bmatrix}.$$

Норма Фробениуса результата равна $\sqrt{5^2 + 8^2} = \sqrt{89} = 9.434$. Для получения $\mathbf{x}_1$ делим 5 и 8 на 9.434:

$$\mathrm{x}_1 = \begin{bmatrix} 0.530 \\ 0.848 \end{bmatrix}.$$

На следующей итерации вычисляем

$$\begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} 0.530 \\ 0.848 \end{bmatrix} = \begin{bmatrix} 3.286 \\ 6.148 \end{bmatrix}.$$

Норма Фробениуса результата равна 6.971, поэтому после деления получаем

$$x_2 = \begin{bmatrix} 0.471 \\ 0.882 \end{bmatrix}.$$

Процесс сходится к нормальному вектору, второй элемент которого в два раза больше первого. Таким образом, в пределе получается следующий главный собственный вектор:

$$x = \begin{bmatrix} 0.447 \\ 0.894 \end{bmatrix}.$$

Наконец, вычисляем главное собственное значение:

$$\lambda_1 = \mathbf{x}^T M\mathbf{x} = \begin{bmatrix} 0.447 & 0.894 \end{bmatrix} \begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} 0.447 \\ 0.894 \end{bmatrix} = 6.993 .$$

В примере 11.2 было показано, что истинное главное собственное значение равно 7. Степенной метод дает небольшую погрешность либо из-за ограниченной точности вычислений, как в данном случае, либо потому что мы прекратили итерации раньше, чем дошли до точного значения собственного вектора. При вычислении PageRank небольшая неточность не играла роли, но при вычислении всех собственных пар ошибки накапливаются, если не принять мер предосторожности.

Для нахождения второй собственной пары создадим новую матрицу $M^* = M - \lambda_1 \mathbf{x}\mathbf{x}^T$. Затем применим к $M^*$ степенной метод для вычисления ее наибольшего собственного значения. Полученные $\mathbf{x}^*$ и $\lambda^*$ равны второму по величине собственному значению и соответствующему собственному вектору матрицы $M$.

Интуитивно понятно, что эта операция устраняет влияние собственного вектора, делая соответствующее ему собственное значение равным 0. Формальное доказательство вытекает из двух наблюдений. Если $M^* = M - \lambda \mathbf{x}\mathbf{x}^T$, где $\mathbf{x}$ и $\lambda$ – собственная пара с наибольшим собственным значением, то:

1. $\mathbf{x}$ также является собственным вектором $M^*$, которому соответствует собственное значение 0. Для доказательства заметим, что

$$M^*\mathbf{x} = (M - \lambda \mathbf{x}\mathbf{x}^T)\mathbf{x} = M\mathbf{x} - \lambda \mathbf{x}\mathbf{x}^T\mathbf{x} = M\mathbf{x} - \lambda \mathbf{x} = 0.$$

   На предпоследнем шаге мы воспользовались тем, что $\mathbf{x}^T\mathbf{x} = 1$, поскольку $\mathbf{x}$ – единичный вектор.

2. Наоборот, если $\mathbf{v}$ и $\lambda_v$ – собственная пара $M$, отличная от первой собственной пары $(\mathbf{x}, \lambda)$, то она является также собственной парой $M^*$. Доказательство:

$$M^*\mathbf{v} = (M^*)^T\mathbf{v} = (M - \lambda \mathbf{x}\mathbf{x}^T)^T\mathbf{v} = M^T\mathbf{v} - \lambda \mathbf{x}(\mathbf{x}^T\mathbf{v}) = M^T\mathbf{v} = \lambda_v \mathbf{v}.$$

   Эта цепочка преобразований требует некоторых пояснений.

(*а*) Собственные значения и собственные векторы $M$ и $M^{\mathrm{T}}$ одинаковы. Отметим, что если матрица $M$ симметрична, то $M = M^{\mathrm{T}}$, но это утверждение справедливо, даже если $M$ не симметрична, хотя доказывать это мы здесь не будем.

(*б*) Собственные векторы матрицы *ортогональны*, т. е. скалярное произведение любых двух различных собственных векторов равно 0. Это утверждение мы тоже приводим без доказательства.

**Пример 11.4.** Продолжая пример 11.3, вычисляем

$$M^{*} = \begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix} - 6.993 \begin{bmatrix} 0.447 & 0.894 \end{bmatrix} \begin{bmatrix} 0.447 \\ 0.894 \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ 2 & 6 \end{bmatrix} - \begin{bmatrix} 1.399 & 2.787 \\ 2.797 & 5.587 \end{bmatrix} = \begin{bmatrix} 2.601 & -0.797 \\ -0.797 & 0.413 \end{bmatrix}.$$

Вторую собственную пару мы сможем найти, обработав эту матрицу так же, как исходную матрицу $M$.

## 11.1.4. Матрица собственных векторов

Пусть имеется матрица $M$ размерности an $n \times n$, обозначим $\mathbf{e}_1$, $\mathbf{e}_2$, ..., $\mathbf{e}_n$ ее собственные векторы, рас.сматриваемые как векторы-столбцы. Пусть $E$ – матрица, в которой $i$-ым столбцом является $\mathbf{e}_i$. Тогда $EE^{\mathrm{T}} = E^{\mathrm{T}}E = I$. Объясняется это тем, что собственные векторы матрицы образуют *ортонормальный базис*, т. е. являются ортогональными единичными векторами.

**Пример 11.5.** Для матрицы $M$ из примера 11.2 матрица $E$ равна

$$\begin{bmatrix} 2/\sqrt{5} & 1/\sqrt{5} \\ -1/\sqrt{5} & 2/\sqrt{5} \end{bmatrix}.$$

Поэтому $E^{\mathrm{T}}$ равна

$$\begin{bmatrix} 2/\sqrt{5} & -1/\sqrt{5} \\ 1/\sqrt{5} & 2/\sqrt{5} \end{bmatrix}.$$

Вычисляя $EE^{\mathrm{T}}$, получаем

$$\begin{bmatrix} 4/5+1/5 & -2/5+2/5 \\ -2/5+2/5 & 1/5+4/5 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Аналогично можно вычислить $E^{\mathrm{T}}E$. Заметим, что элементы на главной диагонали – это суммы квадратов элементов каждого собственного вектора, и не удивительно, что они равны 1 – ведь все собственные векторы единичные. Нули вне главной диагонали получились, потому что элемент на пересечении $i$-ой строки и $j$-го столбца равен скалярному произведению $i$-го и

*j*-го собственных векторов. А поскольку собственные векторы ортогональны, то эти скалярные произведения равны нулю.

## 11.1.5. Упражнения к разделу 11.1

**Упражнение 11.1.1.** Найдите единичный вектор, направленный так же, как вектор [1, 2, 3].

**Упражнение 11.1.2.** Закончите пример 11.4, вычислив главный собственный вектор построенной в нем матрицы. Насколько близок он оказался к точному решению (из примера 11.2)?

**Упражнение 11.1.3.** Для любой симметричной матрицы $3 \times 3$

$$\begin{bmatrix} a-\lambda & b & c \\ b & d-\lambda & e \\ c & e & f-\lambda \end{bmatrix}$$

существует кубическое уравнение относительно $\lambda$, выражающее тот факт, что определитель этой матрицы равен 0. Выпишите это уравнение в терминах значений $a, ..., f$.

**Упражнение 11.1.4.** Найдите собственные пары следующей матрицы:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 3 & 5 \end{bmatrix}$$

применив метод из раздела 11.1.2.

**! Упражнение 11.1.5.** Найдите собственные пары следующей матрицы:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 3 & 6 \end{bmatrix}$$

применив метод из раздела 11.1.2.

**Упражнение 11.1.6.** Для матрицы из упражнения 11.1.4:

(*а*) Начав с вектора, все элементы которого равны 1, и применяя степенной метод, найдите приближенное значение главного собственного вектора.

(*б*) Вычислите оценку главного собственного значения матрицы.

(*в*) Постройте новую матрицу, устранив влияние главной собственной пары, как описано в разделе 11.1.3.

(*г*) Из матрицы, построенной в п. (*в*), найдите вторую собственную пару исходной матрицы.

(*д*) Повторите шаги (*в*) и (*г*) для нахождения третьей собственной пары исходной матрицы.

**Упражнение 11.1.7.** Повторите упражнение 11.1.6 для матрицы из упражнения 11.1.5.

# 11.2. Метод главных компонент

Идея *метода главных компонент*, или PCA, заключается в том, чтобы для набора данных, который состоит из кортежей, представляющих точки в многомерном пространстве, наилучшим образом аппроксимировать направления расположения кортежей. Множество кортежей при этом рассматривается как матрица $M$, и мы ищем собственные векторы $MM^{\mathrm{T}}$ или $M^{\mathrm{T}}M$. Матрицу этих собственных векторов можно рассматривать как жесткое вращение многомерного пространства. Если применить это преобразование к исходным данным, то ось, соответствующая главному собственному вектору, определяет направление, вокруг которого разбросаны точки. Точнее, вдоль этой оси достигается максимум дисперсии данных. По-другому можно сказать, что множество точек вытянуто вдоль этой оси с небольшими отклонениями. Второму собственному вектору (со вторым по величине собственным значением), соответствует ось, для которой дисперсия расстояний от первой оси наибольшая, и т. д.

Метод главных компонент можно рассматривать как способ добычи данных. Данные высокой размерности заменяются проекцией на наиболее важные оси, а именно те, которые соответствуют наибольшим собственным значениям. Таким образом, исходные данные аппроксимируются данными гораздо меньшей размерности, и это можно считать своего рода обобщенным представлением исходных данных.

## 11.2.1. Иллюстративный пример

Начнем изложение с искусственного простого примера. Данные в нем двумерные, это слишком мало, чтобы получить от PCA реальную пользу. К тому же, данные, показанные на рис. 11.1, состоят всего из четырех точек, образующих простую фигуру, расположенную вдоль прямой, наклоненной под углом 45 градусов, чтобы проще было следить за вычислениями. Предвидя результат, можно сказать, что наилучшей осью данных является ось с углом наклона 45 градусов, и имеют место небольшие отклонения в перпендикулярном направлении.

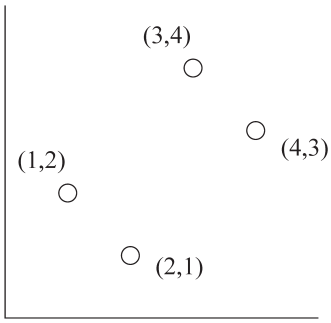



**Рис. 11.1.** Четыре точки в двумерном пространстве

Для начала представим точки матрицей $M$ с четырьмя строками – по одной для каждой точки – и двумя столбцами, соответствующими осям $x$ и $y$. Вот эта матрица:

$$M = \begin{bmatrix} 1 & 2 \\ 2 & 1 \\ 3 & 4 \\ 4 & 3 \end{bmatrix}.$$

Вычисляем $M^{\mathrm{T}}M$

$$M^T M = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 2 & 1 \\ 3 & 4 \\ 4 & 3 \end{bmatrix} = \begin{bmatrix} 30 & 28 \\ 28 & 30 \end{bmatrix}.$$

Мы можем найти собственные значения этой матрицы, решив уравнение

$$(30 - \lambda)(30 - \lambda) - 28 \times 28 = 0$$

как в примере 11.2. Его корни – $\lambda = 58$ и $\lambda = 2$.

Следуя процедуре, описанной в примере 11.2, мы должны решить уравнение

$$\begin{bmatrix} 30 & 28 \\ 28 & 30 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = 58 \begin{bmatrix} x \\ y \end{bmatrix}.$$

Умножив матрицу на вектор, получаем систему двух уравнений

$$30x+28y = 58x$$
$$28x+30y = 58y$$

Оба уравнения сводятся к одному: $x = y$. Следовательно, единичный собственный вектор, соответствующий главному собственному значению 58, равен

$$\begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}.$$

Для второго собственного значения 2 выполняем те же действия. Из матричного уравнения

$$\begin{bmatrix} 30 & 28 \\ 28 & 30 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = 2 \begin{bmatrix} x \\ y \end{bmatrix}.$$

получаем систему двух уравнений:

$$30x+28y = 2x$$
$$28x+30y = 2y$$

откуда находим, что $x = -y$. Следовательно, единичный собственный вектор, соответствующий собственному значению 2, равен

$$\begin{bmatrix} -1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}.$$

Мы обещали записывать собственные векторы, так чтобы первый элемент был положительным, но здесь примем противоположное соглашение, потому что так будет проще следить за преобразованием координат.

Теперь построим $E$ – матрицу собственных векторов для матрицы $M^{\mathrm{T}}M$. Если поставить главный собственный вектор на первое место, то эта матрица имеет вид:

$$E = \begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix}.$$

Любая матрица *ортонормальных векторов* (единичных взаимно ортогональных векторов) представляет вращение вокруг осей в евклидовом пространстве. Показанную выше матрицу можно рассматривать как описывающую вращение на 45 градусов против часовой стрелки. Для примера умножим матрицу $M$, представляющую все четыре точки на рис. 11.1, на $E$:

$$ME = \begin{bmatrix} 1 & 2 \\ 2 & 1 \\ 3 & 4 \\ 4 & 3 \end{bmatrix} \begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{bmatrix} = \begin{bmatrix} 3/\sqrt{2} & 1/\sqrt{2} \\ 3/\sqrt{2} & -1/\sqrt{2} \\ 7/\sqrt{2} & 1/\sqrt{2} \\ 7/\sqrt{2} & -1/\sqrt{2} \end{bmatrix}.$$

Как видим, первая точка, [1, 2], преобразована в точку

$$[3/\sqrt{2},\ 1/\sqrt{2}].$$

Взглянув на рис. 11.2, где штриховой линией показано новое положение оси $x$, мы увидим, что проекция первой точки на эту ось находится на расстоянии $3/\sqrt{2}$ от начала координат. Чтобы убедиться в этом, заметим, что проекция первой и второй точки в исходной системе координат имеет координаты [1.5, 1.5], а расстояние от этой точки до начала координат равно

$$\sqrt{(1.5)^2 + (1.5)^2} = \sqrt{9/2} = 3/\sqrt{2}\ .$$

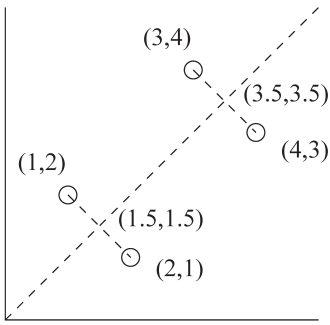



**Рис. 11.2.** Рисунок 11.1 после поворота осей на 45 градусов против часовой стрелки

Разумеется, новая ось $y$ перпендикулярна штриховой прямой. Первая точка находится на расстоянии $1/\sqrt{2}$ от новой оси $x$ в направлении оси $y$. Расстояние между точками [1, 2] и [1.5, 1.5] равно

$$\sqrt{(1-1.5)^2+(2-1.5)^2}=\sqrt{(-1/2)^2+(1/2)^2}=\sqrt{1/2}=1/\sqrt{2}$$

На рис. 11.3 показаны все четыре точки в повернутой системе координат.

Вторая точка, [2, 1], по случайному совпадению проецируется в ту же точку на новой оси $x$. Она расположена под этой осью и отстоит от нее на $1/\sqrt{2}$ в направлении новой оси $y$, и это подтверждается тем, что вторая строка в матрице преобразованных точек имеет вид $[3/\sqrt{2}, -1/\sqrt{2}]$. Третья точка, [3, 4], преобразуется в $[7/\sqrt{2}, 1/\sqrt{2}]$, а четвертая, [4, 3], – в $[7/\sqrt{2}, -1/\sqrt{2}]$. То есть обе они проецируются в одну точку на новой оси $x$, удаленную на расстояние $7/\sqrt{2}$ от начала координат, тогда как сами они расположены выше и ниже новой оси $x$ и отстоят от нее на расстояние $1/\sqrt{2}$ в направлении новой оси $y$.

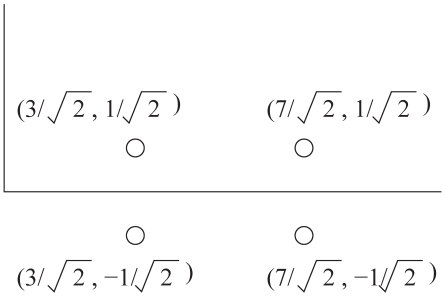



**Рис. 11.3.** Те же точки, что на рис. 1.1, в новой системе координат

## 11.2.2. Использование собственных векторов для понижения размерности

Из рассмотренного примера можно вывести общий принцип. Если $M$ – матрица, строки которой представляют точки в евклидовом пространстве произвольного числа измерений, то можно вычислить матрицу $M^{\mathrm{T}}M$ и ее собственные пары. Обозначим $E$ матрицу, столбцами которой являются собственные векторы, упорядоченные по убыванию собственных значений. Определим диагональную матрицу $L$, в которой на главной диагонали находятся собственные значения $M^{\mathrm{T}}M$ от наибольшего к наименьшему. Тогда, поскольку $M^{\mathrm{T}}M\mathbf{e} = \lambda\mathbf{e} = \mathbf{e}\lambda$ для любого собственного вектора $\mathbf{e}$ и соответствующего собственного значения $\lambda$, то $M^{\mathrm{T}}ME = EL$.

Мы заметили, что $ME$ содержит точки $M$, преобразованные в новую систему координат. В этой системе первая ось (соответствующая наибольшему собственному значению) самая важная; формально это означает, что дисперсия точек вдоль этой оси максимальна. Вторая ось, соответствующая второму собственному значению, – следующая по важности в том же смысле. И так далее для всех собственных пар. Если мы захотим преобразовать $M$ в пространство меньшей размерности, то больше всего информации удастся сохранить, выбрав собственные векторы, соответствующие наибольшим собственным значениям, и проигнорировав остальные.

Обозначим $E_k$ первые $k$ столбцов $E$. Тогда $ME_k$ – $k$-мерное представление $M$.

**Пример 11.6.** Пусть $M$ – матрица из раздела 11.2.1. Поскольку измерений всего два, то понизить размерность мы можем только до $k = 1$, т. е. спроецировать данные на одномерное пространство. Вычисляем матрицу $ME_1$

$$\begin{bmatrix}1 & 2\\ 2 & 1\\ 3 & 4\\ 4 & 3\end{bmatrix}\begin{bmatrix}1/\sqrt{2}\\ 1/\sqrt{2}\end{bmatrix}=\begin{bmatrix}3/\sqrt{2}\\ 3/\sqrt{2}\\ 7/\sqrt{2}\\ 7/\sqrt{2}\end{bmatrix}.$$

В результате этого преобразования все точки $M$ заменяются их проекциями на ось $x$ на рис. 11.3. И хотя две первые и две последние точки проецируются в одну и ту же точку, из всех одномерных представление это лучше всего отражает различия между точками.

## 11.2.3. Матрица расстояний

Вернемся к примеру из раздела 11.2.1, но начнем не с матрицы $M^{\mathrm{T}}M$, а исследуем собственные значения матрицы $MM^{\mathrm{T}}$. Поскольку в матрице $M$ строк больше, чем столбцов, последняя матрица больше первой, но если бы столбцов в $M$ было больше, чем строк, то она наоборот оказалась бы меньше. В нашем примере имеем

$$MM^{T}=\begin{bmatrix}1 & 2\\ 2 & 1\\ 3 & 4\\ 4 & 3\end{bmatrix}\begin{bmatrix}1 & 2 & 3 & 4\\ 2 & 1 & 4 & 3\end{bmatrix}=\begin{bmatrix}5 & 4 & 11 & 10\\ 4 & 5 & 10 & 11\\ 11 & 10 & 25 & 24\\ 10 & 11 & 24 & 25\end{bmatrix}.$$

Мы видим, что матрица $MM^{\mathrm{T}}$, как и $M^{\mathrm{T}}M$, симметрична. Элемент на пересечении $i$-й строки и $j$-го столбца имеет простую интерпретацию: это скалярное произведение векторов, представляющих $i$-ю и $j$-ю точку (строки $M$).

Существует связь между собственными значениями $M^{\mathrm{T}}M$ и $MM^{\mathrm{T}}$. Пусть $\mathbf{e}$ – собственный вектор $M^{\mathrm{T}}M$, т. е.

$$M^{\mathrm{T}}M\mathbf{e}=\lambda\mathbf{e}.$$

Умножим обе части этого равенства на $M$ слева. Тогда

$$MM^{\mathrm{T}}(M\mathbf{e})=M\lambda\mathrm{e}=\lambda(M\mathbf{e}).$$

Таким образом, если $M\mathbf{e}$ – не нулевой вектор $\mathbf{0}$, то он является собственным вектором $MM^{\mathrm{T}}$, а $\lambda$ – собственное значение не только $M^{\mathrm{T}}M$, но и $MM^{\mathrm{T}}$.

Обратное также верно. Если $\mathbf{e}$ – собственный вектор $MM^{\mathrm{T}}$, соответствующий собственному значению $\lambda$, то, умножив равенство $MM^{\mathrm{T}}\mathbf{e}=\lambda\mathbf{e}$ слева на $M^{\mathrm{T}}$, получим $M^{\mathrm{T}}M(M^{\mathrm{T}}\mathbf{e})=\lambda(M^{\mathrm{T}}\mathbf{e})$. Следовательно, если $M^{\mathrm{T}}\mathbf{e}$ не равно $\mathbf{0}$, то $\lambda$ является также собственным значением $M^{\mathrm{T}}M$.

А что, если $M^{\mathrm{T}}\mathbf{e}=0$? В этом случае $MM^{\mathrm{T}}\mathbf{e}$ также равно $\mathbf{0}$, но $\mathbf{e}$ не равен $\mathbf{0}$, потому что $\mathbf{0}$ не может быть собственным вектором. Но поскольку $\mathbf{0}=\lambda\mathbf{e}$, то $\lambda=0$.

Итак, мы заключаем, что множество собственных значений $MM^{\mathrm{T}}$ равно множеству собственных значений $M^{\mathrm{T}}M$ плюс дополнительные нули. Если бы размер-

ность $MM^{\mathrm{T}}$ была меньше размерности $M^{\mathrm{T}}M$, то было бы верно противоположное утверждение: множество собственных значений $M^{\mathrm{T}}M$ равно множеству собственных значений $MM^{\mathrm{T}}$ плюс дополнительные нули.

**Пример 11.7.** Множество собственных значений матрицы $MM^{\mathrm{T}}$ в нашем примере должно включать значения 58 и 2, потому что они являются собственными значениями $M^{\mathrm{T}}M$, как было показано в разделе 11.2.1. Поскольку размерность $MM^{\mathrm{T}}$ равна $4 \times 4$, то у нее есть еще два собственных значения, которые обязаны быть нулями. Матрица собственных векторов, соответствующая значениям 58, 2, 0, 0, показана на рис. 11.4.

$$\begin{bmatrix} 3/\sqrt{116} & 1/2 & 7/\sqrt{116} & 1/2 \\ 3/\sqrt{116} & -1/2 & 7/\sqrt{116} & -1/2 \\ 7/\sqrt{116} & 1/2 & -3/\sqrt{116} & -1/2 \\ 7/\sqrt{116} & -1/2 & -3/\sqrt{116} & 1/2 \end{bmatrix}$$

**Рис. 11.4.** Матрица собственных векторов $MM^{\mathrm{T}}$

## 11.2.4. Упражнения к разделу 11.2

**Упражнение 11.2.1.** Пусть $M$ – матрица

$$\begin{bmatrix} 1 & 1 \\ 2 & 4 \\ 3 & 9 \\ 4 & 16 \end{bmatrix}$$

(*а*) Вычислите $M^{\mathrm{T}}M$ и $MM^{\mathrm{T}}$.

(*б*) Вычислите собственные пары $M^{\mathrm{T}}M$.

! (*в*) Каковы должны быть собственные значения $MM^{\mathrm{T}}$?

! (*г*) Найдите собственные векторы $MM^{\mathrm{T}}$, соответствующие собственным значениям из п. (*в*).

! **Упражнение 11.2.2.** Докажите, что для любой матрицы $M$ матрицы $M^{\mathrm{T}}M$ и $MM^{\mathrm{T}}$ симметричны.

## 11.3. Сингулярное разложение

Рассмотрим теперь другой способ понижения размерности матрицы. Метод *сингулярного разложения* (SVD) дает точное представление матрицы и в то же время позволяет отбросить менее важные его части, оставив приближенное представление любой желаемой размерности. Разумеется, чем меньше выбранная размерность, тем менее точным будет представление.

Начнем с определений, а затем исследуем идею о том, что сингулярное разложение определяет небольшое число «концептов», связывающих строки и столбцы матрицы. Мы покажем, как исключение менее важных концептов дает меньшее представление, хорошо аппроксимирующее исходную матрицу. Затем мы увидим, что эти концепты можно использовать для более эффективного выполнения запросов к исходной матрице, и, наконец, опишем алгоритм вычисления самого сингулярного разложения.

## 11.3.1. Определение сингулярного разложения

Пусть $M$ – матрица размерности $m \times n$ ранга $r$. Напомним, что *рангом* матрицы называется наибольшее число строк (или – эквивалентно – столбцов) таких, что их линейная комбинация равна нулевому вектору тогда и только тогда, когда все коэффициенты равны нулю. Говорят, что эти строки или столбцы *линейно независимы*. Тогда $M$ можно представить в виде произведения матриц $U$, $\Sigma$ и $V$ (см. рис. 11.5), обладающих следующими свойствами:

1. $U$ – *ортонормальная по столбцам* матрица $m \times r$, т. е. все ее столбцы – единичные векторы и их попарные скалярные произведения равны 0.
2. $V$ – ортонормальная по столбцам матрица $n \times r$. Отметим, что всегда можно взять транспонированную к $V$ матрицу, которая будет ортонормальной по строкам.
3. $\Sigma$ – диагональная матрица, т. е. все элементы вне главной диагонали равны нулю. Элементы $\Sigma$ называются *сингулярными* значениями $M$.

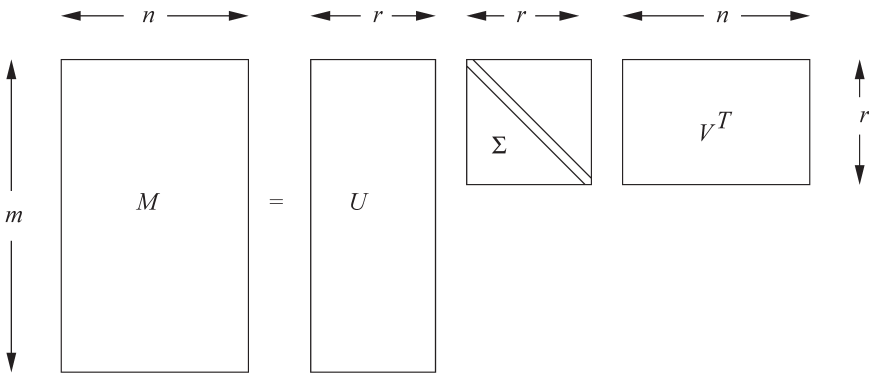



**Рис. 11.5.** Сингулярное разложение

**Пример 11.8.** На рис. 11.6 показана матрица ранга 2, представляющая оценки, поставленные фильмам пользователями. В этом искусственном примере за фильмами скрываются два «концепта»: научная фантастика и любовная драма. Все мальчики оценивают только научную фантастику, а все девочки – только драмы. Именно наличие двух строго разделенных концептов и дает матрицу ранга 2. Это означает, что мы можем выбрать одну

из первых четырех и одну из последних трех строк и убедиться, что никакая их линейная комбинация с ненулевыми коэффициентами не может быть равна 0. Но выбрать три линейно независимых строки не получится. Например, если взять строки 1, 2 и 7, то их линейная комбинация с коэффициентами 3, –1 и 0 дает **0**.

То же самое верно и для столбцов. Два столбца, один из первых трех, другой из последних двух, линейно независимы, но любые три столбца уже линейно зависимы.

Разложение матрицы *M* на рис. 11.6 в произведение *U*, Σ и *V* с точностью до двух знаков после запятой, показано на рис. 11.7. Поскольку ранг *M* равен 2, можно в этом разложении взять $r = 2$. Как вычисляется такое разложение, будет показано в разделе 11.3.6.

| | Матрица | Чужой | Звездные войны | Касабланка | Титаник |
|---|---|---|---|---|---|
| Джо | 1 | 1 | 1 | 0 | 0 |
| Джим | 3 | 3 | 3 | 0 | 0 |
| Джон | 4 | 4 | 4 | 0 | 0 |
| Джек | 5 | 5 | 5 | 0 | 0 |
| Джилл | 0 | 0 | 0 | 4 | 4 |
| Дженни | 0 | 0 | 0 | 5 | 5 |
| Джейн | 0 | 0 | 0 | 2 | 2 |

**Рис. 11.6.** Оценки, поставленные фильмам пользователями

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 3 & 3 & 3 & 0 & 0 \\ 4 & 4 & 4 & 0 & 0 \\ 5 & 5 & 5 & 0 & 0 \\ 0 & 0 & 0 & 4 & 4 \\ 0 & 0 & 0 & 5 & 5 \\ 0 & 0 & 0 & 2 & 2 \end{bmatrix} = \begin{bmatrix} .14 & 0 \\ .42 & 0 \\ .56 & 0 \\ .70 & 0 \\ 0 & .60 \\ 0 & .75 \\ 0 & .30 \end{bmatrix} \begin{bmatrix} 12.4 & 0 \\ 0 & 9.5 \end{bmatrix} \begin{bmatrix} .58 & .58 & .58 & 0 & 0 \\ 0 & 0 & 0 & .71 & .71 \end{bmatrix}$$

$M$ $U$ $\Sigma$ $V^T$

**Рис. 11.7.** Сингулярное разложение матрицы *M* на рис. 11.6

## 11.3.2. Интерпретация сингулярного разложения

Чтобы понять смысл сингулярного разложения, нужно представлять себе *r* столбцов матриц *U*, Σ и *V* как концепты, скрытые в исходной матрице *M*. В примере 11.8 эти концепты ясны: «научная фантастика» и «любовная драма». Будем считать, что строки *M* соответствуют пользователям, а столбцы – фильмам. Тогда матрица *U* связывает пользователей с концептами. Например, пользователю Джо, которому соответствует первая строка матрицы *M* на рис. 11.6, нравится только концепт «научная фантастика». Значение 0.14 на пересечении первой строки и первого столбца *U* меньше некоторых других значений в этом столбце, потому что, хотя Джо смотрит только научно-фантастические фильмы, не все он оценивает одинаково высоко. Во втором столбце первой строки находится 0, потому что любовные драмы Джо не оценивает вообще.

Матрица *V* соотносит фильмы с концептами. Значение 0.58 в первых трех столбцах первой строки $V^T$ означает, что первые три фильма – Матрица, Чужой и Звездные войны – относятся к жанру научной фантастики, а нули в последних

двух столбцах говорят, что с любовной драмой эти фильмы не имеют ничего общего. Аналогично вторая строка матрицы $V^T$ говорит, что Касабланка и Титаник – исключительно драмы.

Наконец, матрица $\Sigma$ определяет силу каждого концепта. В нашем примере сила концепта «научная фантастика» равна 12.4, а сила концепта «любовная драма» равна 9.5. Интуиция подсказывает, что первый концепт сильнее, потому что имеется больше информации о фильмах этого жанра и пользователях, которым они нравятся.

В общем случае концепты разграничены не так отчетливо. В матрицах $U$ и $V$ будет меньше нулей, хотя матрица $\Sigma$ всегда диагональная. Сущности, представленные строками и столбцами $M$ (аналоги пользователей и фильмов) дают вклад в несколько концептов, хотя и в разной степени. На самом деле, разложение в примере 11.8 такое простое, потому что ранг $M$ был равен желательному числу столбцов $U$, $\Sigma$ и $V$. Поэтому мы смогли найти точное разложение $M$, в котором каждая из трех матриц-сомножителей имеет всего два столбца; если вычислять произведение $U\Sigma V^T$ с неограниченной точностью, то получилась бы в точности $M$. На практике жизнь устроена сложнее. Если ранг $M$ больше желательного числа столбцов в матрицах $U$, $\Sigma$ и $V$, то разложение будет неточным. Для получения наилучшей аппроксимации необходимо исключить из точного разложения столбцы $U$ и $V$, соответствующие наименьшим сингулярным значениям. В следующем, немного измененном, примере иллюстрируется эта мысль.

| Матрица | Чужой | Звездные войны | Касабланка | Титаник |
|---|---|---|---|---|
| Джо | 1 | 1 | 1 | 0 | 0 |
| Джим | 3 | 3 | 3 | 0 | 0 |
| Джон | 4 | 4 | 4 | 0 | 0 |
| Джек | 5 | 5 | 5 | 0 | 0 |
| Джилл | 0 | 2 | 0 | 4 | 4 |
| Дженни | 0 | 0 | 0 | 5 | 5 |
| Джейн | 0 | 1 | 0 | 2 | 2 |

**Рис. 11.8.** Новая матрица $M'$, в которой фильм «Чужой» оценили еще два пользователя

**Пример 11.9.** Матрица на рис. 11.8 отличается от показанной на рис. 11.6 только тем, что Джилл и Джейн оценили фильм «Чужие», хотя и не очень высоко. Ранг этой матрицы равен 3; например, первая, шестая и седьмая строки линейно независимы, но легко проверить, что любые четыре строки уже линейно зависимы. На рис. 11.9 показано разложение матрицы на рис. 11.8.

$$\underset{M'}{\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 3 & 3 & 3 & 0 & 0 \\ 4 & 4 & 4 & 0 & 0 \\ 5 & 5 & 5 & 0 & 0 \\ 0 & 2 & 0 & 4 & 4 \\ 0 & 0 & 0 & 5 & 5 \\ 0 & 1 & 0 & 2 & 2 \end{bmatrix}} = \underset{U}{\begin{bmatrix} .13 & .02 & -.01 \\ .41 & .07 & -.03 \\ .55 & .09 & -.04 \\ .68 & .11 & -.05 \\ .15 & -.59 & .65 \\ .07 & -.73 & -.67 \\ .07 & -.29 & .32 \end{bmatrix}} \underset{\Sigma}{\begin{bmatrix} 12.4 & 0 & 0 \\ 0 & 9.5 & 0 \\ 0 & 0 & 1.3 \end{bmatrix}} \underset{V^T}{\begin{bmatrix} .56 & .59 & .56 & .09 & .09 \\ .12 & -.02 & .12 & -.69 & -.69 \\ .40 & -.80 & .40 & .09 & .09 \end{bmatrix}}$$

**Рис. 11.9.** Сингулярное разложение матрицы $M'$ на рис. 11.8

В матрицах $U$, $\Sigma$ и $V$ теперь по три столбца, потому что они составляют разложение матрицы ранга 3. Столбцы $U$ и $V$ по-прежнему соответствуют концептам: первый, как и раньше, «научной фантастике», второй – «любовной драме». Смысл третьего столбца объяснить труднее, но это и не важно, потому что его вес, определяемый третьим ненулевым элементом $\Sigma$, очень мал по сравнению с весами двух первых концептов.

В следующем разделе мы обсудим исключение наименее важных концептов. Например, в примере 11.9 можно было бы исключить третий концепт, потому что он мало что говорит, а тот факт, что соответствующее ему сингулярное значение так мало, подтверждает его незначительность.

## 11.3.3. Понижение размерности с помощью сингулярного разложения

Допустим, мы хотим представить очень большую матрицу $M$ компонентами ее сингулярного разложения $U$, $\Sigma$ и $V$, но и эти матрицы слишком велики для размещения в памяти. Наилучший способ понизить размерность всех трех матриц – положить наименьшие сингулярные значения равными 0. Тогда мы сможем устранить соответствующие им строки матриц $U$ и $V$.

**Пример 11.10.** В разложении из примера 11.9 имеется три сингулярных значения. Предположим, что мы хотим уменьшить число измерений до двух. Положим наименьшее сингулярное значение, 1.3, равным нулю. В результате третий столбец $U$ и третья строка $V^T$ на рис. 11.9 умножаются на одни нули, поэтому их можно было бы вообще исключить. На рис. 11.10 показана аппроксимация $M'$, полученная при использовании только двух наибольших сингулярных значений.

$$\begin{bmatrix} .13 & .02 \\ .41 & .07 \\ .58 & .09 \\ .68 & .11 \\ .15 & -.59 \\ .07 & -.73 \\ .07 & -.29 \end{bmatrix} \begin{bmatrix} 12.4 & 0 \\ 0 & 9.5 \end{bmatrix} \begin{bmatrix} .56 & .59 & .56 & .09 & .09 \\ .12 & -.02 & .12 & -.69 & -.69 \end{bmatrix} = \begin{bmatrix} 0.93 & 0.95 & 0.93 & .014 & .014 \\ 2.93 & 2.99 & 2.93 & .000 & .000 \\ 3.92 & 4.01 & 3.92 & .026 & .026 \\ 4.84 & 4.96 & 4.84 & .040 & .040 \\ 0.37 & 1.21 & 0.37 & 4.04 & 4.04 \\ 0.35 & 0.65 & 0.35 & 4.87 & 4.87 \\ 0.16 & 0.57 & 0.16 & 1.98 & 1.98 \end{bmatrix}$$

**Рис. 11.10.** Исключение наименьшего сингулярного значения из разложения на рис. 11.7

Результирующая матрица очень близка к матрице $M'$ на рис. 11.8. В идеале вся разница должна была бы сводиться только к эффекту обнуления последнего сингулярного значения. Но в этом простом примере различие в немалой степени обусловлено ошибками округления из-за того, что разложение $M'$ было построено с точностью до двух знаков после запятой.

**Сколько сингулярных значений оставлять?**

Существует полезное эвристическое правило: оставлять сингулярные значения, в которых сосредоточено 90 % *энергии* в $\Sigma$. Это означает, что сумма квадратов оставленных сингулярных значений должна быть не меньше 90 % суммы квадратов всех сингулярных значений. В примере 11.10 полная энергия равна $(12.4)^2 + (9.5)^2 + (1.3)^2 = 245.70$, а оставшаяся энергия – $(12.4)^2 + (9.5)^2 = 244.01$. Следовательно, мы оставили свыше 99 % энергии. Но если бы мы удалили и второе сингулярное значение 9.5, то остаток энергии составил бы только $(12.4)^2 / 245.70$, или приблизительно 63 %.

## 11.3.4. Почему обнуление малых сингулярных значений работает

Можно показать, что при обнулении наименьших сингулярных значений достигает минимума среднеквадратичная ошибка между исходной матрицей $M$ и ее аппроксимацией. Поскольку число элементов фиксировано, а извлечение квадратного корня – монотонная операция, то можно упростить и сравнить нормы Фробениуса участвующих матриц. Напомним, что *нормой Фробениуса* матрицы $M$, обозначаемой $\|M\|$, называется квадратный корень из суммы квадратов элементов $M$. Заметим, что если $M$ – разность между матрицей и ее аппроксимацией, то $\|M\|$ пропорциональна среднеквадратичной ошибке.

Чтобы объяснить, почему обнуление наименьших сингулярных значений минимизирует СКО, или норму Фробениуса разности $M$ и ее аппроксимации, вспомним некоторые сведения из матричной алгебры. Предположим, что $M$ – произведение трех матриц $M = PQR$. Обозначим $m_{ij}$, $p_{ij}$, $q_{ij}$, $r_{ij}$ элементы на пересечении строки $i$ и столбца $j$ матриц $M$, $P$, $Q$, $R$ соответственно. Тогда, по определению умножения матриц

$$m_{ij} = \sum_k \sum_\ell p_{ik} q_{k\ell} r_{\ell j}$$

Отсюда

$$\|M\|^2 = \sum_i \sum_j (m_{ij})^2 = \sum_i \sum_j \left( \sum_k \sum_\ell p_{ik} q_{k\ell} r_{\ell j} \right)^2 \qquad (11.1)$$

При возведении в квадрат суммы, как в правой части равенства 11.1, образуется две копии суммы (с разными индексами), и каждый член первой суммы умножается на каждый член второй:

$$\left( \sum_k \sum_\ell p_{ik} q_{k\ell} r_{\ell j} \right)^2 = \sum_k \sum_\ell \sum_m \sum_n p_{ik} q_{k\ell} r_{\ell j} p_{in} q_{nm} r_{mj}$$

Поэтому равенство (11.1) можно переписать в виде

$$\|M\|^2 = \sum_i \sum_j \sum_k \sum_\ell \sum_n \sum_m p_{ik} q_{k\ell} r_{\ell j} p_{in} q_{nm} r_{mj} \qquad (11.2)$$

Теперь рассмотрим случай, когда $P$, $Q$, $R$ образуют сингулярное разложение $M$. Тогда $P$ – ортонормальная по столбцам матрица, $Q$ – диагональная матрица, а $R$ – матрица, транспонированная к ортонормальной по столбцам. Следовательно, *$R$ ортонормальна по строкам*: ее строки являются единичными векторами, и попарные скалярные произведения различных строк равны 0. Поскольку $Q$ – диагональная матрица, элементы $q_{kl}$ и $q_{nm}$ равны 0 во всех случаях, кроме $k = l$ и $n = m$. Поэтому суммирование по $l$ и по $m$ в равенстве (11.2) можно опустить и положить $k = l$, $n = m$. Тогда равенство (11.2) принимает вид

$$\|M\|^2 = \sum_i \sum_j \sum_k \sum_n p_{ik} q_{kk} r_{kj} p_{in} q_{nn} r_{nj} \qquad (11.3)$$

Теперь переупорядочим слагаемые, так чтобы суммирование по $i$ было внутренним. В равенства (11.3) индекс $i$ входит только в сомножители $p_{ik}$ и $p_{in}$, все остальные сомножители являются константами относительно суммирования по $i$. Поскольку $P$ ортонормальна по столбцам, то $\sum_i p_{ik} p_{in}$ равна 1, если $k = n$, и 0 в противном случае. Следовательно, в равенствах (11.3) можно положить $k = n$, опустить сомножители $p_{ik}$ и $p_{in}$ и исключить суммирование по $i$ и $n$. В результате получим

$$\|M\|^2 = \sum_j \sum_k q_{kk} r_{kj} q_{kk} r_{kj} \qquad (11.4)$$

Так как $R$ ортонормальна по строкам, $\sum_j r_{kj} r_{kj}$ равна 1. Следовательно, можно исключить члены $r_{kj}$ и суммирование по $j$, и в результате останется очень простое выражение для нормы Фробениуса:

$$\|M\|^2 = \sum_k (q_{kk})^2 \qquad (11.5)$$

Далее применим эту формулу к матрице $M$ с сингулярным разложением $M = U\Sigma V^{\mathrm{T}}$. Обозначим $\sigma_i$ $i$-й элемент на диагонали $\Sigma$ и предположим, что мы оставляем первые $n$ из $r$ диагональных элементов $\Sigma$, а остальные полагаем равными 0. Обозначим $\Sigma'$ получившуюся в результате диагональную матрицу. Пусть $M' = U\Sigma' V^{\mathrm{T}}$ – получившаяся аппроксимация $M$. Тогда $M - M' = U(\Sigma - \Sigma')V^{\mathrm{T}}$ – матрица, определяющая погрешность аппроксимации.

Применив равенство (11.5) к матрице $M - M'$, мы увидим, что величина $\|M - M'\|^2$ равна сумме квадратов диагональных элементов $\Sigma - \Sigma'$. Но в матрице $\Sigma - \Sigma'$ первые $n$ диагональных элементов равны 0, а $i$-й диагональный элемент для $n < i \le r$ равен $\sigma_i$. Следовательно, $\|M - M'\|^2$ равно сумме квадратов тех элементов $\Sigma$, которые мы положили равными 0. Для минимизации $\|M - M'\|^2$ выберем наименьшие элементы $\Sigma$. При этом получится наименьшее возможное значение $\|M - M'\|^2$ при условии, что оставлены $n$ диагональных элементов и, следовательно, достигается минимум СКО при том же ограничении.

### 11.3.5. Запросы с использованием концептов

В этом разделе мы покажем, как сингулярное разложение позволяет эффективно отвечать на некоторые вопросы с приемлемой точностью. Предположим, что исходная матрица оценок фильмов (матрица ранга 2 на рис. 11.6) разложена, как показано на рис. 11.7. Квинси не входит в число пользователей, присутствующих в исходной матрице, но хочет с помощью нашей системы узнать, какие фильмы ему понравились бы. Он смотрел только один фильм, «Матрица», и поставил ему оценку 4. Поэтому Квинси можно представить вектором $\mathbf{q} = [4, 0, 0, 0, 0]$, как будто это одна из строк исходной матрицы.

Если бы мы применили коллаборативную фильтрацию, то должны были бы сравнить Квинси с другими пользователями, представленными в исходной матрице $M$. Вместо этого мы отобразим Квинси в «пространство концептов», умножив его на матрицу $V$. Получаем $\mathbf{q}V = [2.32, 0]$[3]. Это означает, что Квинси сильно интересуется научной фантастикой и совсем не интересуется любовными драмами. Теперь у нас есть представление Квинси в пространстве концептов, полученное на основе его представления в исходном «пространстве фильмов», но отличное от него. Отобразим это представление обратно в пространство фильмов, умножив вектор [2.32, 0] на $V^{\mathrm{T}}$. Произведение равно [1.35, 1.35, 1.35, 0, 0]. Это позволяет предположить, что Квинси понравились бы фильмы «Чужой» и «Звездные войны», но не «Касабланка» или «Титаник».

Другой вид запросов в пространстве концептов – найти пользователей, похожих на Квинси. С помощью матрицы $V$ мы можем отобразить всех пользователей в пространство концептов. Например, Джо переходит в [1.74, 0], а Джилл – в [0, 5.68]. Отметим, что в этом простом примере каждый пользователь любит только фантастику, либо только любовные драмы, поэтому в каждом векторе один элемент равен 0. В действительности вкусы людей сложнее, они проявляют ненулевой, хотя и не одинаковый интерес к разным концептам. В общем случае измерить сходство пользователей можно с помощью косинусного расстояния в пространстве концептов.

> **Пример 11.11.** В рассматриваемом случае заметим, что векторы концептов для Квинси и Джо равны [2.32, 0] и [1.74, 0]; они не одинаковы, но направлены в одну сторону. Поэтому косинусное расстояние между ними равно 0. С другой стороны, векторы для Квинси и Джилл равны соответственно [2.32, 0] и [0, 5.68]. Их скалярное произведение равно 0, а, значит, угол между ними равен 90 градусов, т. е. косинусное расстояние максимально – равно 1.

### 11.3.6. Вычисление сингулярного разложения матрицы

Сингулярное разложение матрицы $M$ тесно связано с собственными значениями симметричных матриц $M^{\mathrm{T}}M$ и $MM^{\mathrm{T}}$. Эта связь позволяет получить сингулярное

[3] Напомним, что $M^i$ означает умножение матрицы $M$ на себя $i$ раз.

разложение $M$, зная собственные пары этих матриц. Предположим, что имеется сингулярное разложение $M = U\Sigma V^{\mathrm{T}}$. Тогда

$$M^{\mathrm{T}} = (U\Sigma V^{\mathrm{T}})^{\mathrm{T}} = (V^{\mathrm{T}})^{\mathrm{T}}\Sigma^{\mathrm{T}}U^{\mathrm{T}} = V\Sigma^{\mathrm{T}}U^{\mathrm{T}}.$$

Поскольку $\Sigma$ – диагональная матрица, транспонирование оставляет ее неизменной, поэтому $M^{\mathrm{T}} = V\Sigma U^{\mathrm{T}}$.

Далее $M^{\mathrm{T}}M = V\Sigma U^{\mathrm{T}}U\Sigma V^{\mathrm{T}}$. Вспомним, что $U$ – ортонормальная матрица, поэтому $U^{\mathrm{T}}U$ – единичная матрица соответствующей размерности. Следовательно

$$M^{\mathrm{T}}M = V\Sigma^2V^{\mathrm{T}}.$$

Умножая обе части на $V$ слева, получаем

$$M^{\mathrm{T}}MV = V\Sigma^2V^{\mathrm{T}}V.$$

Поскольку $V$ – также – ортонормальная матрица, $V^{\mathrm{T}}V$ – единичная матрица. Таким образом

$$M^{\mathrm{T}}MV = V\Sigma^2 \qquad (11.6)$$

Так как $\Sigma$ – диагональная матрица, то $\Sigma^2$ также диагональна, и элемент на пересечении $i$-й строки и $i$-го столбца равен квадрату соответственного элемента $\Sigma$. Равенство (11.6) нам уже знакомо. Оно означает, что $V$ – матрица собственных векторов $M$T$M$, а элементами диагональной матрицы $\Sigma^2$ являются собственные значения.

Таким образом, алгоритм, который вычисляет собственные пары $M^{\mathrm{T}}M$, дает также и матрицу $V$ в сингулярном разложении $M$, а заодно и сингулярные значения для этого разложения – нужно лишь извлечь квадратные корни из собственных значений $M^{\mathrm{T}}M$.

Осталось вычислить только матрицу $U$, но ее можно найти точно так же, как $V$. Начнем с того, что

$$MM^{\mathrm{T}} = U\Sigma V^{\mathrm{T}}(U\Sigma V^{\mathrm{T}})^{\mathrm{T}} = U\Sigma V^{\mathrm{T}}V\Sigma U^{\mathrm{T}} = U\Sigma^2U^{\mathrm{T}}.$$

Затем посредством таких же манипуляций, как и выше, приходим к выводу, что

$$MM^{\mathrm{T}}U = U\Sigma^2.$$

Следовательно, $U$ – матрица собственных векторов $MM^{\mathrm{T}}$.

Осталось объяснить небольшую деталь. В каждой из матриц $U$ и $V$ по $r$ столбцов, тогда как $M^{\mathrm{T}}M$ – матрица размерности $n \times n$, а $MM^{\mathrm{T}}$ – размерности $m \times m$. И $n$, и $m$ не меньше $r$. Следовательно, у $M^{\mathrm{T}}M$ и $MM^{\mathrm{T}}$ должно быть соответственно $n \times r$ и $m \times r$ дополнительных собственных пар, которые не отражены в $U$, $V$ и $\Sigma$. Но поскольку ранг $M$ равен $r$, все остальные собственные значения равны 0, так что пользы от них нет.

## 11.3.7. Упражнения к разделу 11.3

**Упражнение 11.3.1.** На рис. 11.11 показана матрица $M$. Ее ранг равен 2, в чем легко убедиться, заметив, что первый столбец плюс третий минус удвоенный второй дает 0.

(*а*) Вычислите матрицы $M^{\mathrm{T}}M$ и $MM^{\mathrm{T}}$.

! (*б*) Найдите собственные значения матриц, вычисленных в п. (*а*).

(*в*) Найдите собственные векторы матриц, вычисленных в п. (*а*).

(*г*) Найдите сингулярное разложение исходной матрицы $M$, зная результаты выполнения пунктов (*а*) и (*б*). Обратите внимание, что есть только два ненулевых собственных значения, поэтому в матрице $\Sigma$ должно быть только два сингулярных значения, а в матрицах $U$ и $V$ – по два столбца.

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 5 & 4 & 3 \\ 0 & 2 & 4 \\ 1 & 3 & 5 \end{bmatrix}$$

**Рис. 11.11.** Матрица $M$ для упражнения 11.3.1

(*д*) Положите наименьшее сингулярное значение равным 0 и вычислите одномерную аппроксимацию матрицы $M$ на рис. 11.11.

(*е*) Сколько энергии, заключенной в исходных сингулярных значениях, остается после одномерной аппроксимации?

**Упражнение 11.3.2.** Воспользуемся сингулярным разложением, показанным на рис. 11.7. Предположим, что Лесли поставила оценку 3 фильму «Чужой» и оценку 4 «Титанику», т. е. представление Лесли в «пространстве фильмов» – [0, 3, 0, 0, 4]. Найдите представление Лесли в пространстве концептов. Что это представление говорит о том, какие оценки Лесли поставила бы другим фильмам, включенным в наши данные?

! **Упражнение 11.3.3.** Докажите, что ранг матрицы на рис. 11.8 равен 3.

! **Упражнение 11.3.4.** В разделе 11.3.5 показано, как предсказать, какие фильмы больше всего понравились бы пользователю. Как бы вы применили аналогичную технику, чтобы предсказать, каким пользователям понравился бы данный фильм, не имея никакой другой информации, кроме оценок, поставленных этому фильму несколькими пользователями?

# 11.4. CUR-декомпозиция

С сингулярным разложением связана одна проблема, не проявляющаяся в простом примере из раздела 11.3. Когда данных много, исходная матрица $M$ обычно сильно разрежена, т. е. большинство ее элементов равно 0. Например, матрица, представляющая много документов (строки) и встречающиеся в них слова (столбцы), будет разреженной, потому что большинство слов не встречается в большинстве документов. Точно так же, разреженной будет матрица покупателей и товаров, потому что большинство людей не покупает большинство товаров.

Мы не можем обработать плотные матрицы с миллионами или миллиардами строки и (или) столбцов. Но в сингулярном разложении разреженной матрицы $M$ матрицы $U$ и $V$ будут плотными[4]. Матрица $\Sigma$ диагональная, поэтому она, конеч-

[4] Отметим, что стохастическая матрица, вообще говоря, не обязана быть симметричной. Симметричные и стохастические матрицы – два класса матриц, для которых собственные пары существуют и имеют полезные применения. В этой главе мы рассматриваем методы, относящиеся к симметричным матрицам.

но, разрежена, но, поскольку она гораздо меньше, чем $U$ и $V$, ее разреженность не играет особой роли.

В этом разделе мы рассмотрим еще один подход к разложению матрицы – CUR-декомпозицию. Его достоинство заключается в том, что если $M$ разрежена, то обе большие матрицы (они называются $C$ – от слова «columns», столбцы, и $R$ – от слова «rows», строки), являющиеся аналогами $U$ и $V$ в сингулярном разложении, также разреженные. Лишь средняя матрица (аналог $\Sigma$ в сингулярном разложении) плотная, но в силу ее малости плотность не так страшна.

В отличие от сингулярного разложения, которое является точным, если параметр $r$ не меньше ранга матрицы $M$, CUR-декомпозиция дает лишь аппроксимацию при любом значении $r$. Теоретически можно доказать, что при возрастании $r$ эта аппроксимация стремится к $M$, но на практике $r$ для этого нужно выбирать таким большим, порядка 1 %, что весь метод теряет смысл. Тем не менее, вероятность, что декомпозиция с относительно небольшим значением $r$ окажется полезной и достаточно точной, довольно велика.

**Почему псевдообращение работает**

Предположим, что $M$ представлена в виде произведения трех матриц $XZY$. Если для каждой существует обратная матрица, то, как известно, $M^{-1} = Y^{-1}Z^{-1}X^{-1}$. Поскольку в интересующем нас случае $XZY$ – сингулярное разложение, то $X$ ортонормальна по столбцам, а $Y$ ортонормальна по строкам. В обоих случаях обратная матрица совпадает с транспонированной, т. е. $XX^{T}$ – единичная матрица соответствующего размера, как и $YY^{T}$. Таким образом, $M^{-1} = Y^{T}Z^{-1}X^{T}$.

Мы знаем также, что $Z$ – диагональная матрица. Если на ее главной диагонали нет нулей, то $Z^{-1}$ получается из $Z$ обращением всех диагональных элементов. И лишь когда на диагонали $Z$ встречаются нули, мы не сможем найти такой элемент в соответственной позиции обратной матрицы, чтобы умножение ее на $Z$ давало единичную матрицу. Поэтому приходится прибегать к «псевдообращению», смирившись с тем фактом, что $ZZ^{+}$ является не единичной матрицей, а диагональной, в которой ненулевым элементам $Z$ соответствует 1, а нулевым – 0.

## 11.4.1. Определение CUR-декомпозиции

Пусть $M$ – матрица с $m$ строками и $n$ столбцами. Выберем $r$ – количество «концептов» в разложении. *CUR-декомпозицией* матрицы $M$ называется случайно выбранное множество $r$ столбцов $M$, образующих матрицу $C$ размерности $m \times r$, случайно выбранное множество $r$ строк матрицы $M$, образующих матрицу $R$ размерности $r \times n$. Матрица $U$ размерности $r \times r$ строится по $C$ и $R$ следующим образом:

1. Пусть $W$ – матрица размерности $r \times r$, образованная пересечением столбцов $C$ и строк $R$, т. е. элемент на пересечении строки $i$ и столбца $j$ в матрице $W$ равен элементу $M$, столбец которого совпадает с $j$-ым столбцом $C$, а строка – с $i$-й строкой $R$.
2. Вычислим сингулярное разложение $W = X\Sigma Y^{\mathrm{T}}$.
3. Вычислим матрицу $\Sigma^{+}$ – *псевдообратную матрицу Мура-Пенроуза* для диагональной матрицы $\Sigma$. То есть, если $i$-й диагональный элемент $\Sigma\ \sigma \neq 0$, то заменяем его элементом $1/\sigma$. Если же $i$-й элемент равен 0, то оставляем его без изменения.
4. Полагаем $U = Y\,(\Sigma^{+})^2 X^{\mathrm{T}}$.

Пример, иллюстрирующий весь процесс построения *CUR*-декомпозиции, мы приведем в разделе 11.4.3 и там же остановимся на важном вопросе о том, как выбирать матрицы $C$ и $R$ для получения хорошей аппроксимации $M$.

## 11.4.2. Правильный выбор строк и столбцов

Напомним, что строки и столбцы выбираются случайным образом. Однако выбор должен быть смещенным, чтобы у более важных строк и столбцов было больше шансов оказаться выбранными. В качестве меры важности мы должны использовать квадрат нормы Фробениуса, т. е. сумму квадратов элементов строк или столбца. Обозначим $f = \sum_{ij} m_{ij}^2$, квадрат нормы Фробениуса матрицы $M$. Тогда вероятность $p_i$ выбора строки $i$ равна $\sum_j m_{ij}^2/f$, а вероятность $q_j$ выбора столбца $j$ равна $\sum_i m_{ij}^2/f$.

**Пример 11.12.** Вернемся к матрице $M$ на рис. 11.6, которая для удобства повторена на рис. 11.12. Сумма квадратов элементов $M$ равна 243. Для всех трех столбцов, соответствующих научно-фантастическим фильмам, «Матрица», «Чужой» и «Звездные войны», квадрат нормы Фробениуса равен $1^2 + 3^2 + 4^2 + 5^2 = 51$, поэтому вероятность выбора каждого равна $51/243 = 0.210$. Для каждого из оставшихся двух столбцов квадрат нормы Фробениуса равен $4^2 + 5^2 + 2^2 = 45$, а вероятность выбора $45/243 = 0.185$.

У семи строк $M$ квадрат нормы Фробениуса равен соответственно 3, 27, 48, 75, 32, 50 и 8. Поэтому вероятности их выбора равны 0.012, 0.111, 0.198, 0.309, 0.132, 0.206 и 0.033.

| | Матрица | Чужой | Звездные войны | Касабланка | Титаник |
|---|---|---|---|---|---|
| Джо | 1 | 1 | 1 | 0 | 0 |
| Джим | 3 | 3 | 3 | 0 | 0 |
| Джон | 4 | 4 | 4 | 0 | 0 |
| Джек | 5 | 5 | 5 | 0 | 0 |
| Джилл | 0 | 0 | 0 | 4 | 4 |
| Дженни | 0 | 0 | 0 | 5 | 5 |
| Джейн | 0 | 0 | 0 | 2 | 2 |

**Рис. 11.12.** Та же матрица $M$, что на рис. 11.6

Теперь выберем $r$ столбцов, образующих матрицу $C$. Каждый столбец случайным образом выбирается из столбцов $M$. Но распределение не равномерное: $j$-й

столбец выбирается с вероятностью $q_j$. Напомним, что эта вероятность равна сумме квадратов элементов данного столбца, поделенной на сумму квадратов всех элементов матрицы. Каждый столбец $C$ выбирается независимо из всего множества столбцов $M$, поэтому есть шанс, что какой-то столбец будет выбран более одного раза. Как поступать в такой ситуации мы обсудим после того, как объясним основы CUR-декомпозиции.

Выбрав столбцы из $M$, масштабируем каждый, разделив его элементы на квадратный корень из математического ожидания количества выборов данного столбца. Иначе говоря, если $j$-й столбец $M$ был выбран, то делим его элементы на $\sqrt{rq_j}$. Масштабированный столбец $M$ становится столбцом $C$.

Строки $M$, образующие $R$, выбираются аналогично. Строка $i$ матрицы $M$ выбирается с вероятностью $p_i$. Напомним, что вероятность $p_i$ равна сумме квадратов элементов $i$-й строки, поделенной на сумму квадратов всех элементов $M$. Затем масштабируем выбранную строку, разделив ее элементы на $\sqrt{rp_i}$, если была выбрана $i$-я строка $M$.

**Пример 11.13.** Пусть для нашей CUR-декомпозиции $r = 2$. Предположим, что в результате случайного выбора столбцов из матрицы $M$ на рис. 11.12 первым был выбран столбец «Чужой» (второй по порядку), а затем «Касабланка» (четвертый по порядку). Столбец «Чужой» равен $[1, 3, 4, 5, 0, 0, 0]^T$, и мы должны масштабировать его, разделив на $\sqrt{rp_2}$. Напомним (пример 11.12), что со столбцом «Чужой» ассоциирована вероятность 0.210, поэтому мы делим на $\sqrt{2\times 0.210} = 0.648$. С точностью до двух знаков после запятой получаем $[1.54, 4.63, 6.17, 7.72, 0, 0, 0]^T$. Этот столбец становится первым в матрице $C$.

Для построения второго столбца $C$ берем столбец «Касабланка», равный $[0, 0, 0, 0, 4, 5, 2]^T$, и делим его на $\sqrt{rp_4} = \sqrt{2\times 0.185} = 0.430$. Следовательно, второй столбец $C$ с точностью до двух знаков после запятой равен $[0, 0, 0, 0, 9.30, 11.63, 4.65]^T$.

Теперь выберем строки $R$. С наибольшей вероятностью будут выбраны строки, соответствующие Дженни и Джеку, поэтому предположим, что они и были выбраны, причем Дженни сначала. Таким образом, немасштабированные строки $R$ равны

$$\begin{bmatrix} 0 & 0 & 0 & 5 & 5 \\ 5 & 5 & 5 & 0 & 0 \end{bmatrix}.$$

Чтобы масштабировать строку Дженни, заметим, что с ней ассоциирована вероятность 0.206, поэтому делить нужно на $\sqrt{2\times 0.206} = 0.454$. Вероятность, ассоциированная со строкой Джека, равна 0.309, поэтому делим на $\sqrt{2\times 0.309} = 0.556$. Таким образом, мы нашли матрицу $R$

$$\begin{bmatrix} 0 & 0 & 0 & 11.01 & 11.01 \\ 8.99 & 8.99 & 8.99 & 0 & 0 \end{bmatrix}.$$

### 11.4.3. Построение средней матрицы

Нам осталось построить матрицу $U$, расположенную между $C$ и $R$ в декомпозиции. Напомним, что $U$ – матрица размерности $r \times r$. Начнем построение $U$ с другой матрицы той же размерности, которую обозначим $W$. На пересечении строки $i$ и столбца $j$ матрицы $W$ находится элемент матрицы $M$, строка которого совпадает со строкой, ставшей $i$-й строкой $R$, а столбец совпадает со столбцом, ставшим $j$-ым столбцом $C$.

**Пример 11.14.** Продолжим работать со строками и столбцами, выбранными в примере 11.13. Матрица $W$ выглядит следующим образом

$$W = \begin{bmatrix} 0 & 5 \\ 5 & 0 \end{bmatrix}.$$

Первая строка $W$ соответствует первой строке $R$, т. е. строке Дженни матрицы $M$ на рис. 11.12. В первом столбце находится 0, потому что именно такое значение стоит на пересечении строки Дженни и столбца «Чужой» в $M$ (напомним, что первый столбец $C$ построен по столбцу «Чужой»). Число 5 во втором столбце отражает тот факт, что именно это значение находится на пересечении строки Дженни и столбца «Касабланка» в $M$ (второй столбец $C$ построен по столбцу «Касабланка»). Аналогично вторая строка $W$ содержит элементы на пересечении строки Джек и столбцов «Чужой» и «Касабланка».

Матрица $U$ построена из $W$ путем псевдообращения Мура-Пенроуза, описанного в разделе 11.4.1. Мы должны вычислить сингулярное разложение $W$, допустим $W = X\Sigma Y^{\mathrm{T}}$, и заменить все ненулевые элементы матрицы сингулярных значений $\Sigma$ обратными к ним. В результате получим псевдообратную матрицу $\Sigma^{+}$, а затем положим $U = Y(\Sigma^{+})^2 X^{\mathrm{T}}$.

**Пример 11.15.** Построим $U$ по матрице $W$ из примера 11.14. Сначала вычисляем сингулярное разложение $W$:

$$W = \begin{bmatrix} 0 & 5 \\ 5 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 5 & 0 \\ 0 & 5 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Три матрицы в правой части – это соответственно $X$, $\Sigma$ и $Y^{\mathrm{T}}$. Среди диагональных элементов матрицы $\Sigma$ нет нулей, поэтому для получения псевдообратной матрицы Мура-Пенроуза заменяем каждый элемент обратным ему:

$$\Sigma^{+} = \begin{bmatrix} 1/5 & 0 \\ 0 & 1/5 \end{bmatrix}.$$

Поскольку матрицы $X$ и $Y$ симметричны, они совпадают с транспонированными. Следовательно

$$U = Y(\Sigma^{+})^2 X^{T} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1/5 & 0 \\ 0 & 1/5 \end{bmatrix}^2 \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1/25 \\ 1/25 & 0 \end{bmatrix}.$$

## 11.4.4. Полная CUR-декомпозиция

Итак, у нас есть метод случайного выбора трех матриц $C$, $U$ и $R$. Их произведение аппроксимирует исходную матрицу $M$. В начале обсуждения мы сказали, что близость этой аппроксимации формально гарантируется только при выборе очень большого числа строк и столбцов. Но интуиция подсказывает, что если выбирать «важные» строки и столбцы (с большой нормой Фробениуса), то мы сможем выделить существенные части исходной матрицы даже при небольшом числе строк и столбцов. В качестве примера посмотрим, что получится в сквозном примере из этого раздела.

**Пример 11.16.** Декомпозиция матрицы из нашего примера показана на рис. 11.13. И хотя существует значительная разница между этим результатом и исходной матрицей $M$, особенно в части научной фантастики, все же полученные значения пропорциональны истинным. Этот пример слишком мал, небольшое число строк и столбцов мы выбирали не столько случайно, сколько произвольно, поэтому ожидать сходимости CUR-декомпозиции к точным значениям не приходится.

$$CUR = \begin{bmatrix} 1.54 & 0 \\ 4.63 & 0 \\ 6.17 & 0 \\ 7.72 & 0 \\ 0 & 9.30 \\ 0 & 11.63 \\ 0 & 4.65 \end{bmatrix} \begin{bmatrix} 0 & 1/25 \\ 1/25 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 11.01 & 11.01 \\ 8.99 & 8.99 & 8.99 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0.55 & 0.55 & 0.55 & 0 & 0 \\ 1.67 & 1.67 & 1.67 & 0 & 0 \\ 2.22 & 2.22 & 2.22 & 0 & 0 \\ 2.78 & 2.78 & 2.78 & 0 & 0 \\ 0 & 0 & 0 & 4.10 & 4.10 \\ 0 & 0 & 0 & 5.12 & 5.12 \\ 0 & 0 & 0 & 2.05 & 2.05 \end{bmatrix}$$

**Рис. 11.13.** CUR-декомпозиция матрицы на рис. 11.12

## 11.4.5. Исключение дубликатов строк и столбцов

Вполне возможно, что какая-то строка или столбец будут выбраны более одного раза. Особого вреда в этом нет, хотя ранг матриц, образующих декомпозицию, будет меньше, чем количество выбранных строк и столбцов. Однако можно также объединить $k$ строк $R$, являющихся одной и той же строкой $M$, в одну строку, уменьшив тем самым количество строк в $R$. Точно так же можно поступить с $k$ одинаковыми столбцами $C$. Но, будь то строки или столбцы, оставшийся после объединения единственный вектор следует умножить на $\sqrt{k}$.

При объединении некоторых строк или столбцов может оказаться, что в $R$ меньше строк, чем в $C$ столбцов, или наоборот. В результате матрица $W$ не будет квадратной. Однако вычислить псевдообратную к ней все равно можно, если разложить ее в произведение $W = X\Sigma Y^{\mathrm{T}}$, где $\Sigma$ – диагональная матрица, некоторые строки или столбцы (в зависимости от того, чем больше) которой состоят из одних нулей. Чтобы вычислить для такой диагональной матрицы псевдообратную, нужно обработать все диагональные элементы как обычно (для ненулевых взять обратные, а нулевые оставить как есть), а затем транспонировать результат.

**Пример 11.17.** Предположим, что

$$\Sigma = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 \end{bmatrix}.$$

Тогда

$$\Sigma^{+} = \begin{bmatrix} 1/2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1/3 \\ 0 & 0 & 0 \end{bmatrix}.$$

### 11.4.6. Упражнения к разделу 11.4

**Упражнение 11.4.1.** Сингулярное разложение матрицы

$$M = \begin{bmatrix} 48 & 14 \\ 14 & -48 \end{bmatrix}$$

имеет вид

$$\begin{bmatrix} 48 & 14 \\ 14 & -48 \end{bmatrix} = \begin{bmatrix} 3/5 & 4/5 \\ 4/5 & -3/5 \end{bmatrix} \begin{bmatrix} 50 & 0 \\ 0 & 25 \end{bmatrix} \begin{bmatrix} 4/5 & -3/5 \\ 3/5 & 4/5 \end{bmatrix}.$$

Найдите псевдообратную матрицу Мура-Пенроуза для $M$.

! **Упражнение 11.4.2.** Найдите CUR-декомпозицию матрицы на рис. 11.12, если пары «случайных» строк и столбцов выбираются следующим образом:

(*а*) Столбцы – «Матрица» и «Чужой», строки – Джим и Джон.
(*б*) Столбцы – «Чужой и «Звездные войны», строки – Джек и Джилл.
(*в*) Столбцы – «Матрица» и «Титаник», строки – Джо и Джейн.

! **Упражнение 11.4.3.** Найдите CUR-декомпозицию матрицы на рис. 11.12, если обе выбранные «случайные» строки – Джек, а два столбца – «Звездные войны» и «Касабланка».

## 11.5. Резюме

- *Понижение размерности.* Цель понижения размерности – заменить большую матрицу двумя или более матрицами гораздо меньшей размерности, так чтобы по ним можно было приближенно реконструировать исходную – обычно путем перемножения.
- *Собственные значения и собственные векторы.* У матрицы может быть несколько собственных векторов – таких, что результатом умножения матрицы на вектор является этот же вектор, умноженный на константу. Эта

константа называется собственным значением, ассоциированным с собственным вектором. Вместе собственный вектор и собственное значение называются собственной парой.

- *Нахождение собственных пар степенным методом.* Для нахождения главного собственного вектора (с наибольшим собственным значением) мы можем начать с произвольного вектора и, умножая матрицу на текущий вектор, получать новый. Когда следующий вектор будет мало отличаться от предыдущего, мы можем считать, что последний результат является аппроксимацией главного собственного вектора. Несколько модифицировав матрицу, мы можем с помощью такого же процесса получить вторую собственную пару (со вторым по величине собственным значением), а за ней и все остальные в порядке убывания собственных значений.
- *Метод главных компонент.* В этом методе понижения размерности данные, представляющие собой множество точек в многомерном пространстве, рассматриваются как матрица, строки которой соответствуют точкам, а столбцы – измерениям. У произведения этой матрицы на транспонированную к ней имеются собственные пары, а главный собственный вектор можно рассматривать как направление, вдоль которого точки выстраиваются наилучшим образом. Второй собственный вектор определяет направление, в котором отклонение от главного собственного вектора наибольшее и т. д.
- *Понижение размерности методом главных компонент.* Представляя матрицу небольшим числом ее собственных векторов, мы можем аппроксимировать данные, так чтобы минимизировать среднеквадратичную ошибку для заданного числа столбцов в представляющей матрице.
- *Сингулярное разложение.* Сингулярное разложение матрицы состоит из трех матриц $U$, $\Sigma$ и $V$. Матрицы $U$ и $V$ ортонормальны по столбцам, т. е. ее столбцы, рассматриваемые как векторы, попарно ортогональны, а их длины равны 1. Матрица $\Sigma$ диагональная, а элементы на ее главной диагонали называются сингулярными значениями. Произведение $U$, $\Sigma$ и матрицы, транспонированной к $V$ равно исходной матрице.
- *Концепты.* Сингулярное разложение полезно, когда существует небольшое число концептов, связывающих строки и столбцы исходной матрицы. Например, если исходная матрица представляет оценки, поставленные пользователями (строки) фильмам (столбцы), то в роли концептов могут выступать жанры фильмов. Матрица $U$ связывает строки с концептами, $\Sigma$ описывает силу концептов, а $V$ связывает концепты со столбцами.
- *Запросы с использованием сингулярного разложения.* Разложение можно использовать для соотнесения новых или гипотетических строк исходной матрицы с концептами, представленными разложением. Умножив строку на матрицу $V$, мы получим вектор, показывающий степень соответствия строки каждому концепту.

- *Применение сингулярного разложения для понижения размерности.* В полном сингулярном разложении матрицы $U$ и $V$ обычно такие же большие, как исходная. Чтобы уменьшить число столбцов $U$ и $V$, удалим из $U$, $V$ и столбцы, соответствующие наименьшим сингулярным значениям. При таком выборе минимизируется ошибка реконструирования исходной матрицы по модифицированным $U$, $V$ и $\Sigma$.
- *Разложение разреженных матриц.* Даже в типичном случае, когда исходная матрица разрежена, матрицы, образующие сингулярное разложение, плотные. Идея CUR-декомпозиции заключается в том, чтобы разложить разреженную матрицу на меньшие и тоже разреженные матрицы, произведение которых аппроксимирует исходную.
- *CUR-декомпозиция.* В этом методе мы выбираем из заданной разреженной матрицы множество столбцов $C$ и множество строк $R$, которые играют роль $U$ и $V^T$ в сингулярном разложении; разрешается выбрать любое число строк и столбцов. Выбор производится случайным образом, причем распределение зависит от нормы Фробениуса – квадратного корня из суммы квадратов элементов. Между $C$ и $R$ в произведении находится квадратная матрица $U$, которая строится как псевдообратная к матрице, полученной пересечением выбранных строк и столбцов.

# 11.6. Список литературы

Книга [4] – авторитетный учебник по матричной алгебре.

Метод главных компонент был изобретен больше ста лет назад в работе [6]. Метод сингулярного изображения впервые описан в [3]. Существует много применений этой идеи. Особого упоминания заслуживают работа [1], где она применяется к анализу документов, и работа [8] – применение сингулярного разложения в биологии.

CUR-декомпозиция описывается в [2] и [5]. Наше описание следует более поздней работе [7].

1. S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, R. Harshman «Indexing by latent semantic analysis», J. American Society for Information Science 41:6 (1990).
2. P. Drineas, R. Kannan, M. W. Mahoney «Fast Monte-Carlo algorithms for matrices III: Computing a compressed approximate matrix decomposition», SIAM J. Computing 36:1 (2006), pp. 184–206.
3. G. H. Golub, W. Kahan «Calculating the singular values and pseudoinverse of a matrix», J. SIAM Series B 2:2 (1965), pp. 205–224.
4. G. H.Golub, C.F. Van Loan «Matrix Computations», JHU Press, 1996.
5. M. W. Mahoney, M. Maggioni, P. Drineas «Tensor-CUR decompositions for tensor-based data», SIGKDD, pp. 327–336, 2006.
6. K. Pearson «On lines and planes of closest fit to systems of points in space», Philosophical Magazine 2:11 (1901), pp. 559–572.

7. J. Sun, Y. Xie, H. Zhang, C. Faloutsos «Less is more: compact matrix decomposition for large sparse graphs», Proc. SIAM Intl. Conf. on Data Mining, 2007.
8. M. E. Wall, A. Reichtsteiner, L. M. Rocha «Singular value decomposition and principal component analysis», in A Practical Approach to Microarray Data Analysis (D.P. Berrar, W. Dubitzky, M. Granzow eds.), pp. 91–109, Kluwer, Norwell, MA, 2003.

# ГЛАВА 12. Машинное обучение на больших данных

Многие алгоритмы сегодня относят к категории «машинное обучение». Наряду с остальными рассмотренными в этой книге алгоритмами их цель – извлечение информации из данных. Все алгоритмы анализа данных призваны построить полезное обобщенное представление данных, на основе которого можно принимать решения. Так, в результате анализа частых предметных наборов порождаются ассоциативные правила, которыми затем можно воспользоваться для планирования стратегии продаж и многих других целей.

Однако алгоритмы «машинного обучения» не ограничиваются обобщением данных; в ходе обучения они строят из данных модель, или классификатор, что открывает возможность узнать что-то о данных, которые будут предъявлены в будущем. Например, алгоритмы кластеризации – тема главы 7 – порождают кластеры, которые не только сообщают информацию о проанализированных данных (обучающем наборе), но и позволяют относить к определенному кластеру будущие данные. Поэтому энтузиасты машинного обучения часто называют кластеризацию «обучением *без учителя*», подразумевая под этим, что входные данные не содержат никакой информации, позволяющей алгоритму кластеризации узнать, какими должны быть кластеры. Напротив, в алгоритмах обучения *с учителем*, которые рассматриваются в этой главе, в состав данных включена информация о том, как правильно классифицировать хотя бы некоторые данные. Уже классифицированные данные называются *обучающим набором*.

В этом разделе мы не будем пытаться охватить все многообразные подходы к машинному обучению, а сосредоточимся на методах, пригодных для очень больших данных и допускающих распараллеливание. Мы рассмотрим классический подход к обучению классификатора на основе перцептрона, в котором ищется гиперплоскость, разделяющая классы. А затем обратимся к более современным алгоритмам, в том числе методу опорных векторов. Как и перцептроны, эти методы ищут гиперплоскость, наилучшим образом разделяющую классы, но так чтобы как можно меньше элементов обучающего набора находились близко к гиперплоскости. Мы закончим раздел обсуждением метода ближайших соседей, когда данные классифицируются на основе класса (или классов) ближайших соседей в некотором пространстве.

# 12.1. Модель машинного обучения

В этом разделе мы познакомимся с общими принципами алгоритмов машинного обучения и дадим некоторые определения.

## 12.1.1. Обучающие наборы

Данные, к которым применяется алгоритм машинного обучения (МО), называются обучающим набором. *Обучающий набор* состоит из множества пар (**x**, *y*), называемых *обучающими примерами*, где:

- **x** – *вектор признаков*. Признаки могут быть *категориальными* (значениями являются элементы дискретного множества, например {красный, синий, зеленый}) или *числовыми* (значениями являются целые или вещественные числа);
- *y* – *метка*, т. е. классифицированное значение **x**.

Цель процесса МО – найти функцию $y = f(\mathbf{x})$, которая наилучшим образом предсказывает значение *y*, ассоциированное с любым значением **x**. Тип *y*, вообще говоря, может быть произвольным, но есть несколько часто встречающихся важных случаев.

1. *y* – вещественное число. В этом случае задача МО называется *регрессией*.
2. *y* – булева величина, принимающая значения true или false, которые чаще записывают как +1 и –1 соответственно. В этом случае говорят о задаче *бинарной классификации*.
3. *y* принадлежит некоторому конечному множеству, элементы которого можно рассматривать как «классы», причем каждый элемент представляет ровно один класс. Это так называемая задача *многоклассовой классификации*.
4. *y* принадлежит некоторому потенциально бесконечному множеству, например дереву разбора *x*, интерпретируемого как предложение.

## 12.1.2. Пояснительные примеры

**Пример 12.1.** На рис. 12.1 (повтор рис. 7.1) на график нанесены рост и вес собак трех пород: бигли, чихуахуа и таксы. Эти данные можно рассматривать как обучающий набор при условии, что вместе с парой рост-вес указана порода собаки. Каждая пара (**x**, *y*) в обучающем наборе состоит из вектора признаков **x** вида [рост, вес] и ассоциированной с ним метки *y*, описывающей породу собаки, например: ([5 дюймов, 2 фунта], чихуахуа).

При поиске решающей функции *f* удобно представить себе две прямых, показанных на рис. 12.1 штриховыми линиями. Горизонтальная прямая соответствует росту 7 дюймов и отделяет биглей от чихуахуа и такс. Вертикальная прямая соответствует весу 3 фунта и отделяет чихуахуа от биглей и такс. Алгоритм реализации *f* выглядит следующим образом:

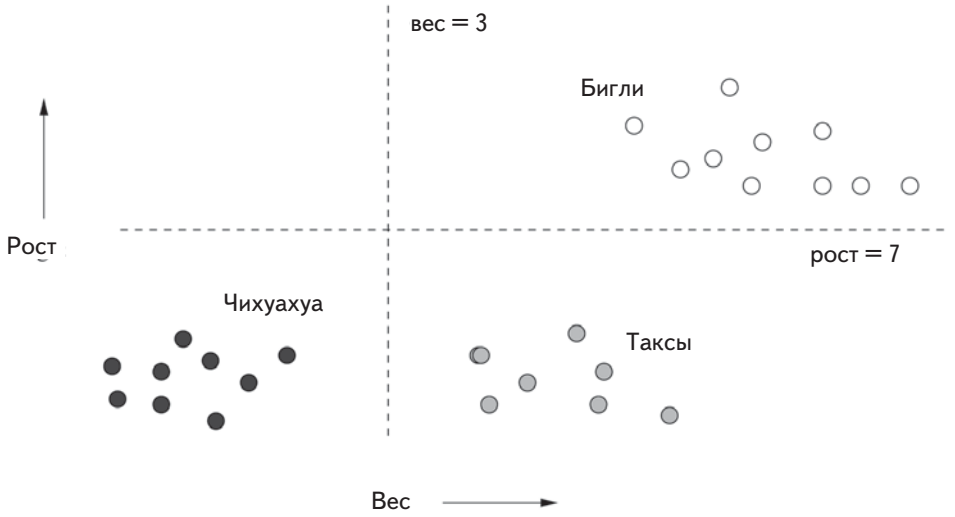



**Рис. 12.1.** Повторение рис. 7.1 – рост и вес собак трех пород

```
if (height > 7) print Beagle
else if (weight < 3) print Chihuahua
else print Dachshund;
```

Напомним, что первоначально, в главе 7, мы ставили целью кластеризовать точки, ничего не зная о породах представляемых ими собак. То есть с векторами роста и веса не были ассоциированы метки. Теперь же мы решаем задачу обучения с учителем для того же обучающего набора, дополненного информацией о классах.

**Пример 12.2.** Точки (0, 2), (2, 1), (3, 4) и (4, 3) на рис. 11.1 (повторенном на рис. 12.2) можно рассматривать как обучающий набор одномерных векторов, т. е. считать, что точка (1, 2) – это пара ([1], 2), где [1] – одномерный вектор признаков **x**, а 2 – ассоциированная с ним метка *y* (и аналогично для других точек).

Допустим, мы хотим «обучить» линейную функцию $f(x) = ax + b$, которая наилучшим образом описывает точки из обучающего набора. Естественная интерпретация слова «наилучший» – минимальная среднеквадратичная ошибка $f(x)$ по сравнению с заданным значением *y*. Иначе говоря, мы хотим минимизировать функцию

$$\sum_{x=1}^{4}(ax+b-y_x)^2 ,$$

где $y_x$ – значение *y*, ассоциированное с *x*. Эта сумма равна

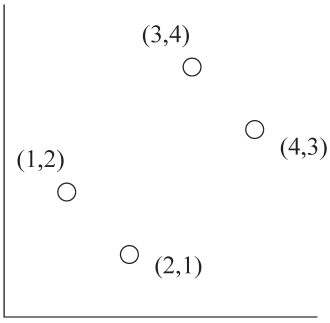



**Рис. 12.2.** Повторение рис. 11.1, интерпретируемого как обучающий набор

$$(a + b - 2)^2 + (2a + b - 1)^2 + (3a + b - 4)^2 + (4a + b - 3)^2.$$

После упрощения получаем $30a^2 + 4b^2 + 20ab - 56a - 20b + 30$. Приравняв частные производные по $a$ и по $b$ нулю, получим

$$60a + 20b - 56 = 0$$
$$20a + 8b - 20 = 0$$

Из этой системы уравнений находим $a = 3/5$, $b = 1$. При таких значениях СКО равна 3.2.

Отметим, что найденная прямая не совпадает с главной осью, найденной для этих точек в разделе 11.2.1. Там мы нашли прямую с угловым коэффициентом 1, проходящую через начало координат, т. е. имеющую уравнение $y = x$. Для нее СКО равна 4. Разница объясняется тем, что обсуждавшийся в разделе 11.2.1 метод главных компонент минимизирует сумму квадратов длин проекций на выбранную ось, которая должна проходить через начало координат. Здесь же минимизируется сумма квадратов расстояний по вертикали между точками и прямой. И даже если бы мы пытались найти прямую, проходящую через начало координат, с наименьшей СКО, то все равно нашли бы не прямую $y = x$. Можете проверить, что для прямой $y = 14/15x$ СКО меньше 4.

**Пример 12.3.** Типичное применение машинного обучения – обучающий набор, элементами которого являются векторы признаков **x** очень высокой размерности с булевыми значениями. Нас будут интересовать данные, включающие документы: сообщения электронной почты, веб-страницы, тексты газетных новостей и т. п. Элементами данных являются слова из некоторого большого словаря. Часто из словаря исключаются часто встречающиеся стоп-слова, поскольку они не несут информации о содержании документа. Кроме того, мы можем ограничить словарь только словами с высокой оценкой TF.IDF (см. раздел 1.3.1), так чтобы остались только слова, отражающие основную тему документа.

Обучающий набор состоит из пар, в которых вектор **x** представляет наличие или отсутствие каждого слова из словаря в документе. Значениями метки $y$ могут быть, скажем, +1 или –1, где +1 означает, что документ (например, почтовое сообщение) является спамом. Наша цель – обучить классификатор распознавать, являются ли будущие сообщения спамом. Это применение машинного обучения мы проиллюстрируем в примере 12.4. Или можно было бы в качестве значения $y$ выбрать темы из некоторого конечного множества, например: «спорт», «политика» и т. д., и считать, что **x** по-прежнему представляет документы, скажем, веб-страницы. Тогда цель – построить классификатор веб-страниц по тематике.

## 12.1.3. Подходы к машинному обучению

Различных алгоритмов МО очень много, и мы не собираемся рассматривать все. Представим лишь основные классы алгоритмов, характеризуемые формой представления функции $f$.

1. *Решающие деревья* кратко обсуждались в разделе 9.2.7. Здесь $f$ представлена в виде дерева, в каждом узле которого хранится функция от $\mathbf{x}$, определяющая, к какому дочернему узлу (или узлам) перейти на следующем шаге. Мы сталкивались только с двоичными деревьями, но в общем случае каждый узел может иметь произвольное число дочерних. Решающие деревья удобны для бинарной и многоклассовой классификации, особенно когда размерность вектора признаков не слишком велика (при большом числе признаков возможно переобучение).

2. *Перцептроны* – это пороговые функции, применяемые к элементам вектора $\mathbf{x} = [x_1, x_2, \ldots, x_n]$. С $i$-ым элементом ассоциирован вес $w_i$, и задан некоторый порог $\theta$. Функция возвращает +1, если

$$\sum_{i=1}^{n} w_i x_i \geq \theta$$

   и –1 в противном случае. Перцептрон подходит для бинарной классификации даже при очень большом числе признаков, например представляющих наличие или отсутствие слов в документе. Перцептроны рассматриваются в разделе 12.2.

3. *Нейронные сети* представляют собой ациклические сети перцептронов, в которых выход одного перцептрона подается на вход другого. Они пригодны для бинарной и многоклассовой классификации, т. к. результат может формироваться несколькими перцептронами, обозначающими разные классы.

4. *Непосредственное обучение на всех примерах* (instance-based learning) подразумевает использование всего обучающего набора для формирования функции $f$. Для вычисления метки $y$, ассоциированной с новым вектором признаков $\mathbf{x}$, может потребоваться исследование всего обучающего набора, хотя обычно он подвергается некоторой предварительной обработке, позволяющей эффективно вычислить $f(\mathbf{x})$. Мы рассмотрим важный частный случай непосредственного обучения – метод $k$ ближайших соседей – в разделе 12.4. Например, в методе одного ближайшего соседа данным назначается тот же класс, что у ближайшего обучающего примера. Существуют варианты метода $k$ ближайших соседей, применимые к любому виду классификации, но мы будем рассматривать только случай, когда $y$ и элементы $\mathbf{x}$ – вещественные числа.

5. *Метод опорных векторов* – прогресс по сравнению с традиционными алгоритмами выбора весов и порога. Его результатом является классификатор, более точный на ранее не предъявлявшихся примерах. Мы обсудим этот метод в разделе 12.3.

## 12.1.4. Архитектура машинного обучения

Алгоритмы машинного обучения можно классифицировать не только по принципиальному алгоритмическому подходу, как в разделе 12.1.3, но и по базовой архитектуре – способу обработки данных и их использования для построения модели.

### Обучение и тестирование

При любом способе обработки данных имеет смысл зарезервировать часть имеющихся данных и не включать ее в обучающий набор. Эти «отложенные в сторонку» данные называют *тестовым*, или *контрольным* набором. Проблема в том, что многие алгоритмы машинного обучения склонны к *переобучению*: они улавливают явления, встречающиеся в обучающем наборе, но не характерные для более крупных совокупностей данных. Анализируя поведение классификатора на тестовом наборе, мы можем сказать, имело место переобучение или нет. Если имело, то необходимо наложить какие-то ограничения на алгоритм. Например, при построении решающего дерева мы могли бы ограничить количество уровней дерева.

На рис. 12.3 показана архитектура обучения и тестирования. Предполагается, что все данные пригодны для обучения (т. е. в состав данных входит информация о классе), но небольшая часть имеющихся данных резервируется для тестирования. Остальные данные используются для построения модели или классификатора. Затем на вход модели подаются тестовые данные. Поскольку класс каждого тестового примера известен, мы можем сказать, хорошо ли ведет себя модель на тестовых данных. Если частота ошибок на тестовых данных не намного хуже, чем на обучающих, то можно предположить, что переобучение, если и есть, то небольшое, и модель можно использовать. С другой стороны, если на тестовых данных классификатор работает гораздо хуже, чем на обучающих, то имеет место переобучение, и нужно переосмыслить способ построения классификатора.

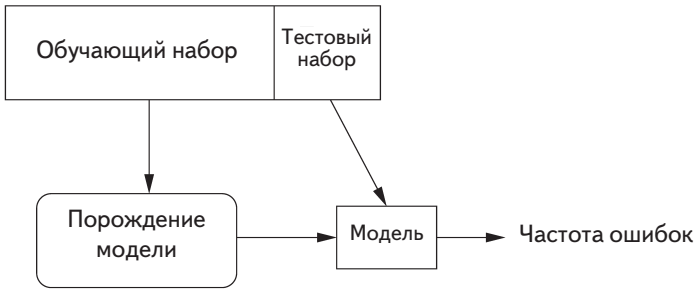



**Рис. 12.3.** Обучающий набор используется для построения модели, а тестовый – для ее проверки

В выборе тестовых данных нет ничего особенного. На самом деле, можно повторить процесс обучения и тестирования несколько раз на тех же самых данных, если разделить данные на $k$ частей одинакового размера. Мы поочередно дела-

ем каждую часть тестовыми данными, а остальные $k - 1$ частей используем как обучающие. Такая архитектура обучения называется *перекрестной проверкой*.

**Обобщение**

Следует помнить, что цель создания модели или классификатора заключается не в том, чтобы классифицировать обучающий набор, а в том, чтобы классифицировать данные, класс которых заранее не известен. Мы хотим классифицировать данные правильно, но зачастую не знаем, справляется с этим модель или нет. Если характер данных со временем изменяется, как бывает в случае распознавания спама, то необходимо время от времени измерять качество классификации всеми доступными способами. Так, в случае почтового спама можно фиксировать частоту извещений о спамных сообщениях, которые не были классифицированы как спам.

## Пакетное и оперативное обучение

Часто используется архитектура *пакетного обучения*, как в примерах 12.1 и 12.2. В этом случае весь обучающий набор доступен в начале процесса и используется так, как необходимо алгоритму, чтобы породить модель раз и навсегда. Альтернативой является *оперативное обучение*, когда обучающие данные поступают потоком и, как любой поток, теряются после обработки. В случае оперативного обучения мы постоянно перестраиваем модель. По мере поступления новых обучающих примеров модель может модифицироваться для их учета. У оперативного обучения есть ряд преимуществ.

1. Оно справляется с очень большими обучающими наборами, поскольку никогда не обрабатывает более одного примера за раз.
2. Оно может адаптироваться к изменениям в совокупности обучающих примеров со временем. Например, именно так Google обучает классификатор спама, адаптируя его к новым видам рассылок, которые получатели объявили спамом.

В некоторых случаях применяют усовершенствованный вариант оперативного обучения – *активное обучение*, когда классификатор иногда может получать обучающие примеры, но чаще получает неклассифицированные данные, которые должен классифицировать. Если классификатор не уверен (например, новый пример расположен очень близко к решающей границе), то он может запросить объективное мнение ценой весьма значительных затрат. Скажем, можно отправить пример Механическому турку и поинтересоваться мнением реальных людей. В таком случае примеры, близкие к границе, становятся обучающими и могут использоваться для модификации классификатора.

## Отбор признаков

Иногда самой трудной частью проектирования хорошей модели или классификатора является решение о том, какие признаки использовать в качестве входных данных для обучающего алгоритма. Вернемся к примеру 12.3, где мы предположили, что классифицировать сообщение как спам можно, глядя на встречающиеся в нем слова. Мы еще изучим такой классификатор более детально в примере 12.4. В примере 12.3 было отмечено, что имеет смысл уделять больше внимания одним словам и меньше другим, например, следует исключить стоп-слова.

Но следует задаться вопросом, а нет ли еще какой-то информации, которая помогла бы принять более правильное решение относительно спама. Например, спам часто генерируется определенными серверами, либо принадлежащими спамерам, либо включенным в состав «ботнета», созданного специально для рассылки спама. Поэтому включение сервера-отправителя или почтового адреса отправителя в состав вектора признаков сообщения позволило бы создать более качественный классификатор и уменьшить частоту ошибок.

## Создание обучающего набора

Возникает естественный вопрос: откуда берутся метки, превращающие данные в обучающий набор? Самый очевидный способ – создание вручную, когда специалист оценивает вектор признаков и правильно классифицирует его. В последнее время для пометки данных стали применяться методы краудсорсинга. Например, во многие приложения встроена возможность использовать Механического турка для пометки данных. Поскольку «турки» не всегда надежны, разумно применять систему, которая позволяет задавать вопрос нескольким разным лицам, пока не будет сформировано явное большинство в пользу какой-то метки.

Часто данные можно найти в вебе, который неявно уже размечен. Например, в открытом каталоге (DMOZ) представлены миллионы тематически помеченных страниц. Если использовать эти данные в качестве обучающего набора, то, возможно, удастся классифицировать другие страницы или документы по теме, исходя из частоты встречаемости слов. Другой подход к тематической классификации – поискать тему в википедии и посмотреть, на какие страницы она ссылается. Можно с уверенностью утверждать, что эти страницы релевантны данной теме.

В некоторых приложениях мы можем воспользоваться оценками, которые пользователи ставят товарам или услугам, как на сайтах Amazon или Yelp. Предположим, к примеру, что мы хотим прикинуть, сколько звездочек было бы поставлено в некотором отзыве или твите, касающемся товара, хотя в действительности этот отзыв не содержит оценки. Если использовать помеченные звездочками отзывы как обучающий набор, то можно определить, какие слова чаще ассоциируются с положительными, а какие – с отрицательными отзывами (это называется *анализом эмоциональной окраски*). Наличие таких слов в других отзывах может рассказать об их эмоциональной окраске.

### 12.1.5. Упражнения к разделу 12.1

**Упражнение 12.1.1.** Повторите пример 12.2 для разных $f(x)$.

(*а*) $f(x) = ax$, т. е. прямая, проходящая через начало координат. Является ли упоминавшаяся прямая $y = 14/15x$ оптимальной?

(*б*) $f(x)$ – квадратичная функция вида $f(x) = ax^2 + bx + c$.

## 12.2. Перцептроны

*Перцептрон* – это линейный бинарный классификатор. Входными данными для него является вектор $\mathbf{x} = [x_1, x_2, ..., x_d]$ с вещественными элементами. С перцептроном ассоциирован вектор вещественных *весов* $\mathbf{w} = [w_1, w_2, ..., w_d]$. Для перцептрона определен также *порог* $\theta$. На выходе перцептрон порождает +1, если $\mathbf{w}\cdot\mathbf{x} > \theta$, и –1, если $\mathbf{w}\cdot\mathbf{x} < \theta$. Специальный случай $\mathbf{w}\cdot\mathbf{x} = \theta$ всегда будет считаться «неправильным», а что это значит, мы подробно опишем в разделе 12.2.1.

Вектор весов $\mathbf{w}$ определяет гиперплоскость размерности $d - 1$ – множество таких точек $\mathbf{x}$, что $\mathbf{w}\cdot\mathbf{x} = \theta$ (рис. 12.4). Точки, расположенные с положительной стороны гиперплоскости, классифицируются как +1, а с отрицательной – как –1. Перцептронный классификатор работает только для *линейно разделимых* данных в том смысле, что существует гиперплоскость, отделяющая положительные точки от отрицательных. Если таких гиперплоскостей много, то перцептрон сходится к одной из них и потому правильно классифицирует все обучающие примеры. Если не существует ни одной такой гиперплоскости, то перцептрон не сходится вообще. В следующем разделе мы обсудим метод опорных векторов, не имеющий такого ограничения: он всегда сходится к какому-то разделителю, который может и не быть идеальным классификатором, но делает все возможное относительно метрики, которую мы опишем в разделе 12.3.

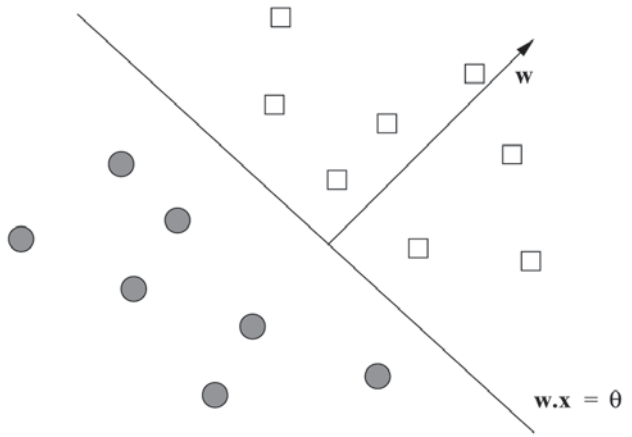



**Рис. 12.4.** Перцептрон разделяет пространство гиперплоскостью на два полупространства

## 12.2.1. Обучение перцептрона с нулевым порогом

Для обучения перцептрона мы пытаемся подобрать такой вектор весов **w** и порог θ, чтобы все векторы признаков, для которых $y = +1$ (*положительные примеры*), находились с положительной стороны гиперплоскости, а те, для которых $y = -1$ (*отрицательные примеры*), – с отрицательной стороны. Иногда это возможно, а иногда нет, поскольку никто не гарантирует, что гиперплоскость, разделяющая положительные и отрицательные примеры, вообще существует.

Для начала предположим, что порог равен 0; для обобщения наших рассуждений на неизвестный порог потребуется лишь небольшое дополнение, описанное в разделе 12.2.4. Изложенный ниже алгоритм сходится к некоторой гиперплоскости, разделяющей положительные и отрицательные примеры, если такая гиперплоскость существует.

1. Инициализировать вектор весов нулями.
2. Выбрать *параметр скорости обучения* η – небольшое положительное вещественное число. От выбора η зависит сходимость перцептрона. Если η слишком мал, то алгоритм сходится медленно, а если слишком велик, то решающая граница будет «плясать», и алгоритм опять-таки будет сходиться медленно и, возможно, вообще не сойдется.
3. Поочередно рассмотреть все обучающие примеры $t = (\mathbf{x}, y)$.
   (*а*) Положить $y' = \mathbf{w} \cdot \mathbf{x}$.
   (*б*) Если $y'$ и $y$ одного знака, ничего не делать: $t$ классифицирован правильно.
   (*в*) Если же $y'$ и $y$ разного знака или $y' = 0$, заменить **w** на $\mathbf{w} + \eta y \mathbf{x}$, т. е. немного подкорректировать **w** в направлении **x**.

В двумерном случае это преобразование **w** показано на рис. 12.5. Обратите внимание, что сдвиг **w** в направлении **x** поворачивает гиперплоскость, перпендикулярную **w** таким образом, что вероятность нахождения **x** по правильную сторону увеличивается, хотя гарантии, что так и будет, все же нет.

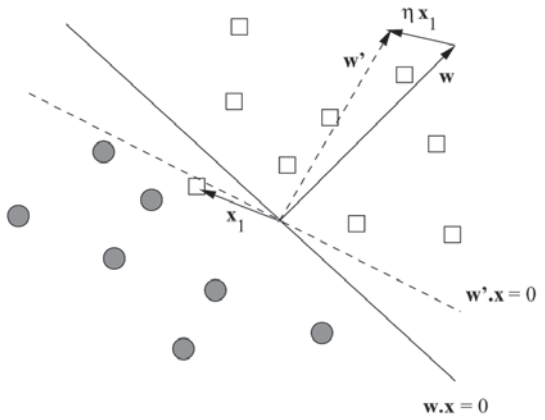



**Рис. 12.5.** Неправильно классифицированная точка $\mathbf{x}_1$ сдвигает вектор **w**

**Пример 12.4.** Рассмотрим обучение перцептрона для распознавания почтового спама. Обучающий набор состоит из пар (**x**, *y*), где **x** – вектор из нулей и единиц, в котором элемент $x_i = 1$, если определенное слово встречается в сообщении, и $x_i = 0$, если не встречается. Метка *y* равна +1, если сообщение спамное, и –1 в противном случае. На практике число слов в обучающем наборе для распознавания спама очень велико, но мы для упрощения будем использовать только пять слов: «and» (и), «viagra» (виагра), «the», «of» (из) и «nigeria» (нигерия). На рис. 12.6 приведен обучающий набор из шести векторов и соответствующих им классов.

| | and | viagra | the | of | nigeria | *y* |
|---|---|---|---|---|---|---|
| **a** | 1 | 1 | 0 | 1 | 1 | +1 |
| **b** | 0 | 0 | 1 | 1 | 0 | –1 |
| **c** | 0 | 1 | 1 | 0 | 0 | +1 |
| **d** | 1 | 0 | 0 | 1 | 0 | –1 |
| **e** | 1 | 0 | 1 | 0 | 1 | +1 |
| **f** | 1 | 0 | 1 | 1 | 0 | –1 |

**Рис. 12.6.** Обучающие данные для распознавания почтового спама

В этом примере мы возьмем скорость обучения $\eta = 1/2$ и будем рассматривать каждый обучающий пример по одному разу в том порядке, в каком они перечислены на рис. 12.6. Начнем с вектора $\mathbf{w} = [0, 0, 0, 0, 0]$, для которого $\mathbf{w}\cdot\mathbf{a} = 0$. Поскольку число 0 не положительное, сдвигаем **w** в направлении **a**, выполнив присваивание $\mathbf{w} := \mathbf{w} + (1/2)(+1)\mathbf{a}$. Новое значение **w** равно

$$\mathbf{w} = [0, 0, 0, 0, 0] + [\tfrac{1}{2}, \tfrac{1}{2}, 0, \tfrac{1}{2}, \tfrac{1}{2}] = [\tfrac{1}{2}, \tfrac{1}{2}, 0, \tfrac{1}{2}, \tfrac{1}{2}]$$

Далее рассматриваем вектор **b**. $\mathbf{w}\cdot\mathbf{b} = [\tfrac{1}{2}, \tfrac{1}{2}, 0, \tfrac{1}{2}, \tfrac{1}{2}]\cdot[0, 0, 1, 1, 0] = \tfrac{1}{2}$. Поскольку ассоциированная с вектором **b** метка *y* равна –1, **b** классифицирован неправильно. Поэтому выполняем присваивание

$$\mathbf{w} := \mathbf{w} + (1/2)(-1)\mathbf{b} = [\tfrac{1}{2}, \tfrac{1}{2}, 0, \tfrac{1}{2}, \tfrac{1}{2}] - [0, 0, \tfrac{1}{2}, \tfrac{1}{2}, 0] = [\tfrac{1}{2}, \tfrac{1}{2}, -\tfrac{1}{2}, 0, \tfrac{1}{2}]$$

Следующим обрабатывается обучающий пример **c**. Вычисляем

$$\mathbf{w}\cdot\mathbf{c} = [\tfrac{1}{2}, \tfrac{1}{2}, -\tfrac{1}{2}, 0, \tfrac{1}{2}].[0, 0, 1, 1, 0] = 0$$

Поскольку с примером **c** ассоциирована метка *y* = +1, то **c** также классифицирован неправильно. Поэтому выполняем присваивание

$$\mathbf{w} := \mathbf{w} + (1/2)(+1)\mathbf{c} = [\tfrac{1}{2}, \tfrac{1}{2}, -\tfrac{1}{2}, 0, \tfrac{1}{2}] + [0, \tfrac{1}{2}, \tfrac{1}{2}, 0, 0] = [\tfrac{1}{2}, 1, 0, 0, \tfrac{1}{2}]$$

Далее рассматриваем обучающий пример **d**:

$$\mathbf{w}\cdot\mathbf{d} = [\tfrac{1}{2}, 1, 0, 0, \tfrac{1}{2}].[1, 0, 0, 1, 0] = 1$$

Так как с **d** ассоциирована метка *y* = –1, то и **d** классифицирован неправильно. Поэтому присваиваем

$$\mathbf{w} := \mathbf{w} + (1/2)(-1)\mathbf{d} = [\tfrac{1}{2}, 1, 0, 0, \tfrac{1}{2}] - [\tfrac{1}{2}, 0, 0, \tfrac{1}{2}, 0] = [0, 1, 0, -\tfrac{1}{2}, \tfrac{1}{2}]$$

Для обучающего примера **e** вычисляем $\mathbf{w}\cdot\mathbf{e} = [0, 1, 0, -\frac{1}{2}, \frac{1}{2}].[1, 0, 1, 0, 1] = \frac{1}{2}$. Поскольку с **e** ассоциирована метка $y = +1$, то **e** классифицирован правильно и **w** изменять не нужно. Аналогичные вычисления для **f**

$$\mathbf{w}\cdot\mathbf{f} = [0, 1, 0, -\frac{1}{2}, \frac{1}{2}].[1, 0, 1, 1, 0] = -\frac{1}{2}$$

показывают, что **f** классифицирован правильно. Повторно проверив примеры от **a** до **d**, мы обнаружим, что этот вектор **w** правильно классифицирует и их тоже. Следовательно, алгоритм сошелся к перцептрону, который правильно классифицирует все обучающие примеры. К тому же, он дает вполне осмысленные результаты: «viagra» и «nigeria» указывают на спам, а «of» не является признаком спама. Слова «and» и «the» он считает нейтральными, хотя, наверное, мы предпочли бы, чтобы у слов «and», «of» и «the» был одинаковый вес.

**Прагматические аспекты обучения на почтовых сообщениях**

Представляя почтовые сообщения или другие объемные документы в виде обучающих примеров, мы не хотим действительно строить векторы из нулей и единиц, содержащие по одному элементу для каждого слова, даже если оно встречается в наборе сообщений всего один раз. Это привело бы к появлению крайне разреженных векторов с миллионами элементов. Вместо этого мы создаем таблицу, в которой всем словам, встречающимся в сообщениях, сопоставлены целые числа 1, 2, ..., обозначающие номер соответствующего им элемента. Обрабатывая сообщение из обучающего набора, мы строим список номеров элементов, равных 1, т. е. применяем стандартное представление разреженного вектора. Если исключить из этого представления стоп-слова, а, возможно, еще и слова с низкой оценкой TF.IDF, то векторы, представляющие сообщения, будут еще более разреженными, а, значит, степень сжатия данных еще выше. Перечислять все элементы необходимо только для вектора **w**, потому что после обработки небольшого числа обучающих примеров он перестанет быть разреженным.

## 12.2.2. Сходимость перцептронов

В начале этого раздела мы сказали, что если данные линейно разделимы, то алгоритм перцептрона сходится к разделителю. Но если данные линейно неразделимы, то рано или поздно вектор весов повторится и алгоритм зациклится. К сожалению, во время работы алгоритма трудно сказать, какой случай имеет место. Если данные велики, то практически нереально запоминать все предыдущие векторы весов и проверять, повторился вектор или нет. Но даже если бы это было возможно, период мог бы оказаться настолько большим, что алгоритм все равно нужно было бы закончить задолго до повторения.

Есть и еще один момент: даже если обучающие данные линейно разделимы, весь набор данных может оказаться линейно неразделимым. А тогда нет смысла гонять алгоритм очень долго в надежде, что он сойдется к разделителю. Поэтому нам нужна стратегия, позволяющая принимать решение об остановке алгоритма обучения перцептрона в предположении, что он не сходится. Приведем несколько стандартных критериев остановки.

1. Останавливаться после фиксированного числа итераций.
2. Останавливаться, когда количество неправильно классифицированных примеров перестает изменяться.
3. Зарезервировать часть обучающего набора в качестве тестового и после каждой итерации прогонять перцептрон на тестовых данных. Останавливать алгоритм, когда количество ошибок на тестовых данных перестает изменяться.

Еще один прием, способствующий сходимости, – уменьшать скорость обучения с увеличением числа итераций. Например, в начале можно было положить скорость обучения равной $\eta_0$, а после $t$-ой итерации уменьшать до $\eta_0/(1 + ct)$, где $c$ – небольшая константа.

## 12.2.3. Алгоритм Winnow

Существует много других правил корректировки весов перцептрона. Не все возможные алгоритмы гарантированно сходятся, даже если существует гиперплоскость, разделяющая положительные и отрицательные примеры. Но есть один алгоритм – Winnow, который сходится наверняка, и его мы сейчас и опишем. В этом алгоритме предполагается, что векторы признаков состоят из нулей и единиц, а метки могут принимать значения +1 или –1. В отличие от базового алгоритма перцептрона, который порождает положительные или отрицательные элементы вектора весов **w**, Winnow порождает только положительные веса.

В общем случае у алгоритма Winnow много параметров, но мы рассмотрим только один простой вариант. Впрочем, во всех вариантах имеется некоторый положительный порог θ. Если **w** – текущий вектор весов, а **x** – рассматриваемый в данный момент вектор признаков из обучающего набора, то мы вычисляем $\mathbf{w}\cdot\mathbf{x}$ и сравниваем результат с θ. Если скалярное произведение слишком мало, а класс **x** равен +1, то необходимо увеличить веса в тех позициях **w**, которым соответствуют единицы в векторе **x**. Эти веса умножаются на некоторое число, большее 1. Чем больше это число, тем выше скорость обучения, поэтому следует выбирать его не слишком близким к 1 (иначе скорость сходимости будет слишком медленной), но и не слишком большим (иначе вектор весов будет осциллировать). Аналогично, если $\mathbf{w}\cdot\mathbf{x} \geq \theta$, но класс **x** равен –1, то нужно уменьшить веса в тех позициях вектора **w**, которым соответствует элементы **x**, равные 1. Эти веса умножаются на число, большее 0, но меньшее 1. И снова мы выбираем это число не слишком близким к 1, но и не слишком малым, чтобы избежать медленной сходимости и осцилляций.

Мы подробно опишем применение этого алгоритма с коэффициентами 2 и 1/2. Инициализируем вектор весов $\mathbf{w} = [w_1, w_2, ..., w_d]$ всеми единицами, и пусть порог θ

равен $d$ – размерности векторов в обучающих примерах. Обозначим очередной обучающий пример $(\mathbf{x}, y)$, где $\mathbf{x} = [x_1, x_2, \ldots, x_d]$.

1. Если $\mathbf{w}\cdot\mathbf{x} > \theta$ и $y = +1$ или $\mathbf{w}\cdot\mathbf{x} < \theta$ и $y = -1$, то пример классифицирован правильно, поэтому модифицировать $\mathbf{w}$ не нужно.
2. Если $\mathbf{w}\cdot\mathbf{x} \leq \theta$, но $y = +1$, то веса в позициях, соответствующих единицам в векторе $\mathbf{x}$, слишком низкие. Удвоим их, т. е. если $x_i = 1$, то положим $w_i := 2w_i$.
3. Если $\mathbf{w}\cdot\mathbf{x} \geq \theta$, но $y = -1$, то веса в позициях, соответствующих единицам в векторе $\mathbf{x}$, слишком высокие. Уменьшим их вдвое, т. е. если $x_i = 1$, то положим $w_i := w_i/2$.

**Пример 12.5.** Вернемся к обучающим данным на рис. 12.6. Инициализируем $\mathbf{w} = [1, 1, 1, 1, 1]$ и положим $\theta = 5$. Первым рассматриваем вектор признаков $\mathbf{a} = [1, 1, 0, 1, 1]$. Произведение $\mathbf{w}\cdot\mathbf{a} = 4$, т. е. меньше $\theta$. Поскольку ассоциированная метка равна +1, этот пример классифицирован неправильно. Если пример с меткой +1 классифицирован неправильно, то мы должны удвоить веса, которые соответствуют единицам в векторе примера; в данном случае все элементы **a**, кроме третьего, равны 1. Поэтому новое значение $\mathbf{w}$ равно $[2, 2, 1, 2, 2]$.

Следующим рассматриваем обучающий пример $\mathbf{b} = [0, 0, 1, 1, 0]$. Произведение $\mathbf{w}\cdot\mathbf{b} = 3$, т. е. меньше $\theta$. Но ассоциированная с **b** метка равна –1, поэтому модифицировать $\mathbf{w}$ не нужно. Для $\mathbf{c} = [0, 1, 1, 0, 0]$ находим $\mathbf{w}\cdot\mathbf{c} = 3 < \theta$, а ассоциированная метка равна +1. Поэтому удваиваем веса, соответствующие элементам **c**, равным 1. Это второй и третий элемент, поэтому новое значение $\mathbf{w}$ равно $[2, 4, 2, 2, 2]$.

Обучающие примеры **d** и **e** классифицированы правильно, так что никаких модификаций не требуется. Но с $\mathbf{f} = [1, 0, 1, 1, 0]$ возникает проблема, т. к. $\mathbf{w}\cdot\mathbf{f} = 6 > \theta$, а ассоциированная с **f** метка равна –1. Поэтому мы должны разделить первый, третий и четвертый элементы $\mathbf{w}$ на 2, поскольку соответствующие элементы **f** равны 1. Новое значение $\mathbf{w}$ равно $[1, 4, 1, 1, 2]$.

Алгоритм еще не сошелся. Мы должны рассмотреть все обучающие примеры от **a** до **f** еще раз. В конце этого процесса алгоритм сойдется к вектору весов $\mathbf{w} = [1, 8, 2, \frac{1}{2}, 4]$, который правильно классифицирует все примеры при выбранном пороге $\theta = 5$. Все двенадцать шагов алгоритма приведены на рис. 12.7, где показаны ассоциированная метка $y$ и скалярное произведение $\mathbf{w}$ и каждого вектора признаков. Последние пять столбцов – это пять элементов $\mathbf{w}$ после обработки очередного обучающего примера.

## 12.2.4. Переменный порог

Предположим теперь, что выбор порога 0, как в разделе 12.2.1, или порога $d$, как в разделе 12.2.3, нежелателен или что мы не знаем, какой оптимальный порог выбрать. Ценой добавления еще одного измерения в векторы признаков мы можем рассматривать $\theta$ как один из элементов вектора весов $\mathbf{w}$. Таким образом:

1. Заменяем вектор весов $\mathbf{w} = [w_1, w_2, \ldots, w_d]$ вектором $\mathbf{w}' = [w_1, w_2, \ldots, w_d, \theta]$.
2. Заменяем каждый вектор признаков $\mathbf{x} = [x_1, x_2, \ldots, x_d]$ вектором $\mathbf{x}' = [x_1, x_2, \ldots, x_d, -1]$.

| x | y | w·x | OK? | and | viagra | the | of | nigeria |
|---|---|---|---|---|---|---|---|---|
| | | | | 1 | 1 | 1 | 1 | 1 |
| **a** | +1 | 4 | нет | 2 | 2 | 1 | 2 | 2 |
| **b** | –1 | 3 | да | | | | | |
| **c** | +1 | 3 | нет | 2 | 4 | 2 | 2 | 2 |
| **d** | –1 | 4 | да | | | | | |
| **e** | +1 | 6 | да | | | | | |
| **f** | –1 | 6 | нет | 1 | 4 | 1 | 1 | 2 |
| **a** | +1 | 8 | да | | | | | |
| **b** | –1 | 2 | да | | | | | |
| **c** | +1 | 5 | нет | 1 | 8 | 2 | 1 | 2 |
| **d** | –1 | 2 | да | | | | | |
| **e** | +1 | 5 | нет | 2 | 8 | 4 | 1 | 4 |
| **f** | –1 | 7 | нет | 1 | 8 | 2 | $\frac{1}{2}$ | 4 |

**Рис. 12.7.** Последовательность модификаций **w**, произведенных алгоритмом Winnow при обработке обучающего набора на рис. 12.6

Затем для нового обучающего набора и нового вектора весов мы можем взять порог 0 и применить алгоритм из раздела 12.2.1. Это допустимо, потому что условие $\mathbf{w}'\cdot\mathbf{x}' \geq 0$ эквивалентно условию $\sum_{i=1}^{d} w_i x_i + \theta \times -1 = \mathbf{w}\cdot\mathbf{x} - \theta \geq 0$, или $\mathbf{w}\cdot\mathbf{x} \geq \theta$. Последнее неравенство является условием положительного ответа перцептрона с порогом $\theta$.

Мы также можем применить алгоритм Winnow к модифицированным данным. Для применения Winnow необходимо, что все элементы векторов признаков были равны 0 или 1. Но в элементе, соответствующем $\theta$, мы можем допустить значение –1, если будем обрабатывать его противоположно тому, как обрабатываются элементы, равные 1. Иначе говоря, если обучающий пример положительный и мы должны вдвое увеличить веса, то вес в позиции, соответствующей порогу, мы вместо этого делим на 2. А если обучающий пример отрицательный, то этот вес мы вместо деления на 2 удваиваем.

**Пример 12.6.** Модифицируем обучающий набор на рис. 12.6, включив шестое «слово» со значением $-\theta$, противоположным порогу. Новые данные показаны на рис. 12.8.

| | and | viagra | the | of | nigeria | $\theta$ | y |
|---|---|---|---|---|---|---|---|
| **a** | 1 | 1 | 0 | 1 | 1 | –1 | +1 |
| **b** | 0 | 0 | 1 | 1 | 0 | –1 | –1 |
| **c** | 0 | 1 | 1 | 0 | 0 | –1 | +1 |
| **d** | 1 | 0 | 0 | 1 | 0 | –1 | –1 |
| **e** | 1 | 0 | 1 | 0 | 1 | –1 | +1 |
| **f** | 1 | 0 | 1 | 1 | 0 | –1 | –1 |

**Рис. 12.8.** Обучающие данные для распознавания почтового спама с добавленным шестым элементом, противоположным порогу

Начинаем с вектора весов **w**, содержащего шесть единиц, он показан в первой строке на рис. 12.9. После вычисления **w**·**a** = 3 ничего делать не нужно, т. к. и обучающий пример, и скалярное произведение положительны. Но для второго обучающего примера **w**·**b** = 1. Т. к. пример отрицательный, а скалярное произведение положительно, мы должны скорректировать веса. Поскольку в векторе **b** единице равны третий и четвертый элемент, то единицы в соответствующих позициях **w** заменяются на 1/2. А последний элемент, соответствующий θ, необходимо удвоить. После этих корректировок получается новый вектор весов [1, 1, $\frac{1}{2}$, $\frac{1}{2}$, 1, 2], показанный в третьей строке на рис. 12.9.

| **x** | *y* | **w**·**x** | OK? | and | viagra | the | of | nigeria | θ |
|---|---|---|---|---|---|---|---|---|---|
| | | | | 1 | 1 | 1 | 1 | 1 | 1 |
| **a** | +1 | 3 | да | | | | | | |
| **b** | –1 | 1 | нет | 1 | 1 | $\frac{1}{2}$ | $\frac{1}{2}$ | 1 | 2 |
| **c** | +1 | $-\frac{1}{2}$ | нет | 1 | 2 | 1 | $\frac{1}{2}$ | 1 | 1 |
| **d** | –1 | $\frac{1}{2}$ | нет | $\frac{1}{2}$ | 2 | 1 | $\frac{1}{2}$ | 1 | 2 |

**Рис. 12.9.** Последовательность модификаций **w**, произведенных алгоритмом Winnow при обработке обучающего набора на рис. 12.8

Вектор признаков **c** – положительный пример, но **w**·**c** = $-\frac{1}{2}$. Поэтому мы должны удвоить второй и третий элементы **w**, поскольку в соответствующих позициях **c** находятся 1, и разделить на 2 последний элемент **w**, соответствующий θ. Получившийся в результате вектор весов **w** = [1, 2, 1, $\frac{1}{2}$, 1, 1] показан в четвертой строке на рис. 12.9. Следующий пример **d** отрицательный. Поскольку **w**·**d** = $\frac{1}{2}$, мы снова должны скорректировать веса. Делим пополам веса в первой и четвертой позиции и удваиваем последний элемент. Это дает **w** = [$\frac{1}{2}$, 2, 1, $\frac{1}{4}$, 1, 2]. Теперь для всех положительных примеров скалярное произведение с вектором весов положительно, а для всех отрицательных примеров отрицательно, поэтому модификацию весов можно прекратить.

Порог построенного перцептрона равен 2. Веса слов «viagra» и «nigeria» равны 2 и 1 соответственно, а у слов «and» и «of» веса меньше. Вес слова «the» равен 1, т. е. это слово является таким же сильным индикатором спама, как «nigeria», что сомнительно. Тем не менее, этот перцептрон правильно классифицирует все примеры.

## 12.2.5. Многоклассовые перцептроны

Обобщить основную идею перцептрона можно несколькими способами. В следующем разделе мы рассмотрим преобразования, в результате которых разделяющие границы могут быть более сложными, чем гиперплоскости. А сейчас посмотрим, как можно использовать перцептроны для классификации с несколькими классами.

Пусть дан обучающий набор с метками, принадлежащими $k$ разным классам. Сначала обучим перцептроны для каждого класса с одинаковыми порогами θ. Это

означает, что для класса $i$ обучающий пример $(\mathbf{x}, i)$ считается положительным, а все примеры вида $(\mathbf{x}, j)$, где $j \neq i$ – отрицательными. Предположим, что после обучения вектор весов перцептрона для класса $i$ равен $\mathbf{w}_i$.

Получив новый вектор $\mathbf{x}$, подлежащий классификации, мы вычисляем $\mathbf{w}_i \cdot \mathbf{x}$ для всех $i = 1, 2, ..., k$. В качестве класса $\mathbf{x}$ мы выбираем то значение $i$, для которого произведение $\mathbf{w}_i \cdot \mathbf{x}$ максимально при условии, что это произведение не меньше $\theta$. В противном случае считаем, что $\mathbf{x}$ не принадлежит ни одному из $k$ классов.

Пусть, например, требуется классифицировать веб-страницы по нескольким тематикам: спорт, политика, медицина и т. д. Мы можем представить страницу вектором, содержащим 1 для каждого слова, встречающегося на странице, и 0 для отсутствующих слов (разумеется, это лишь умозрительное представление, строить такие векторы в реальности мы не собираемся). Для каждой темы существуют определенные характерные для нее слова. Например, на спортивных страницах часто будут встречаться слова «победил», «гол», «играли» и т. д. Вектор весов для этой темы должен назначать большие веса таким характерным словам.

Новую страницу можно отнести к той теме, для которой достигает максимума скалярное произведение вектора страницы на вектор весов. Можно поступить по-другому и относить страницу ко всем темам, для которых это скалярное произведение больше некоторого порога (скорее всего, более высокого, чем порог $\theta$, использованный во время обучения).

## 12.2.6. Преобразование обучающего набора

Перцептрон может использовать для разделения двух классов только линейную функцию, но всегда существует возможность преобразовать векторы обучающего набора, перед тем как применять основанный на перцептроне алгоритм разделения классов. Поясним эту идею на примере.

**Пример 12.7.** На рис. 12.10 нанесены достопримечательности, до которых можно добраться из моего дома. Прямоугольные координаты – широта и долгота места. Некоторые достопримечательности классифицированы как «однодневные поездки» – они расположены довольно близко, так что можно обернуться за день. Другие – как «экскурсии», для их посещения требуется больше одного дня. Двум классам соответствуют кружочки и квадратики. Очевидно, не существует прямой линии, разделяющей однодневные поездки и экскурсии. Но если перейти от декартовых координат к полярным, то в преобразованном пространстве штриховая окружность, показанная на рис. 12.10, становится гиперплоскостью. Формально говоря, мы преобразуем вектор $\mathbf{x} = [x_1, x_1]$ в вектор $[\sqrt{x_1^2 + x_2^2}\,, \arctan(x_2/x_1)]$. На самом деле, мы можем заодно понизить размерность данных. Угловая координата точки не важна, нас интересует только радиус $\sqrt{x_1^2 + x_2^2}$. Поэтому двумерный вектор можно преобразовать в одномерный, единственным элементом которого является расстояние от точки до начала координат. С короткими расстояниями будет ассоциирована метка класса «однодневная поездка», а с более длинными – метка «экскурсия». Обучить такой перцептрон совсем просто.

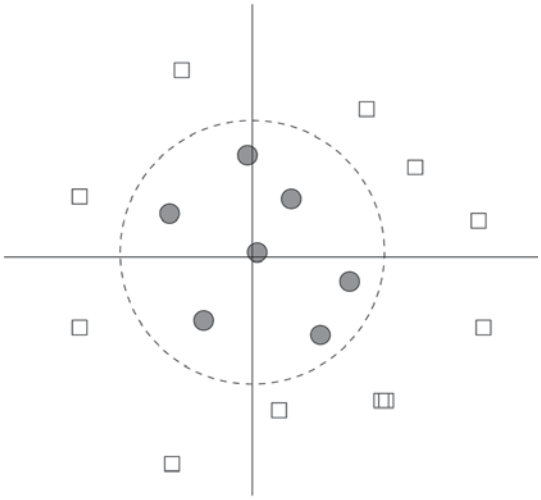


**Рис. 12.10.** В результате преобразования прямоугольных координат в полярные этот обучающий набор преобразуется в такой, для которого существует разделяющая гиперплоскость

## 12.2.7. Проблемы, связанные с перцептронами

Несмотря на описанные выше обобщения, способность перцептронов к классификации данных ограничена. Самая крупная проблема заключается в том, что иногда данные принципиально не допускают линейного разделения. Пример показан на рис. 12.11. Здесь точки двух классов перемешаны вблизи границы, поэтому какую бы прямую ни взять, по обе стороны от нее будут находиться точки из того и другого класса.

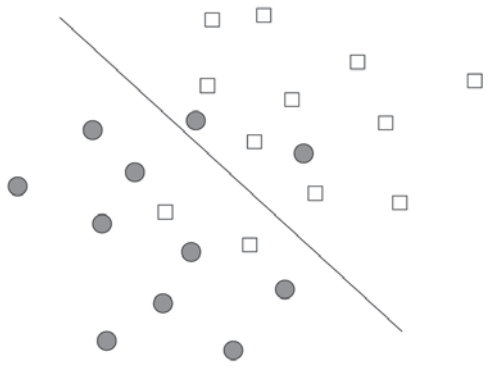


**Рис. 12.11.** Для обучающего набора может не существовать ни одной разделяющей гиперплоскости

Можно возразить, что, судя по наблюдениям из раздела 12.2.6, может найтись функция, преобразующая точки в другое пространство, где они будут линейно разделимы. Возможно, но если и так, то мы, скорее всего, столкнемся с переобучением – ситуацией, когда классификатор очень хорошо работает на обучающих данных, т. к. специально строился так, чтобы каждый пример классифицировался правильно. Но поскольку при его построении использовались особенности обучающего набора, не характерные для будущих примеров, на новых данных он будет работать плохо.

Другая проблема показана на рис. 12.12. Обычно, если классы можно разделить одной гиперплоскостью, то найдется и много других разделяющих гиперпло-

скостей. Но не все они одинаково хороши. Например, если выбрать гиперплоскость, самую дальнюю в направлении по часовой стрелке, то точка, обозначенная вопросительным знаком «?», будет классифицирована как кружочек, хотя интуитивно очевидно, что она ближе к квадратикам. При изучении метода опорных векторов в разделе 12.3 мы увидим, что существует способ выбрать гиперплоскость, так чтобы она разделяла пространство самым справедливым способом.

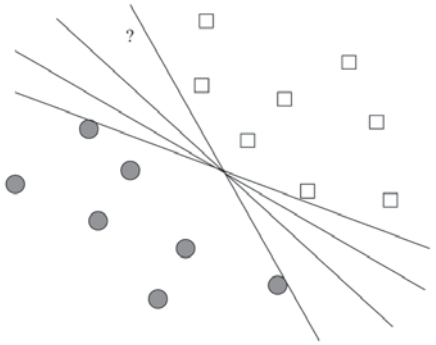



**Рис. 12.12.** В общем случае, если классы вообще можно разделить гиперплоскостью, то это можно сделать многими способами

Еще одну проблему иллюстрирует рис. 12.13. Большинство алгоритмов обучения перцептрона останавливаются, когда не остается неправильно классифицированных точек. И в результате выбирается гиперплоскость, которая просто классифицирует точки правильно. Так, верхняя прямая на рис. 12.13 вплотную подходит к двум квадратикам, а нижняя касается одного кружочка. Если одна из этих прямых соответствует окончательному вектору весов, то веса оказываются смещены в пользу одного из классов. То есть обе они правильно классифицируют обучающие примеры, но верхняя классифицирует новые квадратики, расположенные прямо под ней, как кружочки, а нижняя – кружочки, расположенные прямо над ней, как квадратики. Более справедливый выбор разделяющей гиперплоскости будет описан все в том же разделе 12.3.

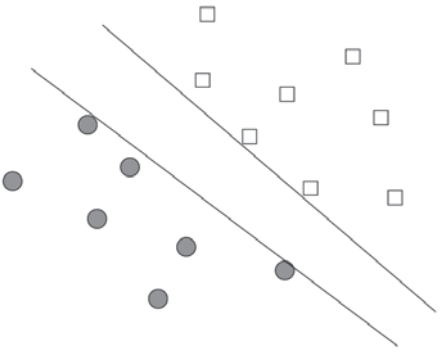


**Рис. 12.13.** Перцептрон сходится, как только разделяющая гиперплоскость достигает области между классами

## 12.2.8. Параллельная реализация перцептронов

Обучение перцептрона – принципиально последовательный процесс. Если размерность векторов очень велика, то некоторого распараллеливания можно достичь за счет параллельного вычисления скалярных произведений. Но, как было сказано в связи с примером 12.4, многомерные векторы часто бывают разреженными и могут быть представлены значительно более компактно.

Чтобы добиться значительного параллелизма, нужно немного модифицировать алгоритм обучения перцептрона, так чтобы вместе с одной и той же оценкой вектора весов **w** использовалось много обучающих примеров. Опишем параллельный алгоритм как задание MapReduce.

- **Функция Map.** Каждый распределитель получает порцию обучающих примеров и знает текущий вектор весов **w**. Распределитель вычисляет произведение $\mathbf{w}\cdot\mathbf{x}$ для каждого вектора признаков $\mathbf{x} = [x_1, x_2, ..., x_k]$ из своей порции и сравнивает результат с меткой $y$, ассоциированной с **x**, которая может быть равна +1 или −1. Если знаки совпадают, то для данного обучающего примера не порождается пара ключ-значение. Если же знаки различны, то для каждого отличного от нуля элемента $x_i$ вектора **x** порождается пара $(i, \eta y x_i)$; здесь $\eta$ – скорость обучения этого перцептрона. Отметим, что $\eta y x_i$ – приращение, которое мы хотели бы прибавить к текущему $i$-му элементу **w**, поэтому если $x_i = 0$, то порождать пару ключ-значение нет необходимости. Однако в интересах распараллеливания мы откладываем это изменение, пока не накопится много изменений для этапа редукции.
- **Функция Reduce.** Для каждого ключа $i$ обрабатывающий его редуктор складывает все ассоциированные с ключом приращения и прибавляет сумму к $i$-му элементу **w**.

Быть может, этих изменений окажется недостаточно для обучения перцептрона. Если **w** продолжает изменяться, то нам придется запустить новое задание MapReduce, которое делает то же самое, но, возможно, с другими порциями обучающего набора. Однако даже если на первом раунде был использован весь обучающий набор, он может быть использован еще раз, потому что после изменения **w** его влияние на **w** будет иным.

### Перцептроны и потоковые данные

Мы рассматривали обучающий набор как хранимые данные, к которым можно обращаться на нескольких проходах. Но перцептроны можно использовать и с потоковыми данными, т. е. можно предполагать, что имеется бесконечная последовательность обучающих примеров, каждый из которых разрешается использовать только один раз. Распознавание спама – хороший пример обучающего потока. Пользователи уведомляют как о спамных сообщениях, так и о сообщениях, которые были сочтены спамом, хотя таковым не являются. Каждое поступающее уведомление рассматривается как обучающий пример и модифицирует текущий вектор весов, быть может, очень незначительно.

Если обучающий набор является потоком, то процесс никогда по-настоящему не сойдется, а точки могут и не быть линейно разделимыми. Однако в любой момент времени у нас имеется приближение к наилучшему возможному разделителю. Более того, если примеры в потоке со временем эволюционируют, как в случае почтового спама, то наше приближение придает недавним примерам большую ценность, чем примерам из далекого прошлого, – как в методе экспоненциально затухающего окна из раздела 4.7.

### 12.2.9. Упражнения к разделу 12.2

**Упражнение 12.2.1.** Модифицируйте обучающий набор на рис. 12.6, включив слово «nigeria» еще и в пример **b** (при этом он все равно должен остаться отрицательным – быть может, кто-то просто рассказывает о своей поездке в Нигерию). Найдите вектор весов, разделяющий положительные и отрицательные примеры, с помощью:

(*а*) базового метода обучения из раздела 12.2.1;
(*б*) алгоритма Winnow из раздела 12.2.3;
(*в*) базового метода с переменным порогом из раздела 12.2.4;
(*г*) алгоритма Winnow с переменным порогом из раздела 12.2.4.

**! Упражнение 12.2.2.** Для следующего обучающего набора

$$([1, 2],+1) \quad ([2, 1],+1)$$
$$([2, 3],-1) \quad ([3, 2],-1)$$

опишите все векторы $\mathbf{w}$ и пороги $\theta$ такие, что гиперплоскость (в действительности прямая), определенная уравнением $\mathbf{w}\cdot\mathbf{x} - \theta = 0$, правильно разделяет точки.

**! Упражнение 12.2.3.** Пусть обучающий набор состоит из таких четырех примеров:

$$([1, 2],-1) \quad ([2, 3],+1)$$
$$([2, 1],+1) \quad ([3, 2],-1)$$

(*а*) Что произойдет при попытке обучить перцептрон классификации этих точек, если в качестве порога задать 0?

**!!** (*б*) Можно ли путем изменения порога получить перцептрон, который будет классифицировать эти точки правильно?

(*в*) Приведите пример полинома второй степени, который преобразует эти точки так, что они становятся линейно разделимыми.

## 12.3. Метод опорных векторов

*Метод опорных векторов* (support-vector machine, SVM) можно считать усовершенствованием перцептрона, призванным решить проблемы, упомянутые в разделе 12.2.7. Этот метод выбирает вполне определенную гиперплоскость, которая не только разделяет точки на два класса, но еще и максимизирует *зазор* – расстояние между гиперплоскостью и ближайшими к ней точками обучающего набора.

### 12.3.1. Механизм метода опорных векторов

Цель метода опорных векторов состоит в выборе гиперплоскости $\mathbf{w}\cdot\mathbf{x} + b = 0$[1], для которой достигает максимума расстояние $\gamma$ между гиперплоскостью и любой точкой обучающего набора. Иллюстрация этой идеи показана на рис. 12.14, где мы видим точки из двух классов и разделяющую их гиперплоскость.

[1] Константа $b$ в этом уравнении гиперплоскости — то же, что число, противоположное порогу $\theta$ в обсуждении перцептронов в разделе 12.2.

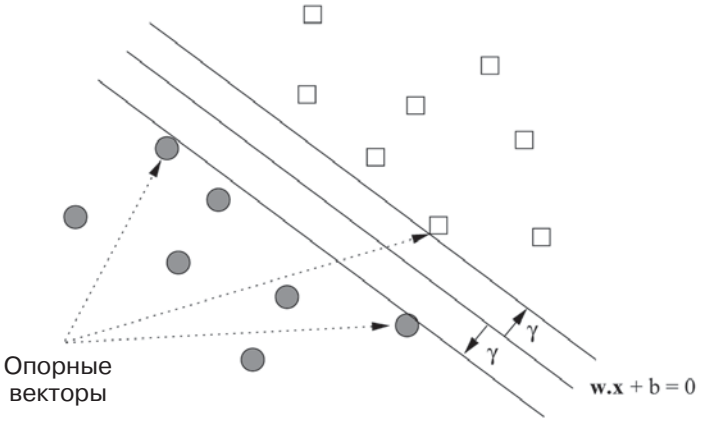



**Рис. 12.14.** Метод SVM выбирает гиперплоскость с максимально возможным зазором γ между ней и точками обучающего набора

Интуитивно мы более уверены в классе точек, находящихся дальше от разделяющей гиперплоскости. Поэтому желательно, чтобы все обучающие точки располагались на максимально возможном удалении от гиперплоскости (но, конечно, по правильную сторону). Дополнительное преимущество выбора разделяющей гиперплоскости с максимальным зазором заключается в том, что в полном наборе данных могут оказаться точки, расположенные ближе к гиперплоскости, хотя в обучающем наборе таких нет. В таком случае шансы на правильную классификацию таких точек выше, чем если бы мы выбрали разделяющую гиперплоскость, в непосредственной близости от которой находятся какие-то точки, – при этом новая точка, также близкая к гиперплоскости, могла бы быть классифицирована неправильно. Эта проблема обсуждалась в разделе 12.2.7 и проиллюстрирована на рис. 12.13.

На рис. 12.14 мы также видим две параллельные гиперплоскости на расстоянии γ от средней гиперплоскости $\mathbf{w}\cdot\mathbf{x} + b = 0$, на каждой из которых заканчивается один или несколько *опорных векторов*. Это точки, которые налагают ограничения на разделяющую гиперплоскость в том смысле, что все они находятся от нее на расстоянии γ. В большинстве случаев у $d$-мерного множества точек имеется $d + 1$ опорных векторов, как на рис. 12.14. Но опорных векторов может быть и больше, если слишком много точек лежит на параллельных гиперплоскостях. Мы еще увидим пример, основанный на рис. 11.1, где все четыре точки являются опорными векторами, хотя обычно в двумерном случае их только три.

Первая попытка сформулировать задачу звучит так.

- Пусть дан обучающий набор $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. Требуется максимизировать γ (путем изменения $\mathbf{w}$ и $b$) при наличии ограничения: $y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geq \gamma$ для любого $i = 1, 2, \ldots, n$.

Отметим, что значение $y_i$, которое может быть равно +1 или –1, определяет, по какую сторону от гиперплоскости должна находиться точка $x_i$, поэтому условие $\geq \gamma$

всегда справедливо. Но, быть может, проще записать его как два отдельных случая: если $y = +1$, то $\mathbf{w}\cdot\mathbf{x} \geq \gamma$, а если $y = -1$, то $\mathbf{w}\cdot\mathbf{x} \leq -\gamma$.

К сожалению, эта формулировка не годится. Проблема в том, что путем увеличения $\mathbf{w}$ и $b$ мы всегда сможем подобрать большее значение $\gamma$. Например, предположим, что $\mathbf{w}$ и $b$ удовлетворяет наложенному ограничению. Если заменить $\mathbf{w}$ на $2\mathbf{w}$, а $b$ на $2b$, то для всех $i$ будем иметь $y_i((2\mathbf{w})\cdot\mathbf{x}_i + 2b) \geq 2\gamma$. Таким образом, $2\mathbf{w}$ и $2b$ оказываются лучше, чем $\mathbf{w}$ и $b$, поэтому не существует ни наилучшего выбора, ни максимального $\gamma$.

## 12.3.2. Нормировка гиперплоскости

Решение проблемы, которую мы описали выше на интуитивном уровне, заключается в том, чтобы нормировать вектор весов $\mathbf{w}$. Единицей измерения в направлении, перпендикулярном разделяющей гиперплоскости, является единичный вектор $\mathbf{w}/\|\mathbf{w}\|$. Напомним, что $\|\mathbf{w}\|$ – норма Фробениуса, или квадратный корень из суммы квадратов элементов $\mathbf{w}$. Мы потребуем, чтобы вектор $\mathbf{w}$ был таким, чтобы параллельные гиперплоскости, на которых заканчиваются опорные векторы, описывались уравнениями $\mathbf{w}\cdot\mathbf{x}+b = +1$ и $\mathbf{w}\cdot\mathbf{x}+b = -1$, как показано на рис. 12.15.

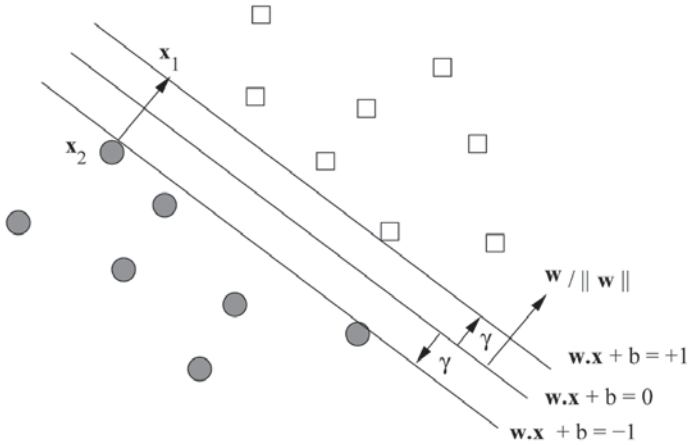



**Рис. 12.15.** Нормировка вектора весов в методе опорных векторов

Нашей целью становится максимизация зазора $\gamma$, который теперь является кратным единичного вектора $\|\mathbf{w}\|$ между разделяющей гиперплоскостью и параллельными ей гиперплоскостями, проходящими через опорные векторы. Рассмотрим один из опорных векторов, например $\mathbf{x}_2$, показанный на рис. 12.15. Обозначим $\mathbf{x}_1$ проекцию $\mathbf{x}_2$ на дальнюю гиперплоскость. Отметим, что $\mathbf{x}_1$ может не быть опорным вектором и даже не принадлежать обучающему набору. Расстояние от $\mathbf{x}_2$ до $\mathbf{x}_1$ в единицах $\mathbf{w}/\|\mathbf{w}\|$ равно $2\gamma$. То есть:

$$\mathrm{x}_1 = \mathrm{x}_2 + 2\gamma\frac{\mathrm{w}}{\|\mathrm{w}\|} \qquad (12.1)$$

Поскольку $\mathbf{x}_1$ лежит на гиперплоскости с уравнением $\mathbf{w}\cdot\mathbf{x} + b = +1$, то $\mathbf{w}\cdot\mathbf{x}_1 + b = 1$. Подставляя сюда значение $\mathbf{x}_1$ из (12.1), получаем

$$\mathrm{w}\cdot\left(\mathrm{x}_2 + 2\gamma\frac{\mathrm{w}}{\|\mathrm{w}\|}\right) + b = 1$$

После перегруппировки членов

$$\mathrm{w}\cdot\mathrm{x}_2 + b + 2\gamma\frac{\mathrm{w}\cdot\mathrm{w}}{\|\mathrm{w}\|} = 1 \qquad (12.2)$$

Но сумма первых двух членов в уравнении (12.2), $\mathbf{w}\cdot\mathbf{x}_2 + b$, равна $-1$, потому что $\mathbf{x}_2$ лежит на гиперплоскости $\mathbf{w}\cdot\mathbf{x} + b = -1$. Если перенести эту $-1$ в правую часть и поделить обе части на 2, то получим:

$$\gamma\frac{\mathrm{w}\cdot\mathrm{w}}{\|\mathrm{w}\|} = 1 \qquad (12.3)$$

Отметим также, что $\mathbf{w}\cdot\mathbf{w}$ – это сумма квадратов элементов $\mathbf{w}$, т. е. $\mathbf{w}\cdot\mathbf{w} = \|\mathbf{w}\|^2$. И следовательно, $\gamma = 1/\|\mathbf{w}\|$.

Это равенство позволяет переформулировать задачу оптимизации, поставленную в разделе 12.3.1. Вместо максимизации $\gamma$ мы будем минимизировать величину $\|\mathbf{w}\|$, обратную $\gamma$ при условии нормировки $\mathbf{w}$. Новая формулировка звучит так:

- Пусть дан обучающий набор $(\mathrm{x}_1, y_1)$, $(\mathrm{x}_2, y_2)$, ..., $(\mathrm{x}_n, y_n)$. Требуется минимизировать $\|\mathbf{w}\|$ (путем изменения $\mathbf{w}$ и $b$) при наличии ограничения: $y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geq 1$ для любого $i = 1, 2, ..., n$.

**Пример 12.8.** Рассмотрим четыре точки на рис. 11.1 и предположим, что половина из них положительные, а другая половина – отрицательные примеры, т. е. обучающий набор выглядит следующим образом:

([1, 2],+1) ([2, 1],–1)
([3, 4],+1) ([4, 3],–1)

Положим $\mathbf{w} = [u, v]$. Наша цель – минимизировать величину $\sqrt{u^2+v^2}$ при ограничениях, следующих из четырех обучающих примеров. Пример, в котором $\mathbf{x}_1 = [1, 2]$ и $y_1 = +1$, дает ограничение $(+1)(u + 2v + b) = u + 2v + b \geq 1$. Второй пример, где $\mathbf{x}_2 = [2, 1]$ и $y_2 = -1$, дает ограничение $(-1)(2u + v + b) \geq 1$, или $2u + v + b \leq -1$. Последние два примера рассматриваются аналогично, и в результате мы получаем четыре ограничения:

$u + 2v + b \geq 1$ $\quad$ $2u + v + b \leq -1$
$3u + 4v + b \geq 1$ $\quad$ $4u + 3v + b \leq -1$

Мы подробно обсудим вопрос об оптимизации при наличии ограничений; для решения этой задачи существует много программных пакетов. В разделе 12.3.4 рассматривается один из методов – градиентный спуск – в контексте более общего применения SVM, когда разделяющей гиперплоскости не существует. Иллюстрация работы этого метода приведена в примере 12.9.

А в данном простом примере решение находится просто: $b = 0$ и $\mathbf{w} = [u, v] = [-1,+1]$. Так получилось, что все четыре ограничения обращаются в равенства, т. е. каждая из четырех точек является опорным вектором. Это необычно, потому что в типичном двумерном случае бывает только три опорных вектора. Но наши положительные и отрицательные примеры лежат на параллельных прямых, поэтому все четыре ограничения удовлетворяются точно.

## 12.3.3. Нахождение оптимальных приближенных разделителей

Теперь рассмотрим нахождение оптимальной гиперплоскости в более общем случае, когда при любом выборе гиперплоскости найдутся точки, лежащие не по ту сторону, что нужно, и, быть может, точки, расположенные по правильную сторону, но слишком близко к самой гиперплоскости, так что не выполняется требование к зазору. Типичная ситуация показана на рис. 12.16. Как видно, две точки классифицированы неправильно: они находятся не по ту сторону гиперплоскости $\mathbf{w}\cdot\mathbf{x} + b = 0$. Мы также видим две правильно классифицированные точки, которые, однако, расположены слишком близко к разделяющей гиперплоскости. Назовем их *плохими* точками.

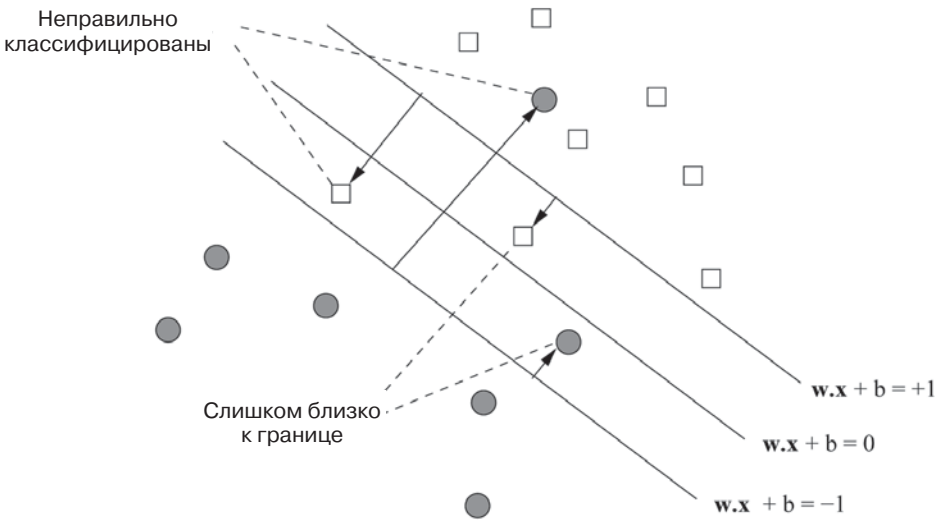



**Рис. 12.16.** Точки, которые неправильно классифицированы или расположены слишком близко к разделяющей гиперплоскости, влекут штраф; размер штрафа пропорционален длине стрелки, ведущей в точку

С каждой плохой точкой ассоциируется штраф при вычислении возможной гиперплоскости. Величина штрафа в единицах, которые еще предстоит определить в процессе оптимизации, показана стрелкой, ведущей от гиперплоскости в

плохую точку, расположенную не по ту сторону. Таким образом, стрелка измеряет расстояние до гиперплоскости $\mathbf{w}\cdot\mathbf{x} + b = 1$ или $\mathbf{w}\cdot\mathbf{x} + b = -1$. Первая представляет собой исходный уровень для обучающих примеров, которые должны быть выше разделяющей гиперплоскости (поскольку метка $y$ равна +1), вторая – исходный уровень для обучающих примеров, которые должны быть ниже (т. к. $y = -1$).

При выписывании функции, которую мы хотим минимизировать, нужно учитывать разные факторы. Интуитивно понятно, что величина $\|\mathbf{w}\|$ должна быть как можно меньше, о чем было сказано в разделе 12.3.2. Но хочется также минимизировать штрафы, ассоциированные с плохими точками. Наиболее распространенный компромисс описывается формулой, в которую входят два члена: $\|\mathbf{w}\|^2/2$ и сумма штрафов, умноженная на константу.

Чтобы понять, почему минимизация члена $\|\mathbf{w}\|^2/2$ имеет смысл, заметим, что минимизация $\|\mathbf{w}\|$ эквивалентна минимизации любой монотонно возрастающей функции от $\|\mathbf{w}\|$, поэтому включение в минимизируемое выражение величины $\|\mathbf{w}\|^2/2$ вполне допустимо. И даже желательно, потому что частная производная этой функции по любому элементу $\mathbf{w}$ равна этому элементу. Точнее, если $\mathbf{w} = [w_1, w_2, ..., w_d]$, то $\|\mathbf{w}\|^2/2$ равно $\frac{1}{2}\sum_{i=1}^{n} w_i^2$, поэтому частная производная $\partial/\partial w_i$ равна $w_i$. И это хорошо, потому что, как мы увидим, производная штрафа по $w_i$ равна константе, умноженной на $x_i$ – соответственный элемент вектора признаков, который содержит обучающий пример, повлекший за собой штраф. Это, в свою очередь, означает, что вектор $\mathbf{w}$ и векторы из обучающего набора соизмеримы в единицах измерения своих элементов.

Таким образом, мы будем рассматривать минимизацию следующей функции:

$$f(\mathrm{w}, b) = \frac{1}{2}\sum_{j=1}^{d} w_i^2 + C\sum_{i=1}^{n} \max\left\{0,\ 1 - y_i\left(\sum_{j=1}^{d} w_j x_{ij} + b\right)\right\} \qquad (12.4)$$

Первый член означает желание минимизировать $\mathbf{w}$, а второй – содержащий константу $C$, которую еще нужно аккуратно выбрать, – представляет штраф за плохие точки. Почему он записывается именно так, объяснено ниже. Мы предполагаем, что имеется $n$ обучающих примеров $(\mathbf{x}_i, y_i)$ для $i = 1, 2, ..., n$ и $\mathbf{x}_i = [x_{i1}, x_{i2}, ..., x_{id}]$. Кроме того, как и раньше, $\mathbf{w} = [w_1, w_2, ..., w_d]$. Отметим, что обе суммы $\sum_{j=1}^{d}$ – не что иное как скалярное произведение векторов.

Константа $C$, называемая *параметром регуляризации*, отражает важность неправильной классификации. Выбирая большое значение $C$, мы говорим, что не хотим неправильно классифицировать точки, но готовы смириться с узким зазором. Если же $C$ мало, значит, наличие некоторого числа неправильно классифицированных точек не страшно, но мы хотим, чтобы большинство точек отстояло далеко от границы (т. е. чтобы зазор был велик).

Мы должны еще объяснить смысл штрафной функции (второй член) в выражении (12.4). В сумме по $i$ имеется по одному члену

$$L(x_i, y_i) = \max\left\{0,\ 1 - y_i\left(\sum_{j=1}^{d} w_j x_{ij} + b\right)\right\}$$

для каждого обучающего примера $\mathbf{x}_i$. Величина $L$ описывается *кусочно-линейной функцией*, показанной на рис. 12.17, и называется *кусочно-линейной функцией потерь*. Обозначим $z_i = y_i(\sum_{j=1}^{d} w_j x_{ij} + b)$. Если $z_i$ больше или равно 1, то значение $L$ равно 0. Но для меньших значений $z_i$ величина $L$ линейно возрастает при убывании $z_i$.

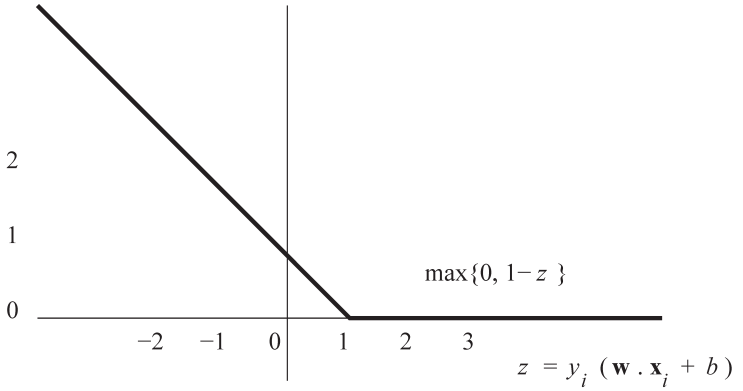



**Рис. 12.17.** Кусочно-линейная функция линейно убывает при $z \le 1$, а затем остается равной 0

Нам нужно будет брать частные производные функции $L(x_i, y_i)$ по каждому $w_j$, поэтому сразу отметим, что производная кусочно-линейной функции разрывна. Она равна $-y_i x_{ij}$ для $z_i < 1$ и 0 для $z_i > 1$. Следовательно, если $y_i = +1$ (т. е. $i$-й обучающий пример положителен), то

$$\frac{\partial L}{\partial w_j} = \text{ if } \sum_{j=1}^{d} w_j x_{ij} + b \ge 1 \text{ then } 0 \text{ else } -x_{ij}$$

С другой стороны, если $y_i = -1$ (т. е. $i$-й обучающий пример отрицателен), то

$$\frac{\partial L}{\partial w_j} = \text{ if } \sum_{j=1}^{d} w_j x_{ij} + b \le -1 \text{ then } 0 \text{ else } x_{ij}$$

Оба случая можно объединить в один, включив значение $y_i$:

$$\frac{\partial L}{\partial w_j} = \text{ if } y_i\left(\sum_{j=1}^{d} w_j x_{ij} + b\right) \ge 1 \text{ then } 0 \text{ else } -y_i x_{ij} \qquad (12.5)$$

## 12.3.4. Нахождение решений в методе опорных векторов с помощью градиентного спуска

Общий подход к оптимизации функции (12.4) – использование квадратичного программирования. Но для больших данных предпочтение отдается другому подходу – *градиентному спуску*. Дело в том, что в отличие от квадратичного программирования, мы можем хранить данные на диске, а не целиком в оперативной па-

мяти. Для реализации градиентного спуска мы вычисляем частные производные функции по $b$ и по каждому элементу $w_j$ вектора $\mathbf{w}$. Поскольку мы хотим минимизировать $f(\mathbf{w}, b)$, то сдвигаем $b$ и элементы $w_j$ в направлении, противоположном направлению градиента. Величина сдвига каждого элемента пропорциональна частной производной по этому элементу.

Первым делом воспользуемся приемом из раздела 12.2.4 и сделаем $b$ частью вектора весов $\mathbf{w}$. Отметим, что на самом деле $b$ – это порог скалярного произведения $\mathbf{w}\cdot\mathbf{x}$ с обратным знаком, поэтому мы можем добавить в конец $\mathbf{w}$ $(d + 1)$-ый элемент, равный $b$, а в конец каждого вектора признаков в обучающем наборе добавить элемент, равный +1 (а не –1, как в разделе 12.2.4).

Мы должны выбрать константу $\eta$ – долю градиента, на которую сдвигать $\mathbf{w}$ на каждой итерации. То есть мы будем выполнять присваивание

$$w_j := w_j - \eta\frac{\partial f}{\partial w_j}$$

для всех $j = 1, 2, ..., d + 1$.

Частная производная $\partial f/\partial w_j$ первого члена в выражении (12.4), $\frac{1}{2}\sum_{j=1}^{d} w_i^2$, вычисляется легко: она равна $w_j$.[2] Но второй член содержит кусочно-линейную функцию, потому записать его производную труднее. Мы воспользуемся для этой цели выражением if-then, как в формуле (12.5):

$$\frac{\partial L}{\partial w_j} = w_i + C\sum_{i=1}^{n}\left(\text{if } y_i\left(\sum_{j=1}^{d} w_j x_{ij} + b\right) \geq 1 \text{ then } 0 \text{ else } -y_i x_{ij}\right) \qquad (12.6)$$

Заметим, что эта формула применима к элементу $w_{d+1}$, равному $b$, как и ко всем весам $w_1, w_2, ..., w_d$. Мы и дальше будем использовать $b$, а не $w_{d+1}$ в условии if-then как напоминание о том, в какой форме описывается искомая гиперплоскость.

Для применения алгоритма градиентного спуска к обучающему набору мы должны выбрать:

1. Значения параметров $C$ и $\eta$.
2. Начальные значения $\mathbf{w}$, включая значение $(d + 1)$-го элемента $b$.

Затем на каждой итерации цикла мы:

(*а*) Вычисляем частные производные $f(\mathbf{w}, b)$ по $w_j$.
(*б*) Корректируем значения весов, вычитая $\eta\partial f/\partial w_j$ из каждого $w_j$.

**Пример 12.9.** На рис. 12.18 изображены шесть примеров – три положительных и три отрицательных. Мы ожидаем, что наилучшая разделяющая их прямая будет горизонтальной, и единственный вопрос – что произойдет с точкой (2, 2) при выборе разделяющей гиперплоскости и масштаба $\mathbf{w}$: неправильная классификация или чрезмерная близость к границе? Первоначально мы выбираем $\mathbf{w} = [0, 1]$ – вертикально направленный вектор дли-

[2] Отметим, однако, что $d$ теперь стало равно $d + 1$, потому что мы включили $b$ в качестве дополнительного элемента $\mathbf{w}$.

ны 1 – и полагаем $b$ =–2. В результате, как видно на рис. 12.18, точка (2, 2) лежит на гиперплоскости, а все три отрицательные точки – точно на границе зазора. Выберем такие параметры градиентного спуска: $C$ = 0.1, $\eta$ = 0.2.

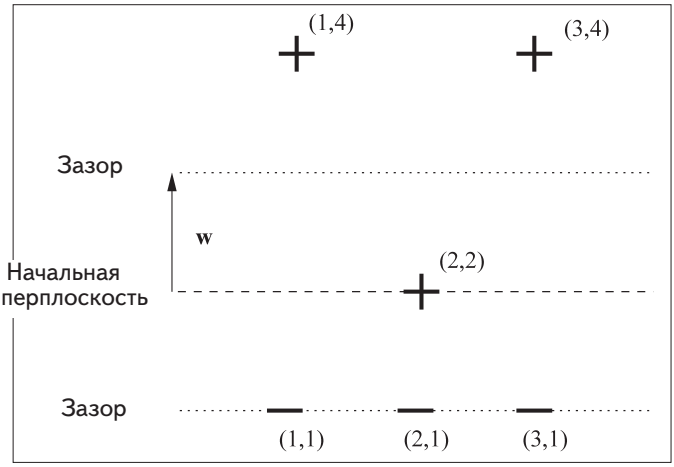



**Рис. 12.18.** Шесть точек для примера градиентного спуска

Начнем с того, что включим $b$ в вектор **w** третьим элементом, и для удобства будем обозначать первые два элемента $u$ и $v$, а не $w_1$ и $w_2$, как обычно. То есть **w** = [$u$, $v$, $b$]. Также пополним двумерные точки в обучающем наборе третьим элементом, который всегда равен 1. Таким образом, обучающий набор принимает вид

([1, 4, 1],+1) ([2, 2, 1],+1) ([3, 4, 1],+1)
([1, 1, 1],–1) ([2, 1, 1],–1) ([3, 1, 1],–1)

В таблице на рис. 12.19 приведены условия if-then и результирующие вклады в сумму по $i$ в формуле (12.6). Результат суммирования затем нужно будет умножить на $C$ и сложить соответственно с $u$, $v$ или $b$.

| | для $u$ | для $v$ | для $b$ |
|---|---|---|---|
| if $u + 4v + b \geq +1$ then 0 else | –1 | –4 | –1 |
| if $2u + 2v + b \geq +1$ then 0 else | –2 | –2 | –1 |
| if $3u + 4v + b \geq +1$ then 0 else | –3 | –4 | –1 |
| if $u + v + b \leq -1$ then 0 else | +1 | +1 | +1 |
| if $2u + v + b \leq -1$ then 0 else | +2 | +1 | +1 |
| if $3u + v + b \leq -1$ then 0 else | +3 | +1 | +1 |

**Рис. 12.19.** Просуммировать все эти члены и умножить на $C$ для получения вкладов плохих точек в частные производные $f$ по $u$, $v$ и $b$

Истинность или ложность каждого из шести условий на рис. 12.19 определяет вклад соответствующего члена в сумму по $i$ в формуле (12.6). Будем представлять состояние каждого условия последовательностью $x$ и $o$, где $x$ означает, что условие не выполняется, а $o$ – что выполняется.

На рис. 12.20 показано несколько первых итераций градиентного спуска.

| | **w** = [*u*, *v*] | *b* | Плохая | ∂/∂u | ∂/∂v | ∂/∂b |
|---|---|---|---|---|---|---|
| (1) | [0.000, 1.000] | −2.000 | охоооо | −0.200 | 0.800 | −2.100 |
| (2) | [0.040, 0.840] | −1.580 | охоххх | 0.440 | 0.940 | −1.380 |
| (3) | [−0.048, 0.652] | −1.304 | охоххх | 0.352 | 0.752 | −1.104 |
| (4) | [−0.118, 0.502] | −1.083 | хххххх | −0.118 | −0.198 | −1.083 |
| (5) | [−0.094, 0.542] | −0.866 | охоххх | 0.306 | 0.642 | −0.666 |
| (6) | [−0.155, 0.414] | −0.733 | хххххх | | | |

**Рис. 12.20.** Начало процесса градиентного спуска

Рассмотрим строку (1). В ней показано начальное значение **w** = [0, 1]. Напомним, что элементы вектора **w** мы обозначаем *u* и *v*, поэтому *u* = 0 и *v* = 1. Показано также начальное значение *b* = −2. Эти значения *u* и *v* используется для вычисления условий на рис. 12.19. Первое условие имеет вид $u + 4v + b \geq +1$. Левая часть равна 0 + 4 + (−2) = 2, поэтому условие выполнено. Однако второе условие, $2u + 2v + b \geq +1$, не выполнено, т. к. его левая часть равна 0 + 2 + (−2) = 0. Если сумма равна 0, то вторая точка (2, 2) лежит точно на разделяющей гиперплоскости, а не за пределами зазора. Третье условие выполнено, т. к. 0 + 4 + (−2) = 2 ≥ +1. Последние три условия также выполнены и даже обращаются в равенства. Например, четвертое условие имеет вид $u + v + b \leq -1$. Его левая часть равна 0 + 1 + (−2) = −1. Таким образом, все шесть условий представлены последовательностью охоооо, как показано на рис. 12.20.

Эти условия мы используем для вычисления частных производных. При вычислении $\partial f/\partial u$ подставляем *u* вместо $w_j$ в выражение (12.6), которое принимает вид

$$u + C(0 + (-2) + 0 + 0 + 0 + 0) = 0 + 1/10\ (-2) = -0.2.$$

Сумма, умножаемая на *C*, получается следующим образом. Для каждого из шести условий на рис. 12.19, берем 0, если условие выполнено, или значение в столбце «для *u*», если не выполнено. Аналогично, подставляя *v* вместо $w_j$, получаем $\partial f/\partial v = 1 + 1/10\ (0 + (-2) + 0 + 0 + 0 + 0) = 0.8$. Наконец, для *b* получаем $\partial f/\partial b = -2 + 1/10\ (0 + (-1) + 0 + 0 + 0 + 0) = -2.1$.

Теперь можно вычислить новые значения **w** и *b*, показанные в строке (2) на рис. 12.20. Поскольку мы взяли $\eta = 1/5$, то новое значение *u* равно 0 − 1/5 (−0.2) = −0.04, новое значение *v* равно 1 − 1/5 (0.8) = 0.84, а новое значение *b* равно −2 − 1/5 (−2.1) = −1.58.

Для вычисления производных, показанных в строке (2) на рис. 12.20, мы должны сначала проверить условия на рис. 12.19. Результаты вычисления первых трех условий не изменились, но три последних больше не выполняются. Например, четвертое условие имеет вид $u + v + b \leq -1$, но 0.04 + 0.84 + (−1.58) = −0.7, а это число больше −1. Таким образом, последовательность плохих точек теперь выглядит так: охоххх. В выражениях для частных производных стало больше ненулевых членов. Например, $\partial f/\partial u$ = 0.04 + 1/10 (0 + (−2) + 0 + 1 + 2 + 3) = 0.44.

Значения **w** и $b$ в строке (3) вычисляются по производным в строке (2) точно так же, как было описано выше. Новые значения не изменяют последовательность плохих точек, она по-прежнему имеет вид охоххх. Но, повторив процесс для строки (4), мы обнаружим, что все шесть условий не выполнены. Например, первое условие имеет вид $u + 4v + b \geq +1$, поскольку $(-0.118 + 4 \times 0.502 + (-1.083)) = 0.807$, т. е. меньше 1. Это означает, что первая точка слишком приблизилась к разделяющей гиперплоскости, хотя классифицирована правильно.

Мы видим, что в строке (5) проблемы с первой и третьей точкой исправлены, и мы вернулись к последовательности охоххх. Но в строке (6) точки снова оказались слишком близко к разделяющей гиперплоскости, о чем свидетельствует вновь возникшая последовательность плохих точек хххххх. Предлагаем вам выполнить еще несколько итераций обновления **w** и $b$.

Возникает вопрос, почему метод градиентного спуска, похоже, сходится к решению, в котором по меньшей мере несколько точек оказываются внутри зазора, хотя имеется очевидная гиперплоскость (горизонтальная, на высоте 1.5) с зазором 1/2, разделяющая положительные и отрицательные примеры. Причина в том, что, выбрав $C = 0.1$, мы сказали, что нам не так важно, есть ли точки внутри зазора, мы даже согласны на неправильную классификацию. А важно лишь, чтобы сам зазор был большим (что соответствует малой норме $\|\mathbf{w}\|$), пусть даже некоторые точки оказываются внутри него.

## 12.3.5. Стохастический градиентный спуск

Алгоритм градиентного спуска, описанный в разделе 12.3.4, часто называют *пакетным* градиентным спуском, поскольку на каждой итерации все обучающие примеры рассматриваются одним «пакетом». Для небольших наборов данных это эффективно, но если набор очень велик, то на многократную – до достижения сходимости – обработку каждого обучающего примера может уходить слишком много времени.

В *стохастическом* методе градиентного спуска рассматривается один или несколько обучающих примеров за раз, и текущая оценка функции ошибок (**w** в случае метода опорных векторов) корректируется в направлении, определяемом небольшой рассмотренной выборкой обучающих примеров. Возможны дополнительные итерации с использованием других наборов обучающих примеров, которые можно выбирать случайно или следуя определенной стратегии. При этом вполне может случиться, что некоторые члены обучающего набора вообще не будут участвовать в алгоритме стохастического градиентного спуска.

**Пример 12.10.** Вспомним алгоритм UV-декомпозиции из раздела 9.4.3. Он был описан как пример пакетного градиентного спуска. Каждый непустой элемент матрицы $M$, которую мы пытаемся аппроксимировать произведением $UV$, можно рассматривать как обучающий пример, а функцией ошибок считать среднеквадратичную ошибку между произведением текущих матриц $U$ и $V$ и матрицей $M$, принимая во внимание только непустые элементы $M$.

Но если $M$ содержит очень много непустых элементов, как в случае, когда $M$ представляет товары, купленные пользователями Amazon, или фильмы, оцененные пользователями Netflix, то практически нереально повторно перебирать все непустые элементы $M$ в ходе корректировки элементов $U$ и $V$. При стохастическом градиентном спуске мы возьмем только один непустой элемент $M$ и вычислим, как нужно изменить каждый элемент $U$ и $V$, чтобы произведение $UV$ было согласовано с этим элементом. При этом мы не станем вносить изменения в элементы $U$ и $V$ в полном объеме, а выберем некоторую скорость обучения $\eta$, меньшую 1, и внесем в каждый элемент $U$ и $V$ лишь долю $\eta$ от величины изменения, при которой $UV$ и $M$ совпали бы в выбранном элементе.

## 12.3.6. Параллельная реализация метода опорных векторов

Один из подходов к распараллеливанию метода SVM аналогичен предложенному для перцептронов в разделе 12.2.8. Мы можем начать с текущих $\mathbf{w}$ и $b$ и параллельно выполнить несколько итераций для каждого обучающего примера. Затем построим новые $\mathbf{w}$ и $b$, усреднив изменения, вычисленные для каждого примера. Если раздать $\mathbf{w}$ и $b$ каждому распределителю, то распределители смогут выполнить в одном раунде столько итераций, сколько мы захотим, а редукторам останется только усреднение результатов. Для каждого раунда необходимо одна итерация MapReduce.

Другой подход состоит в том, чтобы последовать приведенному здесь рецепту, но второй член выражения (12.4) вычислять параллельно. Затем можно будет просуммировать вклады всех обучающих примеров. При таком подходе потребуется один раунд MapReduce для каждой итерации градиентного спуска.

## 12.3.7. Упражнения к разделу 12.3

**Упражнение 12.3.1.** Выполните еще несколько итераций, продолжив рис. 12.20.

**Упражнение 12.3.2.** Приведенные ниже обучающие наборы подчиняются следующему правилу: для положительных примеров сумма всех элементов вектора не меньше 10, а для отрицательных меньше 10.

$$([3,4,5],+1) \quad ([2,7,2],+1) \quad ([5,5,5],+1)$$
$$([1,2,3],-1) \quad ([3,3,2],-1) \quad ([2,4,1],-1)$$

(*а*) Какие из этих шести векторов являются опорными?

! (*б*) Предложите вектор $\mathbf{w}$ и константу $b$ такие, что гиперплоскость с уравнением $\mathbf{w}\cdot\mathbf{x} + b = 0$ хорошо разделяет положительные и отрицательные примеры. Позаботьтесь о том, чтобы масштаб $\mathbf{w}$ был таким, чтобы все точки оказались вне зазора, т. е. для каждого обучающего примера $(\mathbf{x}, y)$ должно выполняться условие $y(\mathbf{w}\cdot\mathbf{x} + b) \geq +1$.

! (*в*) Начав с решения задачи (*б*), воспользуйтесь градиентным спуском для нахождения оптимальных **w** и $b$. Заметим, что если начать с разделяющей гиперплоскости и правильно масштабировать **w**, то второй член в выражении (12.4) всегда будет равен 0, что существенно упрощает задачу.

**! Упражнение 12.3.3.** Приведенные ниже обучающие наборы подчиняются следующему правилу: для положительных примеров сумма всех элементов вектора нечетна, а для отрицательных четна.

([1, 2],+1) ([3, 4],+1) ([5, 2],+1)
([2, 4],–1) ([3, 1],–1) ([7, 3],–1)

(*а*) Предложите начальный вектор **w** и константу $b$, при которых правильно классифицируются хотя бы три точки.

!!(*б*) Начав с решения задачи (*а*), воспользуйтесь градиентным спуском для нахождения оптимальных **w** и $b$.

# 12.4. Обучение по ближайшим соседям

В этом разделе мы рассмотрим несколько примеров «обучения», когда весь обучающий набор, быть может, как-то предварительно обработанный, хранится в памяти, а затем используется для классификации будущих примеров или для вычисления наиболее вероятной метки нового примера. Вектор признаков каждого обучающего примера рассматривается как точка в некотором пространстве. Когда поступает новый пример, подлежащий классификации, мы находим одну или несколько точек, ближайших к нему относительно метрики этого пространства. Оценка метки затем вычисляется путем того или иного комбинирования ближайших примеров.

## 12.4.1. Инфраструктура для вычисления ближайших соседей

Сначала обучающий набор подвергается предварительной обработке и сохраняется в памяти. Решения принимаются, когда поступает новый подлежащий классификации пример, называемый *запросом*.

При проектировании основанного на ближайших соседях алгоритма классификации запросов необходимо принять несколько решений.

1. Какую метрику использовать?
2. Сколько ближайших соседей рассматривать?
3. Как взвешивать ближайших соседей? Обычно для этой цели предоставляется функция (*ядерная функция*) расстояния между запросом и ближайшими к нему соседями в обучающем наборе.
4. Как определяется метка, ассоциированная с запросом? Она должна быть какой-то функцией от меток ближайших соседей, взвешенной с помощью

ядра или не взвешенной. Если взвешивание не производится, то задавать ядерную функцию необязательно.

## 12.4.2. Обучение по одному ближайшему соседу

Простейший случай обучения по ближайшим соседям – взять одного соседа, ближайшего к запросу. В этом случае взвешивать соседей не нужно, поэтому ядерную функцию можно опустить. В типичном случае метку тоже можно выбрать единственным способом: взять метку ближайшего соседа.

**Пример 12.11.** На рис. 12.21 показаны примеры, связанные с собаками; последний раз мы встречались с ними на рис. 12.1. Некоторые примеры мы для простоты опустили, оставив только трех чихуахуа, двух такс и двух биглей. Поскольку векторы роста и веса, описывающие собак, двумерны, существует простой и эффективный способ построения диаграммы Вороного для этих точек: восставить перпендикуляры из середины каждого отрезка, соединяющего пару точек. Вокруг каждой точки образуется область, содержащая все ближайшие к ней точки. Эти области всегда выпуклы, хотя в одном направлении могут простираться в бесконечность[3]. Удивителен также тот факт, что хотя для $n$ точек существует $O(n^2)$ срединных перпендикуляров, диаграмму Вороного можно построить за время $O(n \log n)$.

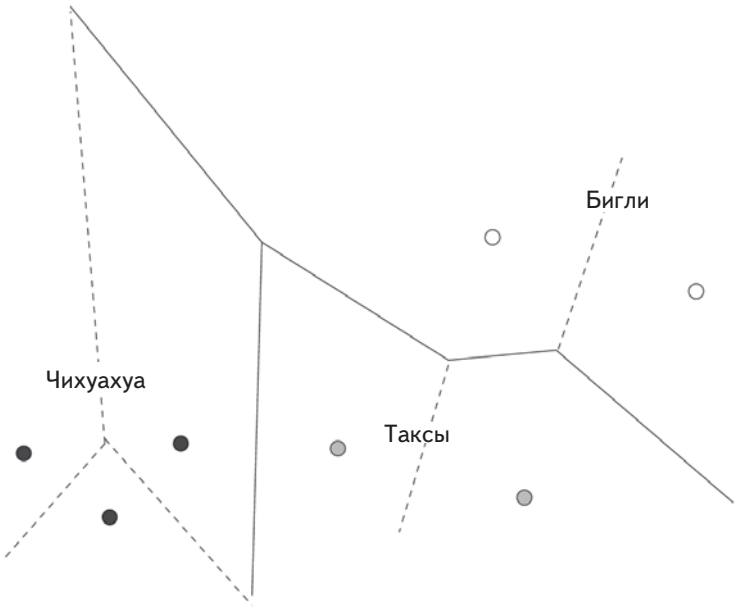



**Рис. 12.21.** Диаграмма Вороного для трех пород собак

[3] Область, окружающая одну точку, выпукла, но объединение областей, соответствующих двум и более точкам, может и не быть выпуклым. Так, на рис. 12.21 мы видим, что область всех такс и область всех биглей не являются выпуклыми, т. е. существует точки $p_1$ и $p_2$, классифицированные как таксы, но такие, что середина отрезка между ними классифицирована как бигль. И наоборот.

На рис. 12.21 изображена диаграмма Вороного для семи точек. Границы, разделяющие собак разных пород, показаны сплошными линиями, а границы между собаками одной породы – штриховыми. Допустим, что поступил запрос $q$. Заметим, что $q$ – точка в пространстве на рис. 12.21. Находим область, в которую попадает $q$, и назначаем $q$ метку того обучающего примера, которому эта область принадлежит. Отметим, что найти область $q$ не так уж трудно. Мы должны определить, по какую сторону от некоторых прямых находится $q$. Это тот же процесс, который мы применяли в разделах 12.2 и 12.3 для сравнения вектора **x** с гиперплоскостью, перпендикулярной вектору **w**. На самом деле, если прямые, по которым проходят границы диаграммы Вороного, предварительно обработаны надлежащим образом, то для принятия решения потребуется $O(\log n)$ сравнений; вовсе необязательно сравнивать $q$ со всеми $O(n \log n)$ прямыми, образующими диаграмму.

## 12.4.3. Обучение одномерных функций

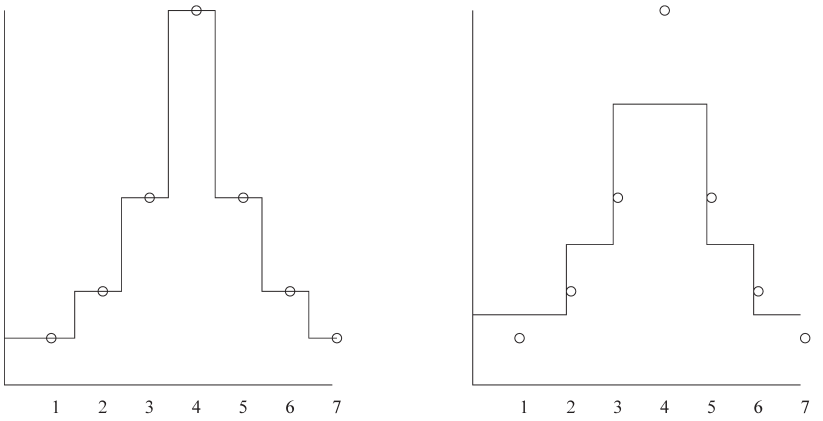
В еще одном простом и полезном примере обучения по ближайшим соседям данные одномерные. Обучающие примеры имеют вид $([x], y)$, и мы будем записывать их просто как $(x, y)$, где $x$ – единственный элемент одномерного вектора. По существу, обучающий набор представляет собой выборку значений функции $y = f(x)$ для некоторых значений $x$, а задача состоит в том, чтобы интерполировать функцию $f$ во всех точках. Решать ее можно разными способами, мы упомянем некоторые наиболее популярные. В разделе 12.4.1 отмечалось, что алгоритм классификации характеризуется несколькими факторами: количество используемых соседей, взвешиваются соседи или нет, и, если взвешиваются, то как вес зависит от расстояния.

Предположим, что используется метод с $k$ ближайшими соседями, и $x$ – запрос. Пусть $x_1, x_2, ..., x_k$ – $k$ ближайших соседей $x$, а $w_i$ – вес, ассоциированный с обучающим примером $(x_i, y_i)$. Тогда в качестве оценки метки $y$ для $x$ берется сумма $\sum_{i=1}^{k} w_i y_i / \sum_{i=1}^{k} w_i$. Это не что иное как средневзвешенное меток $k$ ближайших соседей.

**Пример 12.12.** Проиллюстрируем четыре простых подхода на примере обучающего набора (1, 1), (2, 2), (3, 4), (4, 8), (5, 4), (6, 2), (7, 1). Эти точки представляют функцию, которая имеет пик в точке $x = 4$ и экспоненциально убывает с обеих сторон от нее. Отметим, что промежутки между значениями $x$ одинаковы, но в общем случае не предъявляется никаких требований к регулярности выборки точек. Ниже описаны возможные способы интерполяции.

1. *Ближайший сосед.* Используется только один ближайший сосед. Во взвешивании нет необходимости. Просто берем в качестве $f(x)$ метку $y$, ассоциированную с точкой из обучающего набора, ближайшей к $x$. Результат применения этого правила к описанному выше обучающему набору показан на рис. 12.22($a$).

2. *Среднее двух ближайших соседей*. В качестве числа ближайших соседей берется 2. Каждому соседу присваивается вес 1/2 вне зависимости от их расстояния до $x$. Результат применения этого правила к описанному выше обучающему набору показан на рис. 12.22(*б*).




**(*а*)** Один ближайший сосед **(*б*)** Среднее двух ближайших соседей

**Рис. 12.22.** Результаты применения первых двух правил к примеру 12.12

3. *Средневзвешенное двух ближайших соседей*. Снова берем двух ближайших соседей, но присваиваем им веса, обратно пропорциональные расстоянию до точки запроса. Пусть ближайшими к $x$ являются точки $x_1$ и $x_2$. Сначала предположим, что $x_1 < x < x_2$. Тогда вес $x_1$, обратно пропорциональный ее расстоянию до $x$, равен $1/(x - x_1)$, а вес $x_2$ равен $1/(x_2 - x)$. Средневзвешенное двух меток равно

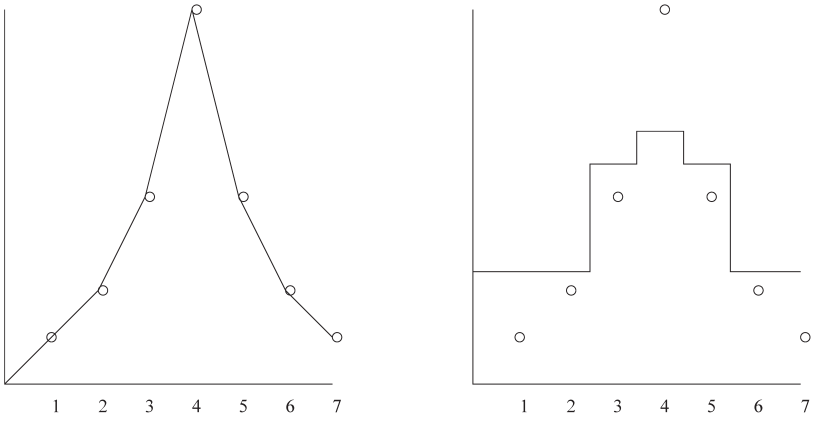
$$\left(\frac{y_1}{x - x_1} + \frac{y_2}{x_2 - x}\right) / \left(\frac{1}{x - x_1} + \frac{1}{x_2 - x}\right).$$

После умножения числителя и знаменателя на $(x - x_1)(x_2 - x)$ эта формула принимает вид

$$\frac{y_1(x_2 - x) + y_2(x - x_1)}{x_2 - x_1}.$$

Это линейная интерполяция двух ближайших соседей, она показана на рис. 12.23(*а*). Если оба ближайших соседа находятся по одну сторону от $x$, то те же самые веса имеют смысл, а результирующая оценка называется *экстраполяцией*. Пример экстраполяции мы видим на том же рис. 12.23(*а*) в диапазоне от $x = 0$ до $x = 1$. В общем случае, когда точки не равноотстоящие, можно найти и внутренние точки, для которых оба ближайших соседа находятся по одну сторону.

4. *Среднее трех ближайших соседей.* Мы можем усреднить любое количество ближайших соседей для оценки метки запроса. На рис. 12.23(*б*) показан результат усреднения нашего обучающего набора по трем ближайшим соседям.




**(*а*)** Средневзвешенное двух соседей **(*б*)** Среднее трех соседей

**Рис. 12.23.** Результаты применения последних двух правил к примеру 12.12

## 12.4.4. Ядерная регрессия

Для построения непрерывной функции, хорошо представляющей данные из обучающего набора, можно рассмотреть все обучающие примеры, но взвешивать точки с помощью функции, убывающей с расстоянием. Часто в качестве такой функции берут кривую нормального распределения («колоколообразную кривую»), так что вес обучающего примера $x$ относительно запроса $q$ равен $e^{-(x-q)^2/\sigma^2}$. Здесь $\sigma$ – стандартное отклонение, а $q$ – среднее распределения. Грубо говоря, точки, удаленные от $q$ не более чем на $\sigma$, имеют большой вес, а отстоящие дальше – малый. Преимущество использования непрерывной ядерной функции, определенной для всех точек обучающего набора, заключается в том, что полученная в результате обучения функция тоже будет непрерывной (см. упражнение 12.4.6, где обсуждается проблема, возникающая при использовании более простого взвешивания).

**Пример 12.13.** Возьмем семь точек из примера 12.12. Чтобы упростить вычисления, будем использовать в качестве ядерной функции не нормальное распределение, а другую непрерывную функцию расстояния – $w = 1/(x - q)^2$. Это означает, что веса убывают пропорционально квадрату расстояния. Рассмотрим запрос $q = 3.5$. Веса $w_1$, $w_2$, ..., $w_7$ семи обучающих примеров $(x_i, y_i) = (i, 8/2^{|i-4|})$ для $i = 1, 2, ..., 7$ приведены в таблице на рис. 12.24.

| (1) | $x_i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| (2) | $y_i$ | 1 | 2 | 4 | 8 | 4 | 2 | 1 |
| (3) | $w_i$ | 4/25 | 4/9 | 4 | 4 | 4/9 | 4/25 | 4/49 |
| (4) | $w_i y_i$ | 4/25 | 8/9 | 16 | 32 | 16/9 | 8/25 | 4/49 |

**Рис. 12.24.** Веса точек относительно запроса $q = 3.5$

В строках (1) и (2) на рис. 12.24 приведены семь обучающих примеров. В строке (3) показаны их веса относительно запроса $q = 3.5$. Например, вес точки $x_1 = 1$ равен $w_1 = 1/(1 - 3.5)^2 = 1/(-2.5)^2 = 4/25$. В строке (4) приведены значения $y_i$, умноженные на вес в строке (3). Например, в столбце для $x_2$ находится значение 8/9, потому что $w_2 y_2 = 2 - (4/9)$.

Чтобы вычислить метку для запроса $q = 3.5$, мы складываем взвешенные значения всех меток из обучающего набора, приведенные в строке (4); эта сумма равна 51.23. Затем результат делится на сумму весов, приведенных в строке (3), она равна 9.29. Таким образом, частное равно 51.23/9.29 = 5.51. Интуитивно эта оценка метки для $q = 3.5$ кажется разумной, т. к. $q$ находится посередине между точками с метками 4 и 8.

## 12.4.5. Данные в многомерном евклидовом пространстве

В разделе 12.4.2 мы видели, что данные в двумерном евклидовом пространстве не представляют особой сложности. А для поиска ближайших соседей в многомерном случае, когда размер обучающего набора велик, разработано несколько специальных структур данных. Мы не будем их рассматривать, т. к. эта тема заслуживает отдельной книги, и подробно освещена в других местах под названием *многомерные индексы*. В списке литературы к этой главе упоминаются некоторые источники информации о таких структурах, как *kd*-деревья, R-деревья и квадродеревья.

### Проблемы с пределом в примере 12.13

Предположим, что $q$ в точности совпадает с одним из обучающих примеров $x$. Если в качестве ядерной функции использовать нормальное распределение, то проблемы с весом $x$ не возникает: он равен 1. Если же взять ядерную функцию из примера 12.13, то вес $x$ будет равен $1/(x - q)^2 = \infty$. По счастью, этот вес встречается как в числителе, так и в знаменателе выражения для оценки метки $q$. Можно показать, что когда $q$ стремится к $x$, метка $x$ доминирует над всеми членами в числителе и в знаменателе, поэтому оценка метки $q$ совпадает с меткой $x$. И это вполне разумно, потому что в пределе $q = x$.

К сожалению, в многомерном случае мало что можно сделать, чтобы избежать поиска в большой части данных. И это еще одно проявление «проклятия размерности», о котором мы говорили в разделе 7.1.3. Одолеть «проклятие» можно двумя способами.

1. *VA-файлы.* Поскольку нам в любом случае придется просматривать значительную часть данных для поиска ближайших к запросу соседей, можно было бы вообще обойтись без сложной структуры данных. Смиримся с тем, что предстоит просматривать большой файл, но сделаем это в два этапа. Сначала создадим сводку файла, используя небольшое количество битов для аппроксимации значений каждого элемента каждого обучающего вектора. Например, если брать только одну четверть старших битов числовых элементов, то можно создать файл, размер которого будет в четыре раза меньше исходного. Однако при просмотре этого файла мы можем построить список кандидатов, среди которых *могут* находиться $k$ ближайших соседей запроса $q$, и этот список может составлять малую толику от всего набора данных. Затем в полном файле мы ищем ближайших соседей только среди этих кандидатов и больше ничего не просматриваем.
2. *Понижение размерности.* Мы можем рассматривать векторы из обучающего набора как матрицу, строками которой являются обучающие примеры, а столбцы соответствуют их элементам. Применим один из способов понижения размерности, рассмотренных в главе 11, чтобы сжать векторы до такой степени, когда становятся практически применимы методы многомерного индексирования. Разумеется, при обработке вектора запроса $q$ к нему необходимо применить точно такое же преобразование и только потом искать ближайших соседей.

## 12.4.6. Неевклидовы метрики

До сих пор мы предполагали, что метрика евклидова. Но большинство методов естественно адаптируется к произвольной метрике $d$. Например, в разделе 12.4.4 мы упоминали функцию нормального распределения в качестве ядерной. Поскольку речь шла об обучающем наборе в одномерном евклидовом пространстве, то мы записывали показатель степени в виде $-(x - q)^2$. Однако для любой метрики $d$ можно было в качестве веса точки $x$ на расстоянии $d(x, q)$ от запроса $q$ взять значение

$$e^{-(d(x-q))^2}/\sigma^2.$$

Отметим, что это выражение имеет смысл, если данные принадлежат какому-нибудь многомерному евклидову пространству, а $d$ – обычная евклидова метрика, или манхэттенское расстояние или еще какое-то расстояние из числа рассмотренных в разделе 3.5.2. Оно имеет смысл и для расстояния Жаккара, и любой другой метрики.

Однако для расстояния Жаккара и других метрик, обсуждавшихся в разделе 3.5, у нас есть еще возможность применить хэширование с учетом близости –

тема главы 3. Напомним, впрочем, что это лишь приближенные методы, которые могут давать ложноотрицательные результаты – обучающие примеры, являющиеся близкими соседями запроса, но не попавшие в результаты поиска.

Если мы готовы смириться с небольшим числом таких ошибок, то можем построить для обучающего набора ячейки и хранить их в качестве представления этого набора. Ячейки устроены так, чтобы можно было найти все (или почти все, поскольку возможны ложноотрицательные результаты) точки из обучающего набора, сходство которых с запросом $q$ не ниже заданного. Или эквивалентно, одна из ячеек, в которую хэшируется запрос, будет содержать все точки, отстоящие от $q$ не дальше определенного порога. Мы надеемся, что в этих ячейках окажется столько ближайших соседей $q$, сколько требуется применяемому методу.

Но даже если для различных запросов расстояния до ближайших соседей различаются очень сильно, не все потеряно. Мы можем выбрать несколько расстояний $d_1 < d_2 < d_3 < ....$ Построим ячейки для хэширования с учетом близости, воспользовавшись каждым из этих расстояний. Получив запрос $q$, начнем с ячеек для расстояния $d_1$. Если найдется достаточно близких соседей, то на этом можно закончить. В противном случае повторим поиск, используя ячейки для $d_2$, и так далее, пока не найдем достаточное количество ближайших соседей.

## 12.4.7. Упражнения к разделу 12.4

**Упражнение 12.4.1.** Допустим, что мы модифицировали пример 12.11 и рассматриваем двух ближайших соседей запроса $q$. Если метки обоих соседей одинаковы, то эту общую метку берем в качестве класса $q$, в противном случае оставляем $q$ не классифицированным.

(*а*) Нарисуйте эскиз границ областей для всех трех пород собак на рис. 12.21.

! (*б*) Верно ли, что любого обучающего набора границы состоят из отрезков прямых?

**Упражнение 12.4.2.** Пусть имеется обучающий набор

([1, 2],+1) ([2, 1],–1)
([3, 4],–1) ([4, 3],+1)

взятый из примера 12.9. Если в качестве метки запроса брать метку одного ближайшего соседа, то какие запросы получат метку +1?

**Упражнение 12.4.3.** Рассмотрим одномерный обучающий набор

(1, 1), (2, 2), (4, 3), (8, 4), (16, 5), (32, 6)

Опишите функцию $f(q)$, возвращающую метку запроса $q$, если используется следующий метод интерполяции:

(*а*) метка ближайшего соседа;

(*б*) среднее меток двух ближайших соседей;

! (*в*) взвешенное по расстоянию среднее двух ближайших соседей;

(*г*) обычное (не взвешенное) среднее трех ближайших соседей.

**! Упражнение 12.4.4.** Примените ядерную функцию из примера 12.13 к данным из упражнения 12.4.3. Какова будет метка $q$ для запросов в диапазоне $2 < q < 4$?

**Упражнение 12.4.5.** Какая функция оценивает метки запросов для данных из примера 12.12 и интерполяции по среднему четырех ближайших соседей?

**!! Упражнение 12.4.6.** Для простых весовых функций типа тех, что использовались в примере 12.12, определять непрерывную функцию необязательно. Мы видим, что функции, построенные на рис. 12.22 и 12.23(*б*) не являются непрерывными, но функция, изображенная на рис. 12.23(*а*), непрерывна. Верно ли, что средневзвешенное двух ближайших соседей всегда дает непрерывную функцию?

# 12.5. Сравнение методов обучения

У каждого из методов, рассмотренных в этой главе и других главах, есть свои преимущества. В этом заключительном разделе мы обсудим следующие вопросы:

- может метод работать с категориальными признаками или только с числовыми?
- эффективен ли метод для работы с многомерными векторами признаков?
- можно ли сказать, что модель, которую строит метод, интуитивно понятна?

**Перцептроны и метод опорных векторов.** Эти методы справляются с миллионами признаков, но только числовых. Они эффективны лишь в случае, когда имеется линейный разделитель или, по крайней мере, гиперплоскость, приближенно разделяющая классы. Однако точки можно разделить и нелинейной границей, если сначала преобразовать их, так чтобы разделитель стал линейным. Модель описывается вектором, нормальным к разделяющей гиперплоскости. Поскольку размерность этого вектора зачастую очень высока, интерпретировать модель крайне трудно.

**Классификация по ближайшим соседям и регрессия.** В этом случае моделью является сам обучающий набор, так что по идее он должен быть интуитивно понятен. Метод может работать с многомерными данными, хотя чем больше размерность, тем более разреженным будет обучающий набор и тем меньше вероятность найти обучающий пример, очень близкий к классифицируемому. То есть из-за «проклятия размерности» применение методов классификации по ближайшим соседям в многомерных пространствах под большим вопросом. Эти методы по-настоящему полезны только для числовых признаков, хотя и допускают категориальные признаки с небольшим числом значений. Например, значения бинарного категориального признака типа {мужчина, женщина} можно было бы заменить на 0 и 1, так что расстояние между лицами одного пола в таком пространстве равно 0, а между лицами разного пола – 1. Однако при наличии трех и более значений уже нельзя заменить значения эквидистантными числами. Наконец, у методов классификации по ближайшим соседям много настраиваемых параметров, в том числе: метрика (например, косинусная или евклидова), количество соседей и ядерная

функция. Результаты классификации сильно зависят от выбора параметров, и во многих случаях не очевидно, какой выбор приведет к оптимальным результатам.

**Решающие деревья.** Мы не обсуждали этот широко применяемый метод в этой главе, хотя кратко затронули его в разделе 9.2.7. В отличие от методов, описанных здесь, решающие деревья применимы как к категориальным, так и к числовым признакам. Порождаемые ими модели обычно вполне понятны, поскольку каждое решение представляется одним узлом дерева. Однако это подход применим только к векторам признаков малой размерности. Причина в том, что построение решающего дерева с большим числом уровней ведет к переобучению, так как после нескольких верхних уровней решения начинают зависеть от особенностей небольших подмножеств обучающего набора, а не от фундаментальных свойств данных. С другой стороны, если в решающем дереве немного уровней, то многие признаки даже не найдут отражения. Поэтому решающие деревья лучше всего использовать для построения ансамбля из многих деревьев небольшой глубины и каким-то образом комбинировать полученные результаты.

# 12.6. Резюме

- *Обучающие наборы.* Обучающий набор состоит из векторов признаков, с каждым из которых ассоциирована метка, показывающая, к какому классу относится объект, представленный данным вектором. Признаки могут быть категориальными – с дискретным списком значений – или числовыми.
- *Тестовые наборы и переобучение.* При обучении классификатора полезно зарезервировать часть обучающего набора и использовать ее в качестве тестового. Обученный классификатор можно проверить на тестовом наборе. Если на тестовом наборе он работает хуже, чем на обучающем, значит, имеет место переобучение, т. е. улавливание особенностей обучающего набора, не характерных для данных в целом.
- *Пакетное и оперативное обучение.* При пакетном обучении обучающий набор доступен постоянно, его можно повторно использовать на нескольких проходах. При оперативном обучении обучающие примеры поступают потоком, так что каждый можно использовать только один раз.
- *Перцептроны.* В этом методе машинного обучения предполагается, что в обучающих данных есть всего две метки класса: положительная и отрицательная. Перцептрон работает, если существует гиперплоскость, разделяющая векторы признаков положительных и отрицательных примеров. Сходимость к этой гиперплоскости обеспечивается путем корректировки ее оценки в направлении среднего текущего множества неправильно классифицированных точек с некоторым коэффициентом, определяющим скорость обучения.
- *Алгоритм Winnow.* Это вариант перцептрона, в котором требуется, чтобы элементы вектора были равны 0 или 1. Обучающие примеры перебирают-

ся циклически и, если текущая классификация примера неправильна, то элементы оценки разделителя в тех позициях, где в векторе признаков находится 1, уменьшаются или увеличиваются в направлении, повышающем вероятность правильной классификации этого примера на следующем раунде.

- *Нелинейные разделители.* Если для обучающего набора не существует линейной функции, разделяющей два класса, то иногда все равно можно использовать для их классификации перцептрон. Необходимо найти функцию, преобразующую точки, так что в преобразованном пространстве разделителем является гиперплоскость.
- *Метод опорных векторов.* Метод опорных векторов (SVM) является усовершенствованием перцептрона в том смысле, что не только находит гиперплоскость, разделяющую положительные и отрицательные примеры, но и максимизирует зазор – расстояние по перпендикуляру от гиперплоскости до ближайших к ней точек. Точки, до которых расстояние минимально, называются опорными векторами. Можно также настроить алгоритм опорных векторов, так что некоторые точки могут находиться близко к гиперплоскости или даже не по ту сторону от нее, но при этом минимизируется ошибка, связанная с такими неправильно расположенными точками.
- *Решение уравнений метода опорных векторов.* Можно построить функцию, зависящую от вектора, нормального к гиперплоскости, длины этого вектора (определяющей зазор) и штрафа для точек, оказавшихся не с той стороны гиперплоскости, что нужно. Параметр регуляризации определяет относительную важность широкого зазора и малого штрафа. Получающиеся уравнения можно решать несколькими способами, в том числе методом градиентного спуска или квадратичного программирования.
- *Обучение по ближайшим соседям.* В этом подходе к машинному обучению весь обучающий набор рассматривается как модель. Для каждой подлежащей классификации точки (запроса) мы ищем $k$ ближайших к ней точек из обучающего набора. Классификация запроса является функцией от меток этих $k$ соседей. В простейшем случае, когда $k = 1$, мы можем в качестве метки запроса взять метку ближайшего соседа.
- *Регрессия.* Распространенный случай обучения по ближайшим соседям, называемый регрессией, имеет место, когда вектор содержит только один признак, и этот признак, равно как и метка, – вещественное число, т. е. данные определяют вещественную функцию одной переменной. Чтобы оценить метку, т. е. значение функции, непомеченной точки, мы можем выполнить некоторое вычисление, в котором участвуют $k$ ближайших соседей. Это может быть, в частности, простое усреднение или взятие взвешенного среднего, где вес соседа задается убывающей функцией расстояния до классифицируемой точки.

# 12.7. Список литературы

Понятие перцептрона было введено в работе [11]. В [7] впервые сформулирована идея максимизации зазора вокруг разделяющей гиперплоскости. Этой теме посвящена хорошо известная книга [10].

Алгоритм Winnow взят из работы [9]. См. также его анализ в работе [1].

Метод опорных векторов появился в работе [6]. Статьи [5] и [4] содержат полезные обзоры. В [8] обсуждается более эффективный алгоритм для случая разреженных признаков (когда большинство элементов вектора признаков равны 0). О методах градиентного спуска можно прочитать в работах [2, 3].

1. A. Blum «Empirical support for winnow and weighted-majority algorithms: results on a calendar scheduling domain», Machine Learning 26 (1997), pp. 5–23.
2. L. Bottou «Large-scale machine learning with stochastic gradient descent», Proc. 19th Intl. Conf. on Computational Statistics (2010), pp. 177–187, Springer.
3. L. Bottou «Stochastic gradient tricks, neural networks» in Tricks of the Trade, Reloaded, pp. 430–445, Edited by G. Montavon, G.B. Orr and K.-R. Mueller, Lecture Notes in Computer Science (LNCS 7700), Springer, 2012.
4. C.J.C. Burges «A tutorial on support vector machines for pattern recognition», Data Mining and Knowledge Discovery 2 (1998), pp. 121–167.
5. N. Cristianini, J. Shawe-Taylor «An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods», Cambridge University Press, 2000.
6. C. Cortes, V. N. Vapnik «Support-vector networks», Machine Learning 20 (1995), pp. 273–297.
7. Y. Freund, R. E. Schapire «Large margin classification using the perceptron algorithm», Machine Learning 37 (1999), pp. 277–296.
8. T. Joachims «Training linear SVMs in linear time», Proc. 12th ACM SIGKDD (2006), pp. 217–226.
9. N. Littlestone «Learning quickly when irrelevant attributes abound: a new linear-threshold algorithm», Machine Learning 2 (1988), pp. 285–318.
10. M. Minsky, S. Papert «Perceptrons: An Introduction to Computational Geometry» (2nd edition), MIT Press, Cambridge MA, 1972.
11. F. Rosenblatt «The perceptron: a probabilistic model for information storage and organization in the brain», Psychological Review 65:6 (1958), pp. 386–408.

# ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

## Е



## Ж



## З



## И



## К

## Л



## М



## Н



## О

## П

## Р



## С

## Т



## У



## Ф



## Х



## Ц



## Ч



## Ш



## Э



## Я



## A

## B



## C



## D



## F



## G



## H



## I



## K



## L



## M



## N



## O



## P



## R



## S



## T

## U



## V



## W



## Y

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. +7(499)782-38-89.

Электронный адрес: **books@alians-kniga.ru**.

Юре Лесковец, Ананд Раджараман, Джеффри Д. Ульман

**Анализ больших наборов данных**

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16. Гарнитура «Петербург».
Печать офсетная. Усл. печ. л. 40,46.
Тираж 100 экз.

Веб-сайт издательства: www.дмк.рф

# Анализ больших наборов данных


Данная книга представляет собой Стэнфордский курс о добыче данных в вебе (Web Mining) с акцентом на анализе данных очень большого объема. В книге принят алгоритмический подход: извлечение данных — это применение алгоритмов к данным, а не использование данных для «обучения» той или иной машины.

***Основные рассматриваемые темы:***

- распределенные файловые системы и технология распределения-редукции (map-reduce) как средство создания параллельных алгоритмов;
- поиск по сходству, в том числе MinHash и хэширование с учетом близости;
- обработка потоков данных и специализированные алгоритмы для работы с быстро поступающими данными;
- принципы работы поисковых систем, в том числе алгоритм Google Page-Rank, распознавание ссылочного спама и метод авторитетных и хаб-документов;
- частые предметные наборы, в том числе поиск ассоциативных правил, анализ корзины, алгоритм Apriori и его усовершенствованные варианты;
- алгоритмы кластеризации очень больших многомерных наборов данных;
- важные задачи: управление рекламой и рекомендательные системы;
- алгоритмы анализа структуры очень больших графов, в особенности графов социальных сетей;
- методы получения важных свойств большого набора данных с помощью понижения размерности;
- алгоритмы машинного обучения, применимые к очень большим наборам данных.

**Инетенет-магазин:**
www.dmkpress.com
**Книга – почтой:**
e-mail: orders@alians-kniga.ru
**Оптовая продажа:**
«Альянс-книга»
Тел./факс: (499) 782-3889
e-mail: books@alians-kniga.ru

ДМК ПРЕСС
издательство
www.дмк.рф

ISBN 978-5-97060-190-7
9 785970 601907