Теория и применение геоинформационных систем и технологий

Географические информационные системы (ГИС) приобретают все большее значение, помогая нам понять сложную социальную, экономическую и природную динамику в ситуациях, где ключевую роль играют пространственные компоненты. Однако фундаментальные алгоритмы, применяемые в ГИС, трудны для понимания в том числе из-за отсутствия логически последовательного изложения.

Настоящая книга – попытка решить эту проблему за счет сочетания строгого формализованного языка с практическими примерами и упражнениями.

Рассматриваемые темы:

- геометрические алгоритмы;
- индексирование пространственных данных;
- пространственный анализ и моделирование.

Теоретический материал подкрепляется кодом на языке Python.

Издание предназначено специалистам по анализу данных, разработчикам, студентам, а также всем, кто хочет разобраться в принципах работы ГИС.

Интернет-магазин:
www.dmkpress.com

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

SAGE Publishing

ДМК Пресс издательство
www.дмк.рф

ISBN 978-5-97060-908-8

АЛГОРИТМЫ ГИС

Нинчуань Сяо

АЛГОРИТМЫ ГИС

Теория и применение геоинформационных систем и технологий

ДМК Пресс издательство

Нинчуань Сяо

Алгоритмы ГИС

GIS Algorithms

Theory and Applications for Geographic Information Science & Technology

Ningchuan Xiao

SAGE

Los Angeles | London | New Delhi
Singapore | Washington DC

Алгоритмы ГИС

Теория и применение геоинформационных систем и технологий

Нинчуань Сяо

ДМК Пресс
издательство
Москва, 2021

УДК 528.1:379.85
ББК 26.8:32.81
С99

Сяо Н.
С99 Алгоритмы ГИС / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2021. – 328 с.: ил.

ISBN 978-5-97060-908-8

В этой книге обсуждаются концептуальные основы географических информационных систем (ГИС) – важнейшие алгоритмы, составляющие фундамент многих операций над пространственными данными. Проследив, как эти данные подаются на вход алгоритма и как алгоритм используется для получения конечного результата, читатель поймет два важных аспекта ГИС: что на самом деле представляют собой геопространственные данные и как они обрабатываются.

В книгу включены алгоритмы, которые обеспечивают измерение важных пространственных свойств (например, расстояния), включение нескольких источников данных с помощью слоев, ускорение анализа за счет применения различных методов индексирования. Уделено внимание алгоритмам решения таких задач пространственного анализа и моделирования, как интерполяция, анализ паттернов и принятие решений с помощью моделей оптимизации.

Издание будет полезно студентам и преподавателям географических и технических вузов, а также всем, кто хочет разобраться в принципах работы ГИС.

УДК 528.1:379.85
ББК 26.8:32.81

ISBN 978-1-491-97429-2 (англ.)
ISBN 978-5-97060-908-8 (рус.)

Посвящается Алестеру, все понимающему

Содержание

Об авторе

Нинчуань Сяо – доцент географического факультета Университета штата Огайо. Он читал разнообразные курсы по картографии, ГИС и пространственному анализу и моделированию. С 2009 по 2012 год занимал пост председателя специальной группы по пространственному анализу и моделированию Ассоциации американских географов. Исследования д-ра Сяо посвящены разработке действенных и эффективных вычислительных методов построения карт и анализа пространственно-временных данных в различных предметных областях, в т. ч. пространственной оптимизации, систем поддержки принятия решений о пространственном размещении, моделировании окружающей среды и экологической обстановки, пространственных моделях распространения эпидемий. Его работы публиковались в ведущих журналах по географии и ГИС. Его текущие проекты связаны с проектированием и реализацией новаторских подходов к анализу и нанесению на карту больших данных из социальных сетей и других онлайновых ресурсов, а также с разработкой поисковых алгоритмов для решения задач пространственного агрегирования. Он также работает в составе междисциплинарных групп над проектами отображения на картах влияния мобильности населения на распространение инфекционных заболеваний и построения моделей влияния окружающей среды на социальную динамику в Северном Камеруне.

Предисловие

Географические информационные системы (геоинформационные системы, ГИС) приобретают все большее значение, помогая нам понять сложную социальную, экономическую и природную динамику в ситуациях, где ключевую роль играют пространственные компоненты. В сравнительно короткой истории развития теории и приложений ГИС часто можно наблюдать процесс отчуждения ГИС как «механизма» от ее пользователей. В некоторых случаях ГИС свелась к черному ящику, который используется для порождения приятных глазу карт для различных целей. Эта тенденция уже проникла в наши учебные программы, в которых значительная доля времени уделяется тому, как научить студентов пользоваться графическими интерфейсами программных пакетов ГИС. Преодоление этой тенденции представляет серьезный вызов для исследователей и преподавателей ГИС.

В данной книге мы будем говорить о важнейших алгоритмах ГИС, являющихся фундаментом многих операций над пространственными данными. Научная дисциплина ГИС всегда была весьма разветвленной, и во многих учебниках общие вопросы рассматриваются поверхностно, лишая студентов возможности полностью вникнуть в концептуальные основы ГИС. Алгоритмы ГИС часто представляют разными способами с использованием различных структур данных, а отсутствие единого представления затрудняет понимание сути алгоритмов. Студенты, посещающие курсы ГИС, специализируются в разных областях знания и не всегда хорошо знакомы с терминами, используемыми при традиционном формальном описании алгоритмов. Из-за этого преподавание алгоритмов на курсах ГИС стало больной темой. Но это не должно служить оправданием исключению алгоритмов из курса. Проследив, как пространственные данные подаются на вход алгоритма и как алгоритм используется для обработки данных и получения конечного результата, мы сможем гораздо лучше понять два важных аспекта ГИС: что на самом деле представляют собой геопространственные данные и как эти данные в действительности обрабатываются.

В этой книге рассматриваются алгоритмы, критические для реализации некоторых базовых функций ГИС. Однако наша цель не в том, чтобы предъявить исчерпывающий длинный перечень алгоритмов. Мы включили только те алгоритмы, которые используются в большинстве современных ГИС или оказали существенное влияние на разработку текущих алгоритмов; поэтому выбор тем может показаться субъективным. Мы исповедуем минималистский взгляд на геопространственные данные, считая их данными о местоположении в пространстве, т. е. фокусируем внимание на координатах как атомарной единице географической информации, допуская, что в большинстве своем геопространственные данные можно рассматривать как наборы точек. Это позволяет в значительной степени избежать ненужных дискуссий о различии векторного и растрового представлений, сведя обсуждение к фундаментальной модели данных. Начав с этого места, мы приступим к изучению многообразных алгоритмов ГИС, которые помогают в выполнении базовых функций, как то: измерение важных пространственных свойств, например расстояния,

включение нескольких источников данных с помощью слоев, ускорение анализа за счет применения различных методов индексирования. Мы также уделим внимание алгоритмам решения таких задач пространственного анализа и моделирования, как интерполяция, анализ паттернов и принятие решений с помощью моделей оптимизации. Разумеется, все эти функции присутствуют во многих пакетах ГИС, как коммерческих, так и с открытым кодом. Однако наша цель – не дублировать то, что там есть, а показать, как это работает, чтобы мы могли реализовать собственную ГИС или, по крайней мере, некоторые ее функции, не полагаясь на программную систему, в названии которой встречается акроним ГИС. Чтобы обрести такую свободу, мы должны опуститься на уровень, где данные видны, а не скрыты, где процессы представлены в виде кода, а не кнопок, а выходные данные так же прозрачны, как входные.

Это не традиционная книга об алгоритмах. В типичной книге по информатике алгоритмы были бы представлены в виде псевдокода, в котором отражены наиболее важные части, а некоторые детали опущены. Но такой подход не годится для многих студентов, изучающих ГИС, поскольку зачастую теоретические аспекты алгоритмов интересуют их не в первую очередь. Стремление понять алгоритмы ГИС обычно проистекает из желания узнать, «как это работает». Поэтому для описания и реализации алгоритмов мы используем реальный работающий код. В качестве языка мы выбрали Python за его простоту, что позволяет не тратить много времени на изучение программирования как такового. Мы старались не слишком увлекаться мощными «пакетными» модулями Python, а показать, как на самом деле устроены алгоритмы. С той же целью мы с самого начала используем единое представление геопространственных данных и, стало быть, единые структуры данных.

Эта книга не появилась бы без помощи других людей. Я глубоко благодарен сообществу ПО с открытым исходным кодом, которое очень сильно поспособствовало формированию наших представлений о пространственных данных и их использовании, так что в этом смысле книга не состоялась бы без программ с открытым исходным кодом. По возникающим у меня вопросам касательно Python я неоднократно просил совета у пользователей сайта Stack Overflow. Онлайновое сообщество LATeX (http://tex.stackexchange.com и http://en.wikibooks.org/wiki/LaTeX) всегда было готово ответить на вопросы по поводу верстки в бессонные ночи, проведенные в работе над книгой. Некоторые открытые пакеты были особенно важны во многих частях книги, в т. ч. связанные с *k*D-деревьями (http://en.wikipedia.org/wiki/K-d_tree и https://code.google.com/p/python-kdtree/) и кригингом (https://github.com/cjohnson318/geostatsmodels). Выражаю также благодарность студентам многочисленных курсов, прочитанных мною в последние годы в США и Китае. Их отзывы, а иногда и критические замечания позволили мне улучшить реализации многих алгоритмов. Спасибо Мей-По Куань за поддержку и Ричарду Ли, Кэтрин Хоу и Мэттью Олдфилду за внимательное прочтение рукописи. Подробные замечания ряда рецензентов ранних вариантов книги существенно помогли мне улучшить текст. Наконец, я благодарен своей семье за терпение и поддержку на протяжении всего процесса работы над книгой.

Нинчуань Сяо
−82.8650°, 40.0556°
декабрь 2014

От издательства

Отзывы и пожелания

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

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

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и SAGE очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Глава 1

Введение

Алгоритмы[1] проектируются для решения вычислительных задач. Вообще говоря, алгоритм – это процесс, состоящий из ряда четко определенных шагов. Например, чтобы вычислить сумму 18 и 19, нужно решить, как быть с тем, что 8 плюс 9 больше 10, хотя в разных культурах этот факт обрабатывается поразному. Даже в такой простой задаче мы ожидаем, что используемые шаги позволят быстро дать правильный ответ. Есть много задач, более трудных, чем сложение, и для их эффективного и правильного решения нужно проектировать шаги вычислений более тщательно.

При разработке ГИС и их приложений алгоритмы важны чуть ли не в каждой детали. Например, щелкая мышью по карте, мы ожидаем получить от компьютера быстрый ответ, содержащий информацию о точке или области, в которой находится курсор мыши. В этой процедуре, встречающейся практически в любом приложении ГИС, участвует несколько алгоритмов, гарантирующих получение приемлемого ответа. Все начинается с поиска объекта (точки, прямой, многоугольника или пикселя), на который пришелся щелчок. Эффективный алгоритм поиска позволяет быстро сузить интересующую нас область. Попытка решить задачу в лоб означала бы проверку каждого объекта, что в случае большого набора данных сделало бы поиск недопустимо долгим. Для решения этой проблемы придумано много алгоритмов индексирования и опроса пространственных данных. В процессе поиска мы должны проверять, соответствует ли объект данных точке на экране. В случае многоугольников требуется решить, находится ли точка внутри многоугольника, для чего нужен специальный алгоритм, который быстро отвечает «да» или «нет» на этот вопрос. Обычно геопространственные данные поступают из многих источников, и принято приводить их к единой системе координат, чтобы разные наборы данных можно было обрабатывать единообразно. Еще одно типичное следствие наличия нескольких источников данных – организация слоев, позволяющих удобно использовать разнородную информацию.

Исследовать алгоритм можно с разных точек зрения. Очевидно, что алгоритм должен решать задачу правильно. Правильность некоторых алгоритмов доказать легко. Например, ниже в этой главе мы познакомимся с двумя алгоритмами поиска, правильность которых не вызывает сомнений. Другие

---

[1] Само слово «алгоритм» происходит от латинского *algorismus*, которое, в свою очередь, происходит от имени персидского математика, астронома и географа *Аль-Хорезми*, внесшего большой вклад в алгебру и географическую картину мира.

алгоритмы не столь очевидны, для доказательства их правильности необходим формальный анализ. Второе свойство алгоритмов – эффективность, или время работы. Конечно, мы всегда хотим, чтобы алгоритм работал быстро, но существуют теоретические пределы быстродействию алгоритма, зависящие от задачи. Мы будем обсуждать некоторые проблемы такого рода в конце книги при рассмотрении пространственной оптимизации. Помимо правильности и времени работы, при рассмотрении алгоритмов зачастую следует принимать во внимание организацию данных и конкретную реализацию.

1.1. Вычислительные аспекты алгоритмов

Пусть имеется неупорядоченный список n точек. Мы хотим найти в этом списке конкретную точку. Сколько времени для этого понадобится? Это разумный вопрос. Но на фактическое *время* влияет множество разных факторов: выбор языка программирования, квалификация программиста, платформа, количество и быстродействие процессоров и т. д. Более полезный способ оценки времени – количество *шагов*, необходимых для завершения работы, и анализ общей стоимости алгоритма в терминах количества шагов. Конечно, стоимость каждого шага – величина переменная, зависящая от того, что понимается под шагом. Но все это равно более надежный способ описания времени вычислений, потому что можно выделить ряд шагов – например, простые арифметические операции, вычисление логических выражений, доступ к памяти компьютера для выборки данных, присваивание значения переменной, – стоимость которых постоянна. Если мы сможем посчитать, сколько шагов необходимо для выполнения некоторой процедуры, то будем иметь неплохое представление о том, сколько времени эта процедура займет, что особенно полезно при сравнении алгоритмов.

Но вернемся к нашему списку точек. Если точки могут храниться в произвольном порядке, то лучшее, что можно сделать, – перебирать их одну за другой, пока не найдем искомую точку или не дойдем до конца списка. Предположим, что список называется `points`, и мы хотим узнать, включает ли он точку `p0`. Для выполнения поиска можно воспользоваться следующим простым алгоритмом (листинг 1.1).

Листинг 1.1 ❖ Линейный поиск точки `p0` в списке

```
1 для каждой точки p в points:
2     если p совпадает с p0:
3         вернуть p и остановиться
```

Алгоритм в листинге 1.1 называется линейным поиском; мы просто перебираем все точки для поиска нужной информации. Сколько шагов придется сделать? Первая строка – заголовок цикла, который будет выполнен n раз, если искомая точка окажется последней в списке. Стоимость одной итерации цикла постоянна, поскольку список хранится в памяти компьютера, а основные операции в данном случае – доступ к информации по фиксированному адресу в памяти и переход к следующему адресу. Предположим, что стои-

мость этой операции равна c_1 и что мы выполняем ее в цикле не более n раз. Вторая строка – логическое сравнение двух точек. Она тоже выполняется не более n раз, потому что находится внутри цикла. Пусть стоимость сравнения тоже постоянна и равна c_2. В строке 3 просто возвращается значение найденной точки; стоимость этой операции равна c, и выполняется она только один раз. В лучшем случае мы найдем искомую точку уже на первой итерации цикла, поэтому полная стоимость будет равна $c_1 + c_2 + c$, или, упрощая, константе $b + c$. Но в худшем случае нам придется просмотреть все элементы от начала до конца, поэтому полная стоимость будет равна $c_1n + c_2n + c$, или $bn + c$, где b и c – константы, а n – длина списка (размер задачи). В среднем, если список представляет собой случайный набор точек и мы ищем в нем случайную точку, ожидаемая стоимость составит $c_1n/2 + c_2n/2 + c$, или $b'n + c$, и мы знаем, что $b' < b$, т. е. стоимость ниже, чем в худшем случае.

Насколько нам интересны фактические значения b, b' и c в этом анализе? Как они влияют на общую стоимость вычислений? Не слишком сильно, поскольку это константы. Но если складывать их много раз, то влияние будет существенным, а количество сложений в общем случае зависит от размера задачи n. Когда n достигает определенного уровня, влияние самих констант оказывается минимальным, а *рост* полной стоимости вычислений определяется величиной n.

Стоимость некоторых алгоритмов пропорциональна n^2, что сильно отличает их от алгоритмов со стоимостью, пропорциональной n. Например, в листинге 1.2 приведена простая процедура для вычисления наименьшего расстояния между парами точек в списке, содержащем n точек. Здесь первый цикл (строка 2) выполняется n раз, и стоимость каждой его итерации равна t_1, а второй цикл (строка 3) – n^2 раз с той же стоимостью итерации. Код сравнения (строка 4) выполняется n^2 раз, стоимость одного сравнения обозначим t_2. Очевидно, что вычисление расстояния (строка 5) обходится дороже остальных, более простых операций, но все равно постоянно, поскольку входные данные фиксированы, а для выполнения вычислений нужно конечное и не слишком большое количество шагов. Будем считать, что стоимость вычисления одного расстояния равна константе t_3. Поскольку расстояние между точкой и ей самой не вычисляется, то код вычисления расстояния будет выполняться $n^2 - n$ раз, как и сравнение в строке 6 (стоимость которого обозначим t_4). Стоимость присваивания в строке 7 постоянна и равна t_5, а выполняться оно может не более $n^2 - n$ в худшем случае, когда каждое следующее расстояние меньше предыдущего. Последняя строка выполняется только один раз, ее стоимость обозначим c. В итоге полное время работы этого алгоритма равно $t_1n + t_1n^2 + t_2n^2 + t_3(n^2 - n) + t_4(n^2 - n) + t_5(n^2 - n) + c$, или, упрощая, $an^2 + bn + c$. Теперь понятно, что время работы данного алгоритма определяется величиной n^2.

Листинг 1.2 ❖ Линейный поиск для нахождения наименьшего расстояния между парами точек в списке

```
1 пусть mindist - очень большое число
2 для каждой точки p1 в points:
3     для каждой точки p2 в points:
```

```
4         если p1 не совпадает с p2:
5             пусть d – расстояние между p1 и p2
6             если d < mindist:
7                 mindist = d
8 вернуть mindist и остановиться
```

В обоих рассмотренных примерах величина n определяет полную стоимость вычислений, и мы говорим, что стоимость алгоритма линейного поиска имеет порядок n, а стоимость алгоритма нахождения минимального расстояния – порядок n^2. В случае линейного поиска мы также знаем, что при увеличении n полная стоимость ограничена сверху величиной bn. А что можно сказать насчет нижней границы? Мы знаем, что в лучшем случае время выполнения постоянно, т. е. имеет порядок n^0, но в общем случае это не так. Если мы можем найти верхнюю, но не нижнюю границу времени выполнения, то пользуемся нотацией O для обозначения порядка. В нашем примере порядок равен $O(n)$ как в среднем, так и в худшем случае (поскольку не константы определяют полную стоимость). Говорят также, что время работы, или временная сложность алгоритма линейного поиска, равно $O(n)$. Поскольку нотация O относится к верхней границе, т. е. к худшему случаю, имеется в виду временная сложность также в худшем случае.

Существуют алгоритмы, для которых время работы не ограничено сверху. Но нам известна нижняя граница, и тогда используется нотация Ω. Когда говорят, что время работы равно $\Omega(n)$, это значит, что временная сложность алгоритма по порядку величины никак не меньше n, хотя верхняя граница неизвестна. Бывают также алгоритмы, для которых известны и верхняя, и нижняя границы времени работы, тогда используется нотация Θ. Например, время работы $\Theta(n^2)$ означает, что алгоритм в любом случае занимает время порядка n^2. Так обстоит дело для алгоритма нахождения наименьшего расстояния, поскольку всегда выполняется n^2 итераций цикла вне зависимости от результата сравнения в строке 6. Будет точнее сказать, что временная сложность равна $\Theta(n^2)$, а не $O(n^2)$, потому что нам известно, что ее нижняя граница также имеет порядок n^2.

Теперь организуем точки, хранящиеся в списке, в виде дерева, как показано на рис. 1.1. Это двоичное дерево, потому что из каждого узла, начиная с корня, может исходить не более двух ветвей. Здесь в корне дерева хранится точка (6, 7), она показана сверху. Все точки, для которых координата X меньше или равна, чем у точки в корне, хранятся в узлах, находящихся слева от корня, а точки, для которых координата X больше, чем у точки в корне, – в узлах справа от корня. На втором уровне мы видим точки (4, 6) и (9, 4). Для каждой из них слева находятся точки с меньшей или равной координатой Y, а справа – с большей. Спускаясь вниз по дереву, мы попеременно используем координаты X и Y, пока не найдем нужную нам точку или не дойдем до конца ветви (листового узла).

Чтобы воспользоваться этой древовидной структурой для поиска точки, мы начинаем с корня (поиск в дереве всегда начинается с корня) и спускаемся вниз, решая на каждом уровне, по какой ветви идти. Например, чтобы найти точку (1, 7), мы пойдем из корня по левой ветви, потому что координата X искомой точки равна 1, т. е. меньше, чем в корне. Затем мы пойдем из узла

(4, 6) по правой ветви, потому что координата *Y* (7) меньше, чем хранится в текущем узле. Мы пришли в узел (3, 7) на третьем уровне дереве, и из него пойдем по левой ветви, потому что координата *X* искомой точки меньше, чем в узле. Дойдя до узла (2, 9), мы пойдем налево, потому что координата *Y* искомой точки меньше, чем в узле. Резюмируем: зная дерево, мы можем написать алгоритм поиска в нем; он показан в листинге 1.3.

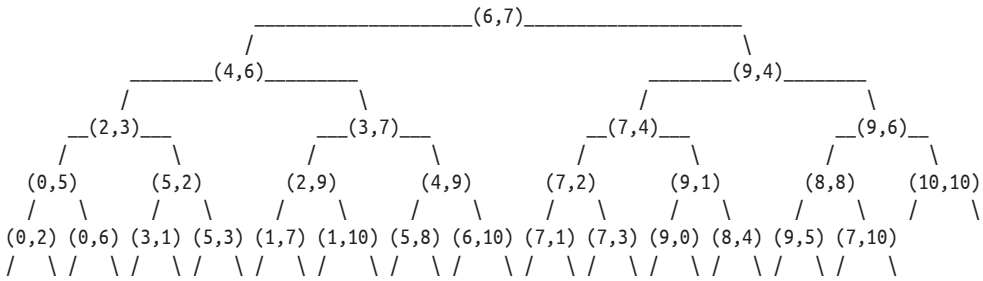

Рис. 1.1 ❖ Дерево, в котором хранится 29 случайно выбранных точек. Каждый узел помечен координатами *X* и *Y*, изменяющимися в диапазоне от 0 до 10

Листинг 1.3 ❖ Двоичный поиск точки `p0` в дереве

```
 1 пусть t – корень дерева
 2 пока t не пусто:
 3   пусть p – точка в узле t
 4   если p совпадает с p0:
 5       вернуть p и остановиться
 6   если t расположен на том уже уровне, что p0:
 7       coordp, coordp0 = координаты X точек p и p0
 8   иначе:
 9       coordp, coordp0 = координаты Y точек p и p0
10   если coordp0 <= coordp:
11       t = левая ветвь t
12   иначе:
13       t = правая ветвь t
```

Эта процедура называется двоичным поиском по дереву. Вспоминая наше обсуждение времени работы, легко видеть, что время работы этого алгоритма определяется количеством итераций цикла `пока` (строка 2), которое зависит от высоты дерева, т. е. числом ребер на пути от корня до самого удаленного листового узла. Высота показанного выше дерева равна 4, оно может содержать до 31 узла (в нашем примере их всего 29). Вообще, в двоичном дереве высоты H можно сохранить до $2^0 + 2^1 + 2^2 + \ldots + 2^H = 2^{H+1} - 1$ элементов данных. Иными словами, если имеется n точек, размещенных во всех узлах идеально сбалансированного двоичного дерева, в котором все листовые узлы находятся на одном и том же уровне, то $2^{H+1} - 1 = n$ и, следовательно, $H = \log_2(n + 1) - 1$. Если задано такое дерево, то время работы алгоритма имеет порядок $\log_2(n + 1)$. Поскольку по порядку величины – это то же самое, что $\log_2 n$, мы говорим, что временная сложность равна $O(\log_2 n)$. Для сбалансированного, но не идеально сбалансированного двоичного дерева, в котором раз-

ность высот листовых узлов не больше 1, временная сложность по-прежнему равна $O(\log_2 n)$, т. е. самый дальний листовой узел имеет высоту H. Но если сбалансированность гарантировать невозможно, то все становится хуже. На рис. 1.2 приведен пример несбалансированного дерева, в котором хранятся те же точки, но высота равна 8. Итак, мы знаем, что алгоритм двоичного поиска эффективнее, чем линейный поиск, потому что $O(\log_2 n) < O(n)$, но фактическое время работы зависит от того, как построено дерево, и может быть больше $O(\log_2 n)$, если дерево не сбалансировано.

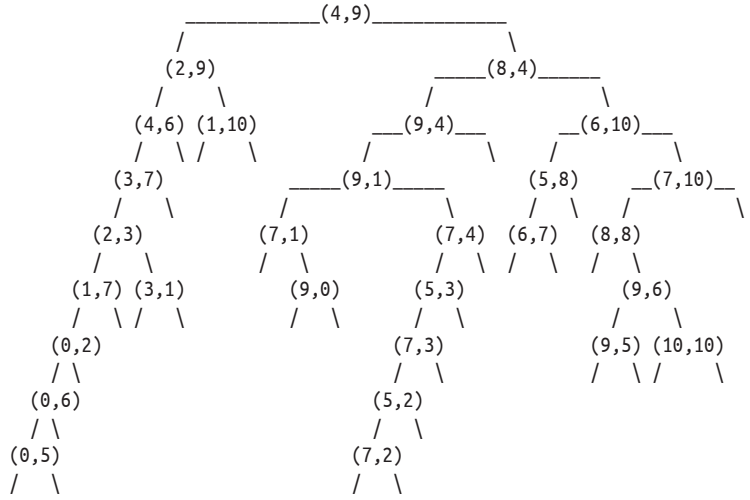

Рис. 1.2 ❖ Несбалансированное дерево, в котором хранится 29 случайных точек

Поскольку хранение точек в сбалансированном двоичном дереве может заметно повысить эффективность поиска, то не поможет ли оно заодно и сократить время вычисления кратчайшего расстояния между двумя точками? Временная сложность рассмотренного выше решения путем полного перебора равна $\Theta(n^2)$, она быстро возрастает с увеличением n. Было бы разумно поискать более эффективный способ нахождения наименьшего из попарных расстояний между точками в списке. Такое решение действительно существует, и ключ к нему – использование древовидных структур. Мы вернемся к этой теме во второй части книги, когда будем обсуждать различные древовидные структуры более подробно. В этой книге мы не будем уделять много внимания теоретическому анализу сложности алгоритмов, а вместо этого ограничимся эмпирическим анализом, выполняя алгоритмы с разными наборами данных.

Разговор о поиске, в особенности о двоичном поиске по дереву, логически подводит нас к вопросу о структуре данных: как хранить и организовывать данные, чтобы повысить эффективность алгоритма? Структура дерева – хороший пример того, как данные, первоначально представленные в виде списка, можно реорганизовать для увеличения производительности поиска. Многие структуры данных зависят от конкретной задачи. Некоторые струк-

туры довольно сложны, но увеличение объема занятой данными памяти часто компенсируется уменьшением времени работы.

1.2. Кодирование

Алгоритмы можно описывать разными способами. В этой главе мы описывали алгоритмы линейного и двоичного поиска словесно. В теоретической работе достаточно формального описания, в котором детально специфицированы все шаги, но исполнимость не является обязательным условием. Такое описание называется псевдокодом, потому что это не настоящий компьютерный код, хотя и очень близко к нему. В данной книге мы избрали более практичный путь – описывать алгоритм на реальном языке программирования, в качестве какового решили взять Python.

У составления компьютерной программы (т. е. кодирования) для описания алгоритма имеется важное преимущество: любой алгоритм можно сразу же выполнить. Поэтому все относящееся к работе алгоритма представлено в самой книге в виде простого текста. Код становится частью текста, и, следовательно, открывается возможность экспериментировать с ним, модифицировать и улучшать. Однако при таком подходе информации может оказаться слишком много, особенно если язык программирования заставляет писать вспомогательный код, без которого программа не работает. Например, во многих языках требуется ставить специальные символы или скобки в конце строки, иначе компилятор считает программу синтаксически некорректной. Добавление этих символов затрудняет чтение и мешает сосредоточиться на основном содержании текста. Мы выбрали в этой книге Python в основном за его простой синтаксис, а также за изобилие популярных, эффективных и хорошо сопровождаемых модулей. Все приведенные в данной книге программы были протестированы для версии Python 2.7, которая была стабильной и широко распространенной на момент написания книги. В большинстве программ используются только базовые средства Python, поэтому велики шансы, что они будут совместимы и с последующими версиями Python.

1.3. Как использовать эту книгу

Основной текст книги разделен на три большие части. Идея в том, чтобы сначала обсудить самые фундаментальные аспекты данных – геометрические, а затем переходить к более сложным темам: индексированию пространственных данных, а также пространственному анализу и моделированию. В конце каждой главы имеется раздел «Примечания», в котором описываются основные работы, связанные с рассмотренным материалом. Мы также включили три приложения, стремясь помочь читателям разобраться в языке программирования Python и в структуре включенных в книгу программ.

Часть I посвящена описанию местоположения, а конкретно координатам, которые необходимы для представления геопространственной информа-

ции. В главе 2 мы рассмотрим несколько алгоритмов для вычисления разных видов расстояния, например расстояния между точками или от точки до прямой. Мы также обсудим вычисление центра тяжести многоугольника и широко распространенный алгоритм, который эффективно отвечает на вопрос, находится ли точка внутри многоугольника. Последняя тема главы 2 – преобразование систем координат, включая картографические проекции. В главе 3 рассматривается традиционная операция ГИС – наложение. Хотя эта тема «старая», фактическое вычисление результата наложения двух многоугольников может быть весьма трудоемким, пусть и не особенно сложным. Многие вопросы, обсуждаемые в этой части книги, связаны с дисциплиной, называемой вычислительной геометрией. Но нас будут интересовать в основном вещи, касающиеся ГИС.

Основной темой части II является идея индексирования пространственных данных. У пространственной информации есть свои особенности. Хотя основной подход к индексированию – разделяй и властвуй – остается в силе и для пространственных данных, из-за наличия двух (а в некоторых случаях и большего числа) измерений приходится проектировать более специализированные алгоритмы. В главе 4 мы введем основные понятия, связанные с индексированием, и сосредоточимся на разработке структуры дерева. Глава 5 посвящена *k*D-деревьям, которые обычно применяются для индексирования данных о точках. В главе 6 рассматривается популярный метод квадродеревьев, или деревьев квадрантов для индексирования точек и растровых данных. В главе 7 в обсуждение включаются прямые и многоугольники.

Часть III посвящена главной теме приложений ГИС: пространственному анализу и моделированию. Сначала в главе 8 мы рассмотрим интерполяцию координат точек и сравним два стандартных метода: обратных взвешенных расстояний и кригинг. Мы также включили алгоритм имитации данных под названием *смещение средней точки*, взятый из литературы по фрактальной геометрии. Глава 9 посвящена анализу пространственных паттернов, в т. ч. вычислению индекса *I* Морана. В главу 10 включены алгоритмы анализа сетей, в частности вычисления кратчайшего пути. Две главы мы посвятили пространственной оптимизации: в главе 11 рассматриваются точные методы, а в главе 12 – некоторые эвристические методы.

Мы также включили три приложения, содержащих технические детали, относящиеся к кодированию. Сразу хотим сказать, что хотя в основном это книга об алгоритмах, кодирование тоже занимает в ней важное место. Поэтому мы добавили краткое введение в язык Python. Далее следует столь же краткое введение в мощную библиотеку GDAL/OGR, точнее в ее интерфейс с Python, и в Python-библиотеку пространственного анализа PySAL. Наша цель – помочь читателю быстро начать работу с этими библиотеками и получить представление о том, как «реальные» наборы данных связаны с обсуждаемыми в книге вопросами (и, конечно, кодом).

Большинство программ, встречающихся в книге, хранятся в отдельных файлах, так что их можно использовать в других программах. В таком случае в заголовке листинга указывается имя файла. Для организации программ мы пользуемся каталогами. В последнем приложении перечислены все встречающиеся Python-программы и наборы данных.

Каждую главу можно было бы легко развернуть в отдельную книгу, где соответствующая тема рассматривается более полно. Таким образом, эту книгу можно рассматривать как обзор тематики алгоритмов ГИС. Лучший способ охватить представленные в книге вопросы во всей широте – кодирование. Сейчас ведется работа над коллективной страницей на Github[1], где читатели смогут делиться мыслями о реализации как включенных в книгу алгоритмов, так и новых алгоритмов, которые в книгу не вошли. Теоретический вывод не находится в фокусе нашего внимания, чего нельзя сказать об эмпирическом анализе. Мы включили в основной текст и в упражнения много экспериментов. Но это не значит, что вы не можете экспериментировать самостоятельно, особенно когда речь идет об инновационных направлениях. Книгу можно будет считать успешной, только если она достигнет двух целей: во-первых, на основе усвоенных алгоритмов и кода читатели смогут разрабатывать собственные инструменты, отвечающие особенностям наборов данных и требованиям приложений, а во-вторых, написание кода станет привычкой при работе с геопространственными данными.

[1] https://github.com/gisalgs.

Часть I

ГЕОМЕТРИЧЕСКИЕ АЛГОРИТМЫ

Глава 2

Базовые геометрические операции

> Представьте себе огромный лист бумаги, на котором Отрезки прямых, Треугольники, Квадраты, Пятиугольники, Шестиугольники и другие фигуры, вместо того чтобы неподвижно оставаться на своих местах, свободно перемещаются по всем направлениям вдоль поверхности, не будучи, однако, в силах ни приподняться над ней, ни опуститься под нее, подобно теням (только твердым и со светящимися краями), и вы получите весьма точное представление о моей стране и моих соотечественниках.
>
> *Эдвин Э. Эбботт «Флатландия. Роман о четвертом измерении»*

Эта главе посвящена основным алгоритмам ГИС, относящимся к геометрическим операциям. Сначала мы познакомимся с вычислением расстояния между двумя точками, а затем перейдем к объектам в пространстве большего числа измерений, в т. ч. алгоритму «точка внутри многоугольника», вычислению расстояния от точки до прямой, нахождению центроида и площади многоугольника. Мы также рассмотрим две картографические проекции и обсудим, как преобразовать геопространственные данные из одной системы координат в другую.

2.1. Точка

Прежде чем начать обсуждение, определим структуру данных для представления точки, которой будем пользоваться на протяжении всей книги. Это класс Python под названием `Point` (листинг 2.1). Нас интересует только двумерный случай, поэтому мы храним в классе лишь координаты *X* и *Y* точки. Мы переопределяем несколько встроенных методов Python, чтобы предоставить удобные средства. Метод `__getitem__` позволяет ссылаться на координаты *X* и *Y*, указывая соответственно индексы 0 и 1. Метод `__len__` возвращает количество измерений (в данном случае два). Мы также считаем

две точки одинаковыми, если их координаты X и Y совпадают (метод `__eq__`), и различными в противном случае (метод `__ne__`). Кроме того, нам нужен метод сравнения точек – точка в левом нижнем квадранте всегда считается «меньше» точки в правом верхнем квадранте. Точнее, если даны две точки p_1 и p_2, то мы считаем, что $p_1 < p_2$, если координата X точки p_1 меньше. Если координаты X двух точек совпадают, то меньшей считается точка, у которой меньше координата Y. Это правило закодировано в переопределенных операторах сравнения `__lt__` (меньше), `__gt__` (больше), `__le__` (меньше или равно) и `__ge__` (больше или равно). Мы также выводим координаты точки при ее печати (методы `__str__` и `__repr__`). Хотя в этой главе описанный способ упорядочения и сравнения не используется, он очень пригодится в последующих главах. Метод `distance` нужен для вычисления евклидова расстояния между двумя точками.

Листинг 2.1 ❖ Структура данных для представления точки (point.py)

```
from math import sqrt
class Point():
    """Класс для представления точки в декартовой системе координат."""
    def __init__(self, x=None, y=None):
        self.x, self.y = x, y
    def __getitem__(self, i):
        if i==0: return self.x
        if i==1: return self.y
        return None
    def __len__(self):
        return 2
    def __eq__(self, other):
        if isinstance(other, Point):
            return self.x==other.x and self.y==other.y
        return NotImplemented
    def __ne__(self, other):
        result = self.__eq__(other)
        if result is NotImplemented:
            return result
        return not result
    def __lt__(self, other):
        if isinstance(other, Point):
            if self.x<other.x:
                return True
            elif self.x==other.x and self.y<other.y:
                return True
            return False
        return NotImplemented
    def __gt__(self, other):
        if isinstance(other, Point):
            if self.x>other.x:
                return True
            elif self.x==other.x and self.y>other.y:
                return True
            return False
```

```
36          return NotImplemented
37      def __ge__(self, other):
38          if isinstance(other, Point):
39              if self > other or self == other:
40                  return True
41              else:
42                  return False
43              return False
44          return NotImplemented
45      def __le__(self, other):
46          if isinstance(other, Point):
47              if self < other or self == other:
48                  return True
49              else:
50                  return False
51              return False
52          return NotImplemented
53      def __str__(self):
54          if type(self.x) is int and type(self.y) is int:
55              return "({0},{1})".format(self.x,self.y)
56          else:
57              return "({0:.1f}, {1:.1f})".format(self.x,self.y)
58      def __repr__(self):
59          if type(self.x) is int and type(self.y) is int:
60              return "({0},{1})".format(self.x,self.y)
61          else:
62              return "({0:.1f}, {1:.1f})".format(self.x,self.y)
63      def distance(self, other):
64          return sqrt((self.x-other.x)**2 + (self.y-other.y)**2)
```

Класс `Point` легко расширить для более гибкого представления точек. Например, мы предполагаем, что X и Y – декартовы координаты (в этом предположении определен метод `distance`), но это ограничение можно ослабить, добавив в класс еще один член `CS`, который задает вид системы координат; тогда расстояние нужно будет вычислять соответственно. Можно создать подкласс, который наследует классу `Point` и определяет точки в пространстве более высокой размерности. Можно добавить в класс время и другие атрибуты.

2.2. Расстояние между двумя точками

Расстояние между двумя точками можно вычислить разными способами, зависящими от предметной области и системы координат, в которой представлены точки. Самый распространенный способ – измерение евклидова расстояния по прямой между двумя точками на декартовой плоскости по формуле

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2},$$

где x_1 и x_2 – абсциссы (X) двух точек, а y_1 и y_2 – их ординаты (Y). Метод `distance` класса `Point` возвращает именно евклидово расстояние между двумя точками (листинг 2.1).

В некоторых ситуациях евклидово расстояние – не самый подходящий способ измерения расстояния между двумя точками, особенно когда речь о городе, где перемещаться из одной точки в другую нужно только по улицам (рис. 2.1). В таком случае используется манхэттенское расстояние:

$$d_1 = |x_1 - x_2| + |y_1 - y_2|.$$

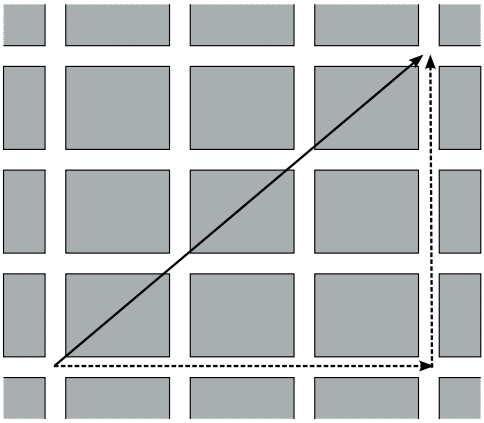

Рис. 2.1 ❖ Евклидово расстояние (сплошная линия) и манхэттенское расстояние (пунктирная линия). Серые прямоугольники представляют городские здания, между которыми проходят дороги

Если координаты точек измерены на поверхности сферы, то приведенные выше формулы не дают правильного расстояния между точками. В этом случае кратчайшее расстояние измеряется по дуге большой окружности, проходящей через две точки. Точнее, если положение точки определяется широтой, или углом с плоскостью экватора (φ), и долготой, или углом между меридианом, проходящим через точку, и нулевым меридианом (λ), то длина дуги большой окружности α (угол) между двумя точками определяется по формуле

$$\cos\alpha = \sin\varphi_1 \sin\varphi_2 + \cos\varphi_1 \cos\varphi_2 \cos\lambda,$$

где $d\lambda = |\lambda_1 - \lambda_2|$ – абсолютная величина разности между долготами двух точек. Но при вычислениях мы пользуемся следующей формулой гаверсинуса, чтобы обойти трудности при работе с отрицательными значениями:

$$a = \sin^2 \frac{d\varphi}{2} + \cos\varphi_1 \cos\varphi_2 \sin^2 \frac{d\lambda}{2},$$

$$c = 2\arcsin\min(1, \sqrt{a}),$$

где функция min возвращает меньшее из двух значений, чтобы избежать последствий ошибок округления, из-за которых a может оказаться больше 1; тогда расстояние можно вычислить по формуле

$$d = cR,$$

где R – радиус сферической Земли (3959 миль, или 6371 км). Важно помнить, что перед применением этой формулы широту и долготу нужно преобразовать в радианы.

Теперь напишем на Python программу для вычисления расстояния между двумя точками на сфере (листинг 2.2). Здесь широта и долгота выражены в градусах, и мы преобразуем их в радианы (строки 13–16). Пользуясь этим кодом, мы можем вычислить расстояние между Колумбусом, штат Огайо (40° с.ш., 83° з.д.) и Пекином (39.91° с.ш., 116.56° в.д.), оно равно 6780 миль (10 911 км).

Листинг 2.2 ❖ Вычисление расстояния по дуге большой окружности (spherical_distance.py)

```
 1 import math
 2 def spdist(lat1, lon1, lat2, lon2):
 3     """
 4     Вычисляет расстояние по дуге большой окружности, зная
 5     широту и долготу двух точек.
 6     Вход
 7       lat1, lon1: широта и долгота первой точки в градусах
 8       lat2, lon2: широта и долгота второй точки в градусах
 9     Выход
10       d: расстояние по дуге большой окружности
11     """
12     D = 3959                          # радиус Земли в милях
13     phi1 = math.radians(lat1)
14     lambda1 = math.radians(lon1)
15     phi2 = math.radians(lat2)
16     lambda2 = math.radians(lon2)
17     dlambda = lambda2 - lambda1
18     dphi = phi2 - phi1
19     sinlat = math.sin(dphi/2.0)
20     sinlong = math.sin(dlambda/2.0)
21     alpha=(sinlat*sinlat) + math.cos(phi1) * \
22         math.cos(phi2) * (sinlong*sinlong)
23     c=2 * math.asin(min(1, math.sqrt(alpha)))
24     d=D*c
25     return d
26
27 if __name__ == "__main__":
28     lat1, lon1 = 40, -83              # Колумбус, штат Огайо
29     lat2, lon2 = 39.91, 116.56        # Пекин
30     print spdist(lat1, lon1, lat2, lon2)
```

2.3. Расстояние от точки до прямой

Пусть $ax + by + c = 0$ – уравнение прямой на плоскости, где a, b, c – константы. Расстояние от точки (x_0, y_0) на этой плоскости до прямой равно

$$\frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}.$$

Это можно доказать, вычислив длину перпендикуляра, опущенного из точки (x_0, y_0) на прямую $ax + by + c = 0$. Обозначим (x_1, y_1) точку пересечения прямой и перпендикуляра. Мы знаем, что тангенс угла наклона перпендикулярной прямой равен b/a. Отсюда

$$\frac{y_1 - y_0}{x_1 - x_0} = \frac{b}{a},$$

что дает

$$a(y_1 - y_0) - b(x_1 - x_0) = 0.$$

Возведем обе части равенства в квадрат и выполним простое преобразование:

$$a^2(y_1 - y_0)^2 + b^2(x_1 - x_0)^2 = 2ab(x_1 - x_0)(y_1 - y_0).$$

Прибавим $a^2(x_1 - x_0)^2 + b^2(y_1 - y_0)^2$ к обеим частям. Тогда левую часть можно записать в виде

$$(a^2 + b^2)[(y_1 - y_0)^2 + (x_1 - x_0)^2],$$

а правую часть – в виде

$$[a(x_1 - x_0) + b(y_1 - y_0)]^2 = [ax_1 + by_1 - ax_0 - by_0]^2.$$

Поскольку точка (x_1, y_1) принадлежит также исходной прямой, $ax_1 + by_1 + c = 0$. Следовательно,

$$(a^2 + b^2)[(y_1 - y_0)^2 + (x_1 - x_0)^2] = [ax_1 + by_1 + c]^2,$$

Член $[(y_1 - y_0)^2 + (x_1 - x_0)^2]$ – это квадрат расстояния между (x_0, y_0) и прямой $ax + by + c = 0$. Таким образом, искомое расстояние равно

$$\sqrt{(y_1 - y_0)^2 + (x_1 - x_0)^2} = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}.$$

В большинстве приложений ГИС легко получить информацию об отрезке, заданном двумя своими концевыми точками. Поэтому необходимо вычислить значения a, b, c в приведенных выше уравнениях по двум точкам. Обозначим их (x_1, y_1) и (x_2, y_2) и положим $dx = x_1 - x_2$, $dy = y_1 - y_2$. Тогда уравнение прямой можно записать в виде с угловым коэффициентом:

$$y = \frac{dy}{dx}x + n,$$

где n – постоянная, которую мы хотим вычислить. Подставляя в это уравнение координаты первой концевой точки, имеем

$$y_1 = \frac{dy}{dx}x_1 + n,$$

откуда

$$n = y_1 - \frac{dy}{dx}x_1.$$

Таким образом, мы получили общий вид уравнения прямой, проходящей через две точки:

$$y = \frac{dy}{dx}x + y_1 - \frac{dy}{dx}x_1.$$

Умножим обе части этого равенства на dx и изменим порядок членов:

$xdy - ydx + y_1dx - x_1dy = 0$.

Отсюда $a = dy$, $b = -dx$, $c = y_1dx - x_1dy$. Зная эти параметры, легко вычислить расстояние от точки до отрезка прямой (листинг 2.3).

Листинг 2.3 ❖ Python-программа для вычисления расстояния между точкой и прямой (point2line.py)

```
import math
from point import *

def point2line(p, p1, p2):
    """
    Вычисляет расстояние между точкой и прямой.
    Вход
      p: точка
      p1 и p2: две точки, определяющие прямую
    Выход
      d: расстояние от точки p до прямой p1p2
    """
    x0 = float(p.x)
    y0 = float(p.y)
    x1 = float(p1.x)
    y1 = float(p1.y)
    x2 = float(p2.x)
    y2 = float(p2.y)
    dx = x1-x2
    dy = y1-y2
    a = dy
    b = -dx
    c = y1*dx - x1*dy
    if a==0 and b==0:                # точки p1 и p2 совпадают
```

```
25          d = math.sqrt((x1-x0)*(x1-x0) + (y1-y0)*(y1-y0))
26      else:
27          d = abs(a*x0+b*y0+c)/math.sqrt(a*a+b*b)
28      return d
29
30 if __name__ == "__main__":
31      p, p1, p2 = Point(10,0), Point(0,100), Point(0,1)
32      print point2line(p, p1, p2)
33      p, p1, p2 = Point(0,10), Point(1000,0.001), Point(-100,0)
34      print point2line(p, p1, p2)
35      p, p1, p2 = Point(0,0), Point(0,10), Point(10,0)
36      print point2line(p, p1, p2)
37      p, p1, p2 = Point(0,0), Point(10,10), Point(10,10)
38      print point2line(p, p1, p2)
```

Мы продемонстрировали применение этого кода на нескольких тестовых примерах (строки 31–38). Печатается следующая информация:

```
10.0
9.9999090909
7.07106781187
14.1421356237
```

2.4. Центроид и площадь многоугольника

Будем представлять многоугольник P с n вершинами в виде последовательности (x_1, y_1), (x_2, y_2), ..., (x_n, y_n), в которую добавим еще одну точку (x_{n+1}, y_{n+1}), чтобы многоугольник был замкнутым. Если в многоугольнике нет дырок, а его граница не имеет самопересечений, то координаты центроида определяются по формулам:

$$x = \frac{1}{6A}\sum_{i=1}^{n}(x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i);$$

$$y = \frac{1}{6A}\sum_{i=1}^{n}(y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i),$$

где A – площадь многоугольника. Если многоугольник выпуклый, то центроид находится внутри него. Но для невыпуклых многоугольников центроид может оказаться снаружи. Хотя понятие центроида тесно связано с центром масс многоугольника, точку, находящуюся снаружи, трудно назвать центром, что вынуждает нас поместить центроид внутрь многоугольника.

Чтобы понять смысл этой формулы, возьмем для примера многоугольник на рис. 2.2 с вершинами (a, b, c, d, e, f, g, a); точка a указана дважды, чтобы замкнуть многоугольник. На каждом отрезке границы можно построить трапецию, площадь которой вычисляется по формуле $(x_2 - x_1)(y_2 + y_1)/2$, где индексы 1 и 2 служат для обозначения двух соседних точек в последовательности. Так, площадь трапеции $abb'a'$ равна $(x_b - x_a)(y_b + y_a)/2$, а площадь трапеции $fgg'f'$ равна $(x_g - x_f)(y_g + y_f)/2$ – эта величина отрицательна. Очевидно, что вычисленные по данной формуле площади трапеций под отрезками ab,

bc, *cd* и *de* (обратите внимание на порядок точек) положительны, а площади трапеций под отрезками *ef*, *fg* и *ga* отрицательны. Поэтому сумма этих величин действительно равна площади многоугольника, поскольку площадь фигуры, ограниченной ломаной $efgaa'e'$, вычитается из площади фигуры, ограниченной ломаной $abcdee'a'$. Таким образом, площадь многоугольника вычисляется по формуле

$$A = \frac{1}{2}\sum_{i=1}^{n}(x_{i+1} - x_i)(y_{i+1} + y_i).$$

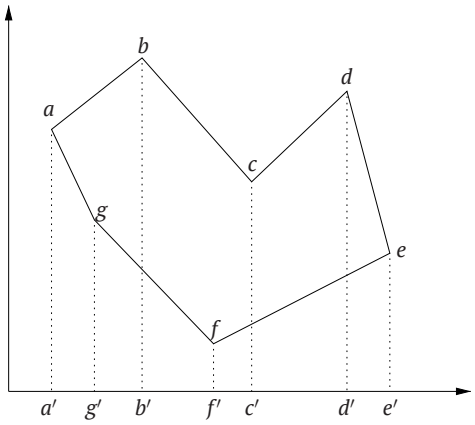

Рис. 2.2 ❖ Использование трапеций для вычисления площади многоугольника. Пунктирные линии проецируют вершины многоугольника на горизонтальную ось

Эта формула основана на разложении многоугольника в последовательность трапеций. Ее можно переписать в таком виде:

$$A = \frac{1}{2}\sum_{i=1}^{n}(x_{i+1}y_i - x_i y_{i+1}).$$

Хотя вторая формула дает такой же результат, как и первая, в прошлом, когда вычисления производились вручную, она была удобнее. В зависимости от порядка точек в последовательности формула площади многоугольника может давать отрицательное значение. Поэтому для получения истинной площади многоугольника нужно взять абсолютную величину.

Python-программа в листинге 2.4 возвращает центроид и площадь многоугольника, вычисленные по приведенным выше формулам. Мы также протестировали программу на примере многоугольника, заданного списком точек.

Листинг 2.4 ❖ Python-программа для вычисления площади и центроида многоугольника (centroid.py)

```
1 from point import *
2
3 def centroid(pgon):
4     """
```

```
    Вычисляет центроид и площадь многоугольника.
    Вход
      pgon: список объектов Point
    Выход
      A: площадь многоугольника
      C: центроид многоугольника
    """
    numvert = len(pgon)
    A = 0
    xmean = 0
    ymean = 0
    for i in range(numvert-1):
        ai = pgon[i].x*pgon[i+1].y - pgon[i+1].x*pgon[i].y
        A += ai
        xmean += (pgon[i+1].x+pgon[i].x) * ai
        ymean += (pgon[i+1].y+pgon[i].y) * ai
    A = A/2.0
    C = Point(xmean / (6*A), ymean / (6*A))
    return A, C

# TEST
if __name__ == "__main__":
    points = [ [0,10], [5,0], [10,10], [15,0], [20,10],
               [25,0], [30,20], [40,20], [45,0], [50,50],
               [40,40], [30,50], [25,20], [20,50], [15,10],
               [10,50], [8, 8], [4,50], [0,10] ]
    polygon = [ Point(p[0], p[1]) for p in points ]
    print centroid(polygon)
```

2.5. Определение положения точки относительно прямой

Результат вычисления площади многоугольника может оказаться положительным или отрицательным в зависимости от порядка точек. Это очень интересная особенность, позволяющая использовать знак площади для определения того, по какую сторону от прямой лежит точка. Для этого нужно вычислить площадь треугольника, образованного данной точкой и двумя точками на прямой. Пусть a, b, c – три вершины треугольника. По выведенной выше формуле получаем площадь треугольника в виде

$$A(abc) = \frac{1}{2}(x_b y_a - x_a y_b + x_c y_b - x_b y_c + x_a y_c - x_c y_a).$$

Включив в скобки член $x_b y_b - x_b y_b$, мы можем переписать эту формулу в более простом виде:

$$\begin{aligned}\text{sideplr}(abc) &= 2A(abc) \\ &= (x_a - x_b)(y_c - y_b) - (x_c - x_b)(y_a - y_b).\end{aligned}$$

Если точка *a* лежит слева от вектора, направленного из *b* в *c*, то вычисление по этой формуле дает отрицательное значение. Например, в конфигурациях точек *a*, *b*, *c* на рис. 2.3A–C площадь треугольника *abc* (обратите внимание на порядок точек) вычисляется так, что площади трапеций, расположенных ниже треугольника (например, трапеции под отрезком *cb* на рис. 2.3A), всегда вычитаются из полной площади (например, суммы площадей трапеций под отрезками *ba* и *ac* на рис. 2.3A). Это приводит к отрицательному значению. С другой стороны, если *a* лежит справа от вектора *bc* (см. рис. 2.3D–F), то значение будет положительно. Код, определяющий положение точки относительно прямой, приведен в листинге 2.5.

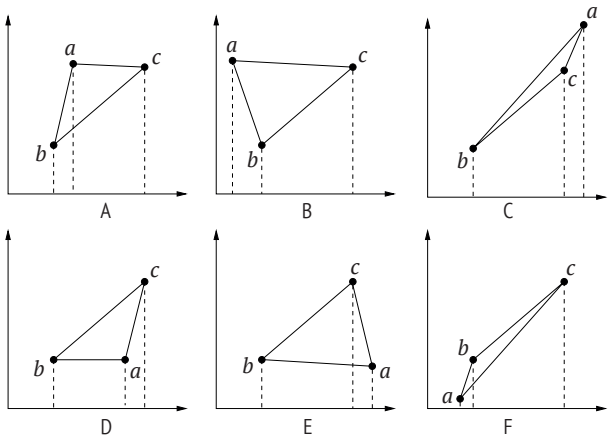

Рис. 2.3 ❖ Положение точки *a* относительно прямой *bc*

Листинг 2.5 ❖ Определение взаимного расположения точки и прямой (sideplr.py)

```
from point import *

def sideplr(p, p1, p2):
    """
    Вычисляет, по какую сторону от вектора p1p2 находится точка p.
    Вход
      p: точка
      p1, p2: начало и конец вектора
    Выход
      -1: p слева p1p2
      0: p на прямой p1p2
      1: p справа p1p2
    """
    return int((p.x-p1.x)*(p2.y-p1.y)-(p2.x-p1.x)*(p.y-p1.y))

if __name__ == "__main__":
    p=Point(1,1)
    p1=Point(0,0)
    p2=Point(1,0)
    print "Положение точки %s относительно прямой %s->%s: %d"%(
```

```
21          p, p1, p2, sideplr(p, p1, p2))
22      print "Положение точки %s относительно прямой %s->%s: %d"%(
23          p, p2, p1, sideplr(p, p2, p1))
24      p = Point(0.5, 0)
25      print "Положение точки %s относительно прямой %s->%s: %d"%(
26          p, p1, p2, sideplr(p, p1, p2))
27      print "Положение точки %s относительно прямой %s->%s: %d"%(
28          p, p2, p1, sideplr(p, p2, p1))
```

В результате выполнения тестов будет напечатано:

```
Положение точки (1,1) относительно прямой (0,0)->(1,0): -1
Положение точки (1,1) относительно прямой (1,0)->(0,0): 1
Положение точки (0.5, 0.0) относительно прямой (0,0)->(1,0): 0
Положение точки (0.5, 0.0) относительно прямой (1,0)->(0,0): 0
```

Если все три точки лежат на одной прямой, то программа вернет 0. Таким образом, мы легко можем узнать, лежит ли точка на прямой или вне нее. Эта техника окажется очень полезной в следующем разделе, когда мы будем обсуждать пересечение отрезков.

2.6. Пересечение отрезков прямых

Сначала рассмотрим пересечение двух прямых: L_1, проходящей через точки (x_1, y_1) и (x_2, y_2), и L_2, проходящей через точки (x_3, y_3) и (x_4, y_4). Их угловые коэффициенты равны соответственно

$$\alpha_1 = \frac{y_2 - y_1}{x_2 - x_1} \quad \text{и} \quad \alpha_2 = \frac{y_4 - y_3}{x_4 - x_3}.$$

Вычислить их точку пересечения просто. Сначала запишем уравнения прямых в виде

$$y = \alpha_1(x - x_1) + y_1$$

и

$$y = \alpha_2(x - x_3) + y_3.$$

Отсюда легко вычислить абсциссу точки пересечения:

$$x = \frac{\alpha_1 x_1 - \alpha_2 x_3 + y_3 - y_1}{\alpha_1 - \alpha_2},$$

– и ее ординату:

$$y = \alpha_1(x - x_1) + y_1.$$

Понятно, что нужно рассмотреть несколько случаев. Если угловые коэффициенты двух прямых равны, то прямые не пересекаются. А что, если одна или обе прямые вертикальны, т. е. угловой коэффициент бесконечен? В этом

случае, если $x_2 - x_1$ и $x_4 - x_3$ одновременно равны нулю, мы имеем две параллельные прямые. Если же только одна из этих величин равна нулю, то абсцисса точки пересечения равна координате x вертикальной прямой. Например, если $x_1 = x_2$, то точка пересечения имеет координаты $x = x_1$ и $y = \alpha_2(x_1 - x_3) + y_3$.

Но если рассматривать только отрезки прямых, то все описанное выше может оказаться ненужным, потому что отрезки могут и не пересекаться. Чтобы проверить, так ли это, нужно рассмотреть концы отрезков. Если оба конца одного отрезка лежат по одну сторону от другого, то отрезки не пересекаются. Для проверки расположения точки относительно отрезка мы можем воспользоваться алгоритмом `sideplr` из предыдущего раздела.

Прежде чем приводить формальный алгоритм вычисления точки пересечения двух отрезков, определим структуру данных для эффективного хранения информации об отрезке (листинг 2.6). Нам понадобится номер стороны (`e`) и координаты концов. Мы сохраняем исходное значение левого конца отрезка (`lp0` в строке 23), потому что в дальнейшем левый конец будет изменяться при вычислении нескольких точек пересечения, сдвигающихся слева направо. Наконец, в атрибуте `status` мы будем хранить признак, показывающий, является ли левый конец отрезка исходным левым концом (по умолчанию) или внутренней точкой, появившейся в результате пересечения с другими отрезками из-за наложения многоугольников, а в атрибуте `c` – свойства отрезка. Часть описанных возможностей понадобится нам в следующем разделе.

Для представления концов отрезков в классе `Segment` используется ранее разработанный класс `Point` из листинга 2.1. Мы также определяем операции сравнения отрезков, переопределив встроенные функции Python, в т. ч. `__eq__` (равно) и `__lt__` (меньше). Мы считаем, что отрезок s_1 меньше отрезка s_2, если s_1 целиком расположен ниже s_2. Мы также включили функцию для проверки того, является ли точка одним из концов отрезка.

Листинг 2.6 ❖ Структура данных для представления отрезков (linesegment.py)

```
1 from point import *
2 from sideplr import *
3
4 ## Два статуса левого конца
5 ENDPOINT = 0 ## изначальный левый конец
6 INTERIOR = 1 ## внутренняя точка отрезка
7
8 class Segment:
9     """
10    Класс для представления отрезков прямых.
11    """
12    def __init__(self, e, p0, p1, c=None):
13        """
14        Конструктор класса Segment.
15        Вход
16          e: ИД отрезка, целое число
17          p0, p1: концы отрезка, объекты Point
18        """
```

```
        if p0>=p1:
            p0,p1 = p1,p0             # p0 всегда левый конец
            self.edge = e             # ИД, задается для всех сторон
            self.lp = p0              # левый конец
            self.lp0 = p0             # первоначальный левый конец
            self.rp = p1              # правый конец
            self.status = ENDPOINT    # статус отрезка
            self.c = c                # c: ИД свойства
    def __eq__(self, other):
        if isinstance(other, Segment):
            return (self.lp==other.lp and self.rp==other.rp)\
                or (self.lp==other.rp and self.rp==other.lp)
        return NotImplemented
    def __ne__(self, other):
        result = self.__eq__(other)
        if result is NotImplemented:
            return result
        return not result
    def __lt__(self, other):
        if isinstance(other, Segment):
            if self.lp and other.lp:
                lr = sideplr(self.lp, other.lp, other.rp)
                if lr == 0:
                    lrr = sideplr(self.rp, other.lp, other.rp)
                    if other.lp.x < other.rp.x:
                        return lrr > 0
                    else:
                        return lrr < 0
                else:
                    if other.lp.x > other.rp.x:
                        return lr < 0
                    else:
                        return lr > 0
        return NotImplemented
    def __gt__(self, other):
        result = self.__lt__(other)
        if result is NotImplemented:
            return result
        return not result
    def __repr__(self):
        return "{0}".format(self.edge)
    def contains(self, p):
        """
        Возвращает True, если точка p является концом отрезка
        """
        if self.lp == p:
            return -1
        elif self.rp == p:
            return 1
        else:
            return 0
```

В листинге 2.7 приведен пример вычисления точки пересечения отрезков. Сначала мы с помощью функции `test_intersect` определяем, пересекаются ли отрезки (строка 60), для чего нужно проверить расположение концов одного отрезка относительно другого, вызвав функцию `sideplr` (например, строка 41). Если оба конца одного отрезка лежат по одну сторону от другого, то отрезки не пересекаются. В противном случае в функции `getIntersectionPoint` используются уравнения, выведенные в начале этого раздела, чтобы вычислить точку пересечения. Эта функция предполагает, что два входных отрезка пересекаются (так что факт пересечения нужно проверить заранее). Тестовые отрезки (строки 57 и 58) пересекаются в точке (1.5, 2.5).

Листинг 2.7 ❖ Вычисление точки пересечения отрезков (intersection.py)

```
 1 from linesegment import *
 2 from sideplr import *
 3
 4 def getIntersectionPoint(s1, s2):
 5     """
 6     Вычисляет точку перечения отрезков s1 и s2.
 7     Предполагается, что s1 and s2 пересекаются.
 8     Факт пересечения необходимо проверить до вызова этой функции.
 9     """
10     x1 = float(s1.lp0.x)
11     y1 = float(s1.lp0.y)
12     x2 = float(s1.rp.x)
13     y2 = float(s1.rp.y)
14     x3 = float(s2.lp0.x)
15     y3 = float(s2.lp0.y)
16     x4 = float(s2.rp.x)
17     y4 = float(s2.rp.y)
18     if s1.lp < s2.lp:
19         x1,x2,y1,y2,x3,x4,y3,y4=x3,x4,y3,y4,x1,x2,y1,y2
20     if x1 != x2:
21         alpha1 = (y2-y1)/(x2-x1)
22     if x3 != x4:
23         alpha2 = (y4-y3)/(x4-x3)
24     if x1 == x2:                    # s1 расположен вертикально
25         y = alpha2*(x1-x3)+y3
26         return Point([x1, y])
27     if x3==x4:                      # s2 расположен вертикально
28         y = alpha1*(x3-x1)+y1
29         return Point([x3, y])
30     if alpha1 == alpha2:            # прямые параллельны
31       return None
32     # нужно вычислить
33     x = (alpha1*x1-alpha2*x3+y3-y1)/(alpha1-alpha2)
34     y = alpha1*(x-x1) + y1
35     return Point(x, y)
36
37 def test_intersect(s1, s2):
38     if s1==None or s2==None:
39         return False
40     # проверка: концы s2 лежат по одну сторону от s1
```

```
    lsign = sideplr(s2.lp0, s1.lp0, s1.rp)
    rsign = sideplr(s2.rp, s1.lp0, s1.rp)
    if lsign*rsign > 0:
        return False
    # проверка: концы s1 лежат по одну сторону от s2
    lsign = sideplr(s1.lp0, s2.lp0, s2.rp)
    rsign = sideplr(s1.rp, s2.lp0, s2.rp)
    if lsign*rsign > 0:
        return False
    return True

if __name__ == "__main__":
    p1 = Point(1, 2)
    p2 = Point(3, 4)
    p3 = Point(2, 1)
    p4 = Point(1, 4)
    s1 = Segment(0, p1, p2)
    s2 = Segment(1, p3, p4)
    s3 = Segment(2, p1, p2)
    if test_intersect(s1, s2):
        print getIntersectionPoint(s1, s2)
        print s1==s2
        print s1==s3
```

2.7. Операция «точка внутри многоугольника»

Выяснение того, находится ли точка внутри многоугольника, – одна из важнейших операций при использовании ГИС. Например, щелкая мышью в точке на цифровой карте мира, мы ожидаем быстро получить информацию о регионе, в котором эта точка находится. В случае простого (несамопересекающегося) выпуклого многоугольника точка находится внутри, если она лежит по одну и ту же сторону от всех сторон многоугольника. Этот подход выглядит вычислительно сложным, поскольку требуется много операций умножения, а кроме того, он не работает для невыпуклых многоугольников (рис. 2.4).

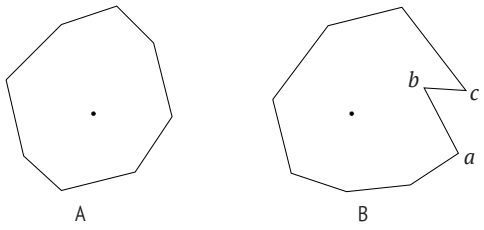

Рис. 2.4 ❖ Определение расположения точки относительно многоугольника путем выяснения того, по какую сторону точка расположена относительно его сторон. Предполагается, что задана последовательность вершин (в порядке обхода по часовой стрелке или против часовой стрелки): (A) точка находится по одну сторону от всех отрезков; (B) точка находится с противоположных сторон отрезков *ab* и *bc*

2.7.1. Алгоритм чет-нечет

Алгоритм чет-нечет – популярный метод, известный также под названиями алгоритм бросания лучей (ray-casting) или алгоритм числа пересечений. В этом алгоритме проверка пересечения выполняется путем проведения полупрямой (бросания луча) из точки. Если луч пересекает границу многоугольника в четном числе точек, то точка находится внутри многоугольника, в противном случае снаружи. Для удобства будем рисовать луч горизонтально (рис. 2.5). В целом процесс бесхитростный, но мы рассчитываем избежать фактического вычисления точек пересечения, потому что эта операция занимает много времени, особенно если производится многократно.

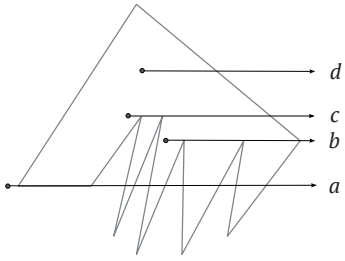

Рис. 2.5 ❖ Алгоритм «точка внутри многоугольника». Начальная точка луча *a* находится вне многоугольника, а начальные точки лучей *b*, *c* и *d* – внутри многоугольника

На рис. 2.6 изображены различные случаи, показывающие, когда вычислять точку пересечения необходимо, а когда без этого можно обойтись. Мы видим горизонтальный луч, начинающийся в точке *A*, и несколько отрезков, обозначенных буквами от *a* до *e* и *b′*. Мы пытаемся проверить, пересекается ли отрезок с лучом, не вычисляя саму точку пересечения. Отрезки *a*, *d* и *e* не пересекаются с лучом, потому что абсциссы обоих концов лежат по одну сторону от начала луча (*a*), или ординаты обоих концов лежат по одну сторону луча (*d* и *e*). Отрезок *c* мы засчитываем как пересечение, потому что он действительно пересекает луч, но вычислять точку пересечения не нужно, т. к. мы заведомо знаем, что этот отрезок пересекает луч: ординаты концов *c* лежат по разные стороны от луча, а их абсциссы расположены справа от точки *A*. Для отрезков *b* и *b′* невозможно сходу сказать, пересекают они луч или нет, так что придется вычислять точку пересечения. По счастью, нам нужно вычислить только абсциссу точки пересечения, чтобы решить, пересекается отрезок с лучом или нет. В случае отрезка *b* абсцисса точки пересечения находится слева от точки *A*, поэтому отрезок не пересекает луч. В случае отрезка *b′* точка пересечения находится справа от точки *A*, поэтому отрезок пересекает луч.

Имеется важное исключение, часто встречающееся в реальных приложениях. Что, если луч проходит через оба конца отрезка? А если луч проходит через один конец отрезка? В таких случаях мы можем автоматически прибавить к соответствующим концам очень маленькое значение, так что тео-

ретически они окажутся «над» лучом. В результате мы можем быть уверены, что либо отрезок пересекает луч, либо нет – третьего не дано. В реализации этого алгоритма мы считаем, что конец отрезка расположен «выше» луча, если его ордината «больше или равна» ординате луча. В противном случае считается, что отрезок расположен ниже луча. Например, считается, что луч *b* на рис. 2.5 пересекает границу многоугольника пять раз (а не один), а луч *a* пересекает границу восемь раз.

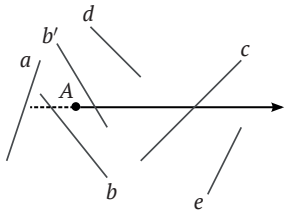

Рис. 2.6 ❖ Различные случаи при вычислении точки пересечения

Функция `pip_cross` в листинге 2.8 возвращает признак, показывающий, находится ли точка внутри многоугольника, используя алгоритм чет-нечет. В переменных `p1` и `p2` мы храним соседние вершины многоугольника. Переменные `yflag1` и `yflag2` показывают положение ординат двух вершин относительно луча (выше/на или ниже), а переменные `xflag1` и `xflag2` – положение абсцисс (слева/на или справа). В строке 25 проверяется, лежат ли точки `p1` и `p2` по разные стороны от луча. Если да, то далее в строке 28 проверяется, лежат ли они по разные стороны от начальной точки луча. Если обе точки находятся справа от начальной, то мы засчитываем одно пересечение. В противном случае придется вычислить абсциссу точки пересечения (строка 34) и решить, пересекается ли отрезок с лучом (строка 35).

Листинг 2.8 ❖ Алгоритм чет-нечет проверки расположения точки относительно многоугольника (point_in_polygon.py)

```
 1 import math
 2 from point import *
 3
 4 def pip_cross(point, pgon):
 5     """
 6     Определяет, находится ли точка внутри многоугольника. В основе кода
 7     лежит программа на C из книги Haines "Graphics Gems IV" (1994).
 8     Вход
 9       pgon: список вершин многоугольника
10       point: точка
11     Выход
12       Возвращает булево значение True или False и сколько раз луч
13       пересекает границу многоугольника
14     """
15     numvert = len(pgon)
16     tx=point.x
```

```
    ty=point.y
    p1 = pgon[numvert-1]
    p2 = pgon[0]
    yflag1 = (p1.y >= ty)              # p1 на одном уровне с point или выше нее
    crossing = 0
    inside_flag = 0
    for j in range(numvert-1):
        yflag2 = (p2.y >= ty)          # p2 на одном уровне с point или выше нее
        if yflag1 != yflag2:           # по разные стороны от луча
            xflag1 = (p1.x >= tx)      # слева от p1
            xflag2 = (p2.x >= tx)      # слева от p2
            if xflag1 == xflag2:       # обе точки справа
                if xflag1:
                    crossing += 1
                    inside_flag = not inside_flag
            else:
                m = p2.x - float((p2.y-ty))*\
                    (p1.x-p2.x)/(p1.y-p2.y)
                if m >= tx:
                    crossing += 1
                    inside_flag = not inside_flag
        yflag1 = yflag2
        p1 = p2
        p2 = pgon[j+1]
    return inside_flag, crossing

if __name__ == "__main__":
    points = [ [0,10], [5,0], [10,10], [15,0], [20,10],
               [25,0], [30,20], [40,20], [45,0], [50,50],
               [40,40], [30,50], [25,20], [20,50], [15,10],
               [10,50], [8, 8], [4,50], [0,10] ]
    ppgon = [Point(p[0], p[1]) for p in points ]
    inout = lambda pip: "ВНУТРИ" if pip is True else "СНАРУЖИ"
    point = Point(10, 30)
    print "Точка %s находится %s"%(
        point, inout(pip_cross(point, ppgon)[0]))
    point = Point(10, 20)
    print "Точка %s находится %s"%(
        point, inout(pip_cross(point, ppgon)[0]))
    point = Point(20, 40)
    print "Точка %s находится %s"%(
        point, inout(pip_cross(point, ppgon)[0]))
    point = Point(5, 40)
    print "Точка %s находится %s"%(
        point, inout(pip_cross(point, ppgon)[0]))
```

Для тестирования программы мы взяли многоугольник и точки, показанные на рис. 2.7. Программа напечатала следующие результаты:

```
Точка (10,30) находится ВНУТРИ
Точка (10,20) находится ВНУТРИ
Точка (20,40) находится ВНУТРИ
Точка (5,40) находится СНАРУЖИ
```

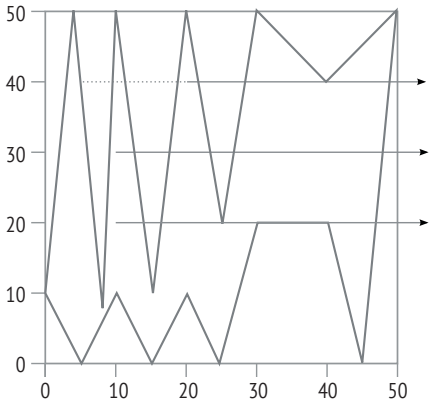

Рис. 2.7 ❖ Тесты для алгоритма чет-нечет

2.7.2. Алгоритм на основе числа оборотов

В обоих случаях, показанных на рис. 2.4, мы видим, что стороны многоугольника совершают оборот вокруг внутренней точки. Если же точка находится вне многоугольника, то стороны не оборачиваются вокруг нее. Алгоритм на основе числа оборотов пользуется этим свойством, чтобы определить, находится точка внутри или снаружи многоугольника. В общем случае, если даны два n-мерных вектора $X = (x_1, x_2, \ldots, x_n)$ и $Y = (y_1, y_2, \ldots, y_n)$, то имеем $X \cdot Y = |X||Y|\cos\theta$, где θ – угол между векторами, $X \cdot Y$ – скалярное произведение двух векторов, равное $\sum_{i=1}^{n} x_i y_i$, а $|X|$ и $|Y|$ – нормы (или длины) X и Y соответственно. Если задан отрезок между точками $v_i = (x_i, y_i)$ и $v_{i+1} = (x_{i+1}, y_{i+1})$ и точка $p = (x, y)$, то угол между p и отрезком обозначается θ_i и вычисляется по формуле

$$\theta_i = \arccos\left[\frac{\overrightarrow{v_i p} \cdot \overrightarrow{v_{i+1} p}}{|\overrightarrow{v_i p}||\overrightarrow{v_{i+1} p}|}\right]$$
$$= \arccos\left[\frac{(x - x_i)(x - x_{i+1}) + (y - y_i)(y - y_{i+1})}{\sqrt{(x - x_i)^2 + (y - y_i)^2}\sqrt{(x - x_{i+1})^2 + (y - y_{i+1})^2}}\right],$$

где $\overrightarrow{v_i p}$ – вектор из точки v_i в точку p, а $|\overrightarrow{v_i p}|$ – длина этого вектора. Предположим теперь, что многоугольник содержит n вершин и $v1 = v_n$, т. е. многоугольник замкнутый.

Тогда сумма углов между точкой и всеми n сторонами многоугольника, называемая индексом точки относительно кривой или числом оборотов, равна

$$wn = \frac{1}{2\pi}\sum_{i=1}^{n-1}\theta_i.$$

Точка находится внутри многоугольника, если ее индекс не равен нулю.

Описанный выше алгоритм на основе числа оборотов вычислительно неэффективен из-за использования тригонометрических функций. Эту проблему можно решить, оценивая число оборотов сторон многоугольника во-

круг точки по их направлениям. Если сторона поднимается и пересекает уровень *Y* точки, то число оборотов увеличивается на 1, а если опускается, то уменьшается на 1. Как показано на рис. 2.8, левые концы обоих горизонтальных отрезков будут считаться находящимися внутри многоугольника, потому что отрезок *a* пересекает одна восходящая сторона, а отрезок *b* – две восходящие стороны. По существу, этот подход оказывается таким же, как основанный на числе пересечений, и эффективность у них одинаковая. Код обоих вариантов метода на основе числа оборотов приведен в листинге 2.9; функция `pip_wn` реализует традиционный алгоритм, а `pip_wn1` – вариант с подсчетом числа пересечений. Тестовые данные, включенные в листинг, отражают ситуацию, сходную с рис. 2.8.

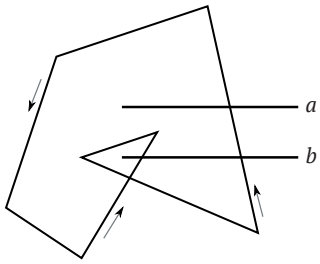

Рис. 2.8 ❖ Проверка принадлежности точки многоугольнику методом на основе числа оборотов. Стрелками показаны направления сторон многоугольника – против часовой стрелки

Листинг 2.9 ❖ Алгоритм на основе числа оборотов (point_in_polygon_winding.py)

```
import math
from point import *

def is_left(p, p1, p2):
    """
    Проверяет, лежит ли точка слева от отрезка, соединяющего
    p1 и p2
    Выход
       0 точка лежит на отрезке
      >0 p лежит слева от отрезка
      <0 p лежит справа от отрезка
    """
    return (p2.x-p1.x)*(p.y-p1.y) - (p.x-p1.x)*(p2.y-p1.y)

def pip_wn(pgon, point):
    """
    Определяет, находится ли точка внутри многоугольника, применяя алгоритм
    на основе числа оборотов с использованием тригонометрических функций.
    В основе коде лежит программа на C из книги Haines  "Graphics Gems IV"
   (1994).
    Вход
      pgon: список вершин многоугольника
      point: точка
```

```
    Выход
      Возвращает булево значение True или False и сколько раз луч
      пересекает границу многоугольника
    """
    if pgon[0] != pgon[-1]:
        pgon.append(pgon[0])
    n = len(pgon)
    xp = point.x
    yp = point.y
    wn = 0
    for i in range(n-1):
        xi = pgon[i].x
        yi = pgon[i].y
        xi1 = pgon[i+1].x
        yi1 = pgon[i+1].y
        thi = (xp-xi)*(xp-xi1) + (yp-yi)*(yp-yi1)
        norm = (math.sqrt((xp-xi)**2+(yp-yi)**2)
               * math.sqrt((xp-xi1)**2+(yp-yi1)**2))
        if thi != 0:
            thi = thi/norm
        thi = math.acos(thi)
        wn += thi
    wn /= 2*math.pi
    wn = int(wn)
    return wn is not 0, wn

def pip_wn1(pgon, point):
    """
    Определяет, находится ли точка внутри многоугольника, применяя алгоритм
    на основе числа оборотов без использования тригонометрических функций.
    В основе коде лежит программа на C из книги Haines  "Graphics Gems IV"
   (1994).
    Вход
      pgon: список вершин многоугольника
      point: точка
    Выход
      Возвращает булево значение True или False и сколько раз луч
      пересекает границу многоугольника
    """
    wn = 0
    n = len(pgon)
    for i in range(n-1):
        if pgon[i].y <= point.y:
            if pgon[i+1].y > point.y:
                if is_left(point, pgon[i], pgon[i+1])>0:
                    wn += 1
        else:
            if pgon[i+1].y <= point.y:
                if is_left(point, pgon[i], pgon[i+1])<0:
                    wn -= 1
    return wn is not 0, wn

```

```
76 if __name__ == "__main__":
77     pgon = [ [2,3], [7,4], [6,6], [4,2], [11,5],
78              [5,11], [2,3] ]
79     point = Point(6, 4)
80     ppgon = [Point(p[0], p[1]) for p in pgon ]
81     print pip_wn(ppgon, point)
82     print pip_wn1(ppgon, point)
```

В алгоритме на основе числа оборотов есть очевидная проблема: начальная (левая) точка луча *b* на рис. 2.8 будет считаться находящейся внутри многоугольника, потому что ее индекс больше нуля. Алгоритм чет-нечет сообщил бы, что эта точка находится снаружи, потому что луч пересекает границу многоугольника дважды. По поводу того, считать ли эту точку внутренней или внешней, было много споров. Но мы можем разрубить этот гордиев узел, потребовав, чтобы прямоугольники были простыми, т. е. чтобы никакая сторона не пересекала никакую другую сторону в точках, отличных от вершин. При таком предположении мы имеем на рисунке два многоугольника, меньший содержит точку, а больший – нет.

2.8. Картографические проекции

Под картографической проекцией понимается процесс преобразования из трехмерной сферической системы координат, в которой положение точки определяется широтой и долготой, в декартову систему координат на плоскости. Такое преобразование часто бывает необходимым, потому что представлять точки земной поверхности на плоском листе бумаге удобнее, чем на сфере. Существует много способов выполнить картографическую проекцию. Мы опишем только два метода, представляющих различные подходы к преобразованию.

Проекция Робинсона (рис. 2.9) – одна из самых распространенных в картографии, и не только. Она не сохраняет ни площади, ни локальные формы, но показывает мир, так чтобы были отчетливо видны приполярные области. Проекция Робинсона отличается от многих других проекций тем, что не основана всецело на математических формулах. Средний меридиан представлен прямой линией, длина которой составляет 0.5072 длины экватора. Параллели изображаются прямыми, расстояние между которыми одинаково

Рис. 2.9 ❖ Проекция Робинсона

в области между 38° с.ш. и 38° ю.ш., но уменьшается при приближении к полюсам. Полюсы в проекции Робинсона изображаются отрезками прямых длиной, равной 0.5322 длины экватора. На каждой параллели меридианы размещены через равные промежутки.

Точнее, начало координат в проекции Робинсона находится в точке пересечения экватора и среднего меридиана. Параметры в табл. 2.1 вычислены Робинсоном. Для каждой из 19 широт в диапазоне от 0° до 90° с шагом 5° Робинсон дает длины параллели и расстояние на карте от параллели до экватора. Поскольку меридианы на каждой параллели размещены с одинаковыми промежутками, легко вычислить, где на параллели находится отметка долготы относительно среднего меридиана, т. е. какова координата *X* этой точки. Однако точная длина известна лишь для отдельных параллелей (столбец 2 в табл. 2.1). И значения координаты *Y* известны только для этих параллелей (заметим, что параллели размещены с постоянным шагом в области между 38° с.ш. и 38° ю.ш.). Для любой другой широты находить длину (*A*) и расстояние до экватора (*B*) нужно путем интерполяции.

Таблица 2.1. Длины параллелей и их расстояния до экватора в проекции Робинсона

Широта (φ)	Длина (*A*)	Расстояние до экватора (*B*)
00	1.0000	0.0000
05	0.9986	0.0620
10	0.9954	0.1240
15	0.9900	0.1860
20	0.9822	0.2480
25	0.9730	0.3100
30	0.9600	0.3720
35	0.9427	0.4340
40	0.9216	0.4958
45	0.8962	0.5571
50	0.8679	0.6176
55	0.8350	0.6769
60	0.7986	0.7346
65	0.7597	0.7903
70	0.7186	0.8435
75	0.6732	0.8936
80	0.6213	0.9394
85	0.5722	0.9761
90	0.5322	1.0000

Однако интерполяция – это игра в угадайку, потому что Робинсон не оставил упоминаний о том, какой метод интерполяции был использован в оригинальной работе. Многие ученые предлагали различные способы аппроксимации проекции Робинсона. Один полезный метод интерполяции состоит в том, чтобы использовать полиномиальную функцию, так чтобы кривая проходила через все заданные в табл. 2.1 точки (широту и *A* или *B*).

Ниже описан алгоритм Невилла для нахождения такой функции. В общем случае предположим, что существует полином $p(x) = \sum_{i=0}^{n} a_i x_i$, который аппроксимирует $n + 1$ пару входных данных $\{(x_i, y_i)\}$, так что $p(x_i) = y_i$, $0 \leqslant i \leqslant n$. В табл. 2.2 показано, как алгоритм Невилла используется для вычисления значения в точке x, если известно пять пар (x_i, y_i) (первые два столбца). Обозначим $p_{i,j}(x)$ – результат интерполяции в точке x полиномом степени $j - i$ (степенью полинома называется максимальный показатель степени x в выражающей его формуле). На итерации 0 (столбец 3) каждое значение равно соответствующему значению y: мы имеем полином степени 0 $p_{i,i} = x^0 y_i = y_i$. Начиная с итерации 1 (столбец 4) каждое значение в столбце является результатом линейной интерполяции двух значений – над ним и под ним – в предыдущем столбце. Например:

$$p_{1,2}(x) = \frac{x_2 - x}{x_2 - x_1} p_{1,1}(x) - \frac{x_1 - x}{x_2 - x_1} p_{2,2}(x)$$
$$= \frac{x_2 - x}{x_2 - x_1} y_1 - \frac{x_1 - x}{x_2 - x_1} y_2.$$

Совершая итерации, мы вычисляем все полиномы первой степени (показаны в третьем столбце), а затем переходим к вычислению следующего столбца и так далее, пока не останется вычислить единственное значение, определяемое входными парами.

Таблица 2.2. Алгоритм Невилла

x_0	y_0	$p_{0,0}(x)$				
			$p_{0,1}(x)$			
x_1	y_1	$p_{1,1}(x)$		$p_{0,2}(x)$		
			$p_{1,2}(x)$		$p_{0,3}(x)$	
x_2	y_2	$p_{2,2}(x)$		$p_{1,3}(x)$		$p_{0,4}(x)$
			$p_{2,3}(x)$		$p_{1,4}(x)$	
x_3	y_3	$p_{3,3}(x)$		$p_{2,4}(x)$		
			$p_{3,4}(x)$			
x_4	y_4	$p_{4,4}(x)$				

Итоговую формулу алгоритма Невилла можно записать в виде определителя квадратной матрицы:

$$p_{i,j}(x) = \frac{1}{x_j - x_i} \begin{vmatrix} p_{i,j-1}(x) & x_i - x \\ p_{i+1,j}(x) & x_j - x \end{vmatrix},$$

или, эквивалентно,

$$p_{i,j}(x) = \frac{(x_j - x) p_{i,j-1}(x) + (x - x_i) p_{i+1,j}(x)}{x_j - x_i}.$$

В листинге 2.10 приведена реализация алгоритма Невилла. В этом коде нам не нужно заводить отдельный двумерный массив для хранения значений

p_{ij}. Если внимательно присмотреться к табл. 2.2, то мы увидим, что на k-й итерации нам нужно сохранить всего $n - k$ значений, так что списка размера n будет вполне достаточно (строка 12). Важно, что после обновления $p[i]$ новое значение не влияет на вычисления на той же итерации (строка 20).

Листинг 2.10 ❖ Алгоритм Невилла (neville.py)

```
 1 def neville(datax, datay, x):
 2     """
 3     Находим интерполированное значение с помощью алгоритма Невилла.
 4     Вход
 5       datax: входные x в списке длины n
 6       datay: входные y в списке длины n
 7       x: точка, в которой ищется интерполированное значение
 8     Выход
 9       p[0]: полином степени n
10     """
11     n = len(datax)
12     p = n*[0]
13     for k in range(n):
14         for i in range(n-k):
15             if k == 0:
16                 p[i] = datay[i]
17             else:
18                 p[i] = ((x-datax[i+k])*p[i]+ \
19                         (datax[i]-x)*p[i+1])/ \
20                         (datax[i]-datax[i+k])
21     return p[0]
```

Теперь мы готовы написать программу для вычисления проекции Робинсона (листинг 2.11). В самом начале мы инициализируем три списка значениями в табл. 2.1. В качестве среднего используется нулевой меридиан, мы еще вернемся к этому вопросу при обсуждении проекции Мольвейде. Функция `transform1` преобразует широту и долготу в систему координат проекции Робинсона. Поскольку все широты в таблице положительны, мы заменяем отрицательные широты (если они возникнут в процессе вычислений) абсолютными величинами (строка 62). В конце мы произведем обратное преобразование, если необходимо. В строке 65 проверяется, что входная широта не больше 90°. В строке 67 мы находим индекс наибольшей широты, которая меньше или равна входной широте. Это делает функция `find_le`, в которой используется эффективный метод двоичного поиска – это возможно, поскольку список `latitudes` отсортирован. Описание функции можно найти в официальной документации по Python[1].

Найдя в таблице индекс широты, непосредственно предшествующей входной широте (т. е. наибольшей из тех, что меньше нее), мы должны определить широты и значения A или B, которые будут использоваться для интерполяции. Использовать все имеющиеся в таблице значения просто, но рискованно. Дело в том, что степень интерполирующего полинома равна $n - 1$,

[1] https://docs.python.org/2/library/bisect.html.

где n – количество входных пар. Если мы будем использовать все значения в таблице, то получим полином степени 18. Скорее всего, это приведет к переподгонке, когда мы получим точные значения в самих заданных точках, но плохую аппроксимацию в интервалах между ними. Поэтому обычно мы берем для интерполяции два значения слева от входной широты и два значения справа. Например, если входная широта равна 12.5, то мы проведем интерполяцию по широтам 5, 10, 15 и 20, а функция `find_le` вернет 2 – индекс широты 10. Задача усложняется, когда входная широта находится близко к началу или к концу таблицы, потому что в этом случае нам не хватит индексов с одной стороны. Например, если входная широта равна 2.5, то нам придется использовать только одну широту слева, т. е. для интерполяции доступно лишь три широты: 0, 5 и 10.

Зная левый индекс, найденный функцией `find_le`, мы можем найти остальные с помощью функции `get_interpolation_range`. Эта функция позволяет задать желаемое количество индексов по обе стороны. По умолчанию с каждой стороны используется два индекса. В строке 68 мы получаем диапазон широт для интерполяции значения B, или координаты Y. Но чтобы интерполировать расстояние от параллели до экватора, мы должны учитывать, что параллели между 38° с.ш. и 38° ю.ш. расположены через равные промежутки. В таком случае мы можем задать только одну табличную широту с каждой стороны от входной. Из предыдущего обсуждения понятно, что в случае, когда имеется всего два известных значения, алгоритм Невилла выполняет линейную интерполяцию, а именно это нам и нужно для равноотстоящих параллелей. Это делается в строке 72.

Мы дважды вызываем алгоритм Невилла: для интерполяции расстояния от входной параллели до экватора (строка 70) и длины параллели (строка 74). В оставшейся части программы производится инициализация с учетом требований проекции Робинсона: длина среднего меридиана равна 0.5072 длины экватора (строка 75) и меридианы пересекают параллели в равноотстоящих точках (строки 76 и 77).

Листинг 2.11 ❖ Преобразование точек в проекцию Робинсона (transform1.py)

```
 1 import bisect
 2 from neville import *
 3 from numpy import fabs
 4
 5 latitudes=[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60,
 6            65, 70, 75, 80, 85, 90]
 7
 8 # длины параллелей на каждой широте в списке latitudes
 9 A=[1.0000, 0.9986, 0.9954, 0.9900, 0.9822, 0.9730, 0.9600,
10    0.9427, 0.9216, 0.8962, 0.8679, 0.8350, 0.7986, 0.7597,
11    0.7186, 0.6732, 0.6213, 0.5722, 0.5322]
12
13 # расстояние от каждой параллели до экватора
14 # эти значения нужно умножать на 0.5072
15 B=[0.0000, 0.0620, 0.1240, 0.1860, 0.2480, 0.3100, 0.3720,
16    0.4340, 0.4958, 0.5571, 0.6176, 0.6769, 0.7346, 0.7903,
```

```
   0.8435, 0.8936, 0.9394, 0.9761, 1.0000]

def find_le(a, x):
    """Находит самый правый элемент, меньший или равный x"""
    i = bisect.bisect_right(a, x)
    if i:
        return i-1
    raise ValueError

def get_interpolation_range(sidelen, n, i):
    """
    Находит диапазон индексов для интерполяции
    проекции Робинсона
    Вход
      sidelen: количество элементов по обе стороны от i,
               включая i в число левых
      n: общее число элементов
      i: индекс наибольшего элемента, меньшего значения
    Выход
      ileft: индекс левого значения (включая)
      iright: индекс правого значения (не включая)
    """
    if i<sidelen:
        ileft = max([0, i-sidelen+1])
    else:
        ileft = i-sidelen+1
        if i>=n-sidelen:
            iright = min(n, i+sidelen+1)
        else:
            iright = i+sidelen+1
    return ileft, iright

def transform1(lon, lat):
    """
    Возвращает результат преобразования lon и lat
    в координаты на проекции Робинсона.
    Вход
      lon: долгота
      lat: широта
    Выход
      x: координата x (начало координат в 0,0)
      y: координата y (начало координат в 0,0)
    """
    n = len(latitudes)
    south = False
    if lat<0:
        south = True
        lat = fabs(lat)
    if lat>90:
        return
    i = find_le(latitudes, lat)
    ileft, iright = get_interpolation_range(2, n, i)
    y = neville(latitudes[ileft:iright],
```

```
70                B[ileft:iright], lat)
71    if lat<=38:
72        ileft, iright = get_interpolation_range(1, n, i)
73    x = neville(latitudes[ileft:iright],
74                A[ileft:iright], lat)
75    y = 0.5072*y/2.0
76    dx = x/360.0
77    x = dx*lon
78    if south:
79        y = -1.0 * y
80    return x, y, ileft, i, iright
```

Для тестирования алгоритма проекции Робинсона мы будем использовать данные двух видов. Программа в листинге 2.12 поможет сгенерировать набор точек для этой цели. Сначала мы хотим нарисовать сетку меридианов и параллелей. Каждая из этих линий состоит из множества точек. Для хранения данной информации мы используем простой список, каждый элемент которого сам является списком из трех значений: идентификатор линии, а также долгота и широта точки. Точки, образующие каждую линию, нужно хранить последовательно, чтобы обеспечить гладкое изображение. Мы используем вложенный цикл, начинающийся в строке 6, для создания параллелей с шагом 10° и вложенный цикл, начинающийся в строке 11, для создания меридианов с таким же шагом. Для каждого меридиана мы берем по одной точке на каждой параллели, кроме параллелей севернее 80° с.ш. и южнее 80° ю.ш., где выборка более плотная (с шагом 1°), чтобы кривые получились гладкими.

Второй набор данных – это очертания материков в масштабе 1:110 млн[1], которые мы хотим нанести на сетку. В исходном наборе данных очертания хранятся в формате файла фигур (shapefile). С помощью модуля OGR мы читаем координаты и преобразуем их в простую структуру данных. Имя файла фигур передается на вход функции (строка 3), а затем используется в строке 24, где модуль OGR открывает файл. Дополнительные сведения о модуле OGR и смежных вопросах см. в приложении В. По ходу дела мы запоминаем количество строк в сетке параллелей и меридианов (`numgraticule`) и общее количество строк (`numline`), позже они понадобятся для рисования проекции карты.

Листинг 2.12 ❖ Подготовка данных для карты мира (worldmap.py)

```
 1 from osgeo import ogr
 2
 3 def prep_projection_data(fname):
 4     points=[]
 5     linenum = 0
 6     for lat in range(-90, 91, 10):
 7         for lon in range(-180, 181, 10):
 8             points.append([linenum, lon, lat])
 9     linenum += 1
10
11     for lon in range(-180, 181, 10):
```

[1] http://www.naturalearthdata.com/downloads/110m-physical-vectors/.

```
12          for lat in range(-90, -80, 1):
13              points.append([linenum, lon, lat])
14          for lat in range(-80, 80, 10):
15              points.append([linenum, lon, lat])
16          for lat in range(80, 91, 1):
17              points.append([linenum, lon, lat])
18          linenum += 1
19
20      numgraticule = linenum
21
22      driveName = "ESRI Shapefile"
23      driver = ogr.GetDriverByName(driveName)
24      vector = driver.Open(fname, 0)
25      layer = vector.GetLayer(0)
26
27      for i in range(layer.GetFeatureCount()):
28          f = layer.GetFeature(i)
29          geom = f.GetGeometryRef()
30          for i in range(geom.GetPointCount()):
31              p = geom.GetPoint(i)
32              points.append([linenum, p[0], p[1]])
33          linenum += 1
34
35      numline = max([p[0] for p in points]) + 1
36
37      return points, numgraticule, numline
```

Последний шаг – преобразовать данные о широте и долготе в координаты на проекции Робинсона. Это делает программа в листинге 2.13. Здесь мы пользуемся мощным модулем построения графиков и визуализации Matplotlib. Конечно, карту можно нарисовать в любом пакете ГИС, но мы остаемся привержены инструментам с открытым исходным кодом конкретно на языке Python. Дополнительные сведения об этом модуле можно найти в приложении А.

Сначала мы получаем данные в исходном формате (строка 7), затем преобразуем их в систему координат проекции Робинсона (строка 11), сохраняя структуру данных: ИД линии, координаты *X* и *Y* (строка 12). Приступая к отображению данных, мы сначала рисуем рамку, в которой будет размещаться карта (строка 24). В цикле `for` мы перебираем все строки данных (строка 14), задаем цвет (строка 15), так чтобы сетка параллелей и меридианов отображалась светло-серым, а очертания материков – темно-серым. Заметим, что в Python цвет можно задать разными способами, и здесь мы в одном случае используем словесное название «lightgrey», а в другом – задание цвета в формате HTML: яркости красной, зеленой и синей составляющей (каждая задается двумя 16-ричными цифрами) в строке, начинающейся знаком «#». В строках 19 и 20 мы получаем из списков точек соответственно координаты *X* и *Y*, которые затем используем для нанесения на график двумерной линии с помощью библиотеки Matplotlib (строка 21). Заметим, что в этот момент линии еще не отображаются, а только добавляются на график, который будет показан позже командой `show`.

Оставшаяся часть кода – процедуры, необходимые для правильного изображения карты. Особенно важна строка 23, в которой говорится, что оси должны масштабироваться, поскольку на проекции они должны быть разной длины. В строке 24 мы получаем текущие оси и вносим изменения, т. к. не хотим, чтобы оси были видны на карте. Результат всех этих действий показан на рис. 2.9, который сгенерирован закомментированной строкой 28, сохраняющей изображение в инкапсулированном формате Postscript, пригодном для публикации.

Листинг 2.13 ❖ Тестирование проекции Робинсона на данных карты мира (test_projection.py)

```
 1 from osgeo import ogr
 2 import matplotlib.pyplot as plt
 3 from transform1 import *
 4 from worldmap import *
 5
 6 fname = '../data/ne_110m_coastline.shp'
 7 pp, numgraticule, numline = prep_projection_data(fname)
 8
 9 points=[]
10 for p in pp:
11     p1 = transform1(p[1], p[2])
12     points.append([p[0], p1[0], p1[1]])
13
14 for i in range(numline):
15     if i<numgraticule:
16         col = 'lightgrey'
17     else:
18         col = '#5a5a5a'
19     ptsx = [p[1] for p in points if p[0]==i]
20     ptsy = [p[2] for p in points if p[0]==i]
21     plt.plot(ptsx, ptsy, color=col)
22
23 plt.axis('scaled')
23 frame = plt.gca()
25 frame.axes.get_xaxis().set_visible(False)
26 frame.axes.get_yaxis().set_visible(False)
27 frame.set_frame_on(False)
28 #plt.savefig(robinson.eps',bbox_inches='tight',pad_inches=0)
29 frame.set_frame_on(True)
30 plt.show()
```

Проекция Мольвейде – равновеликая проекция, сохраняющая площадь. Она описывается следующими уравнениями преобразования:

$$x = \frac{\sqrt{8}}{\pi} R(\lambda - \lambda_0)\cos\theta;$$

$$y = \sqrt{2}R\sin\theta,$$

где R – радиус глобуса, на который проецируется Земля (обычно принимается равным 1), λ – преобразуемая долгота, λ_0 – долгота среднего меридиана, а θ – угловой параметр, который должен удовлетворять условию

$$2\theta + \sin 2\theta = \pi \sin\varphi,$$

где φ – преобразуемая широта. Фактическое значение θ для каждой широты φ оценивается с помощью какого-нибудь итеративного алгоритма, например по методу Ньютона–Рафсона:

$$\Delta\theta' = -\frac{\theta' + \sin\theta' - \pi\sin\varphi}{1 + \cos\theta'}.$$

Чтобы воспользоваться этим итеративным методом, мы вначале присваиваем θ' значение φ и получаем первое значение $\Delta\theta'$. Затем в качестве нового значения θ' выбираем $\theta' + \Delta\theta'$ и повторяем процесс. Так продолжается, пока $\Delta\theta'$ не станет очень малым. В этот момент последнее значение θ' используется для вычисления θ по формуле

$$\theta = \theta'/2.$$

Этот алгоритм реализован в функции `opt_theta` в листинге 2.14; итерации прекращаются, когда $\Delta\theta'$ оказывается меньше 0.0000001. Отметим, что обычно углы выражаются в градусах, но в тригонометрических функциях Python в качестве единицы измерения применяются радианы, поэтому необходимо не забывать о преобразовании одного в другое. Для этой цели служат функции `degrees` и `radians` из библиотеки NumPy (см. введение в NumPy в приложении А). Например, при применении этой функции к широте –30° печатаются следующие результаты:

```
>>> opt_theta(-30, True)
theta = -30.0
delta = -16.8015452415
theta = -46.8015452415
delta = -0.849241867829
theta = -47.6507871093
delta = -0.00275402735705
theta = -47.6535411367
delta = -2.92287909643e-08
-0.41585559653479859
```

Листинг 2.14 ❖ Преобразование точек в проекцию Мольвейде (transform2.py)

```
from numpy import pi, cos, sin, radians, degrees, sqrt

def opt_theta(lat, verbose=False):
    """
    Находим оптимальное значение тета методом Ньютона-Рафсона.
    Вход
      lat: значение широты
      verbose: True, если нужна промежуточная печать
    Выход
```

```
10        theta
11      """
12      lat1 = radians(lat)
13      theta = lat1
14      while True:
15          dtheta = -(theta+sin(theta)-
16                     pi*sin(lat1))/(1.0+cos(theta))
17          if verbose:
18              print "theta =", degrees(theta)
19              print "delta =", degrees(dtheta)
20          if int(1000000*dtheta) == 0:
21              break
22          theta = theta+dtheta
23      return theta/2.0
24
25  def transform2(lon, lat, lon0=0, R=1.0):
26      """
27      Возвращает результат преобразования lon и lat
28      в проекцию Мольвейде.
29      Вход
30        lon: долгота
31        lat: широта
32        lon0: средний меридиан
33        R: радиус глобуса
34      Выход
35        x: координат x (начало координат в точке 0,0)
36        y: координат y (начало координат в точке 0,0)
37        """
38      lon1 = lon-lon0
39      if lon0 <> 0:
40          if lon1>180:
41              lon1 = -((180+lon0)+(lon1-180))
42          elif lon1<-180:
43              lon1 = (180-lon0)-(lon1+180)
44      theta = opt_theta(lat)
45      x = sqrt(8.0)/pi*R*lon1*cos(theta)
46      x = radians(x)
47      y = sqrt(2.0)*R*sin(theta)
48      return x, y
```

Преобразование координат в проекцию Мольвейде производится просто и реализовано в функции `transform2` (строка 25). Во входных аргументах передается, в частности, долгота среднего меридиана (`lon0`). Это, однако, требует дополнительной работы, потому что мы должны гарантировать, что по обе стороны среднего меридиана имеется 180°, хотя входные долготы берутся из исходных данных. Необходимые действия выполняются в блоке кода, начинающемся в строке 39.

Для тестирования проекции Мольвейде нужно внести всего два небольших изменения в код в листинге 2.13: импортировать функцию `transform2` и заменить в строке 11 `transform1` на `transform2`. На рис. 2.10 показаны сетка параллелей и меридианов и очертания материков в проекции Мольвейде.

Рис. 2.10 ❖ Проекция Мольвейде

У проекции Мольвейде есть интересная особенность – меридианы, отстоящие на 90° к востоку и западу от среднего меридиана, преобразуются в идеальные окружности. Поэтому мы можем спроецировать всю поверхность Земли на два круга и расположить их бок о бок. Это часто используется в атласах мира, чтобы разместить восточное и западное полушария на одной странице. На рис. 2.11 показаны пары кругов, получающиеся при разном выборе среднего меридиана.

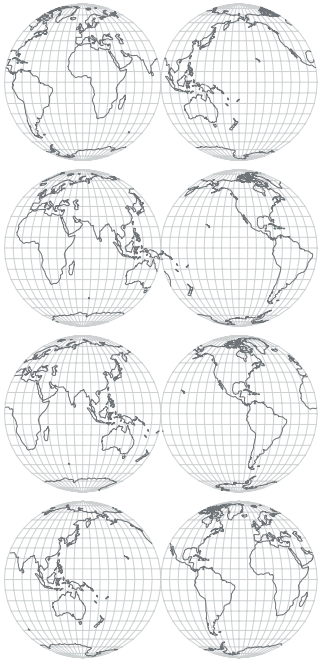

Рис. 2.11 ❖ Проекция Мольвейде карты мира на два круга.
Средние меридианы проходят соответственно по долготе 0°, 60°, 120° и 180°

Что, если мы не станем модифицировать долготы в функции `transform2` в листинге 2.14? Все математические формулы по-прежнему будут работать, но проекция окажется неправильной (рис. 2.12). В упражнениях мы увидим,

какая проблема не позволяет использовать в качестве среднего меридиана долготу, отличную от 0°. Однако это проблема данных, а не формулы и не рассмотренной выше реализации алгоритма.

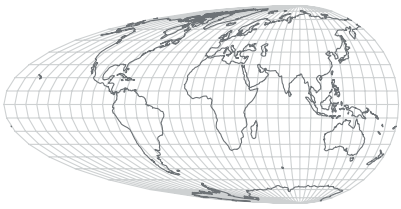

Рис. 2.12 ❖ Проекция Мольвейде со средним меридианом 90° без корректировки долгот

2.9. Примечания

Формулу гаверсинуса для вычисления расстояния по большой окружности можно найти в книге (1984). Формулу расстояния от точки до прямой см. в книге Deza and Deza (2010, стр. 86).

Существует много алгоритмов определения расположения точки относительно многоугольника (Haines, 1994; Huang and Shih, 1997), но в большинстве своем они основаны на теореме Жордана о простых кривых на плоскости (Jordan, 1887). Эта теорема утверждает, что простой многоугольник делит плоскость на две части: внутреннюю и внешнюю, – при этом сам многоугольник является их общей границей. Если не вдаваться в тонкости, то «простой» означает «несамопересекающийся». В этом случае интуитивно понятно, что означает «точка находится внутри многоугольника». Однако семантика нахождения внутри многоугольника – предмет споров, и алгоритмы, основанные на числе пересечений и числе оборотов, могут возвращать разные результаты.

Временная сложность всех алгоритмов, представленных в этой главе, составляет $O(N)$, где N – число сторон многоугольника (Huang and Shih, 1997). Хотя кажется, что это не очень плохо, полное время, необходимое для проверки нахождения точки внутри многоугольника, может быстро расти при увеличении числа сторон. Чтобы уменьшить его, применяются различные способы индексирования, описанные в последующих главах.

Для многоугольников специального вида, в частности монотонных, звездчатых или выпуклых, процедуру можно упростить и сделать более эффективной. Например, для выпуклого многоугольника существует алгоритм со сложностью $O(\log N)$, поскольку никакой луч не может пересекать более двух сторон (O'Rourke, 1998). Для выпуклых (и вообще монотонных) многоугольников можно спроектировать специальный алгоритм, ускоряющий проверку. Например, можно разбить многоугольник на два множества ломаных и каждое множество отсортировать по координатам Y. Тогда для проверки пересечения стороны с лучом можно будет применить двоичный поиск. У. Рэндольф

Фрэнклин (W. Randolph Franklin)[1] предложил также алгоритм для выпуклых многоугольников, состоящий из четырех шагов. На первом шаге находим уравнения прямых, содержащих стороны. На втором шаге записываем эти уравнения в виде $d = ax + by + c$. Точки, принадлежащие прямой, обращают правую часть в нуль. На третьем шаге нормируем каждое уравнение таким образом, что если подставить в него точку, лежащую внутри многоугольника, то результат будет положительным, а для точек вне многоугольника – отрицательным. На четвертом шаге подставляем координаты точки в каждое уравнение. Точка находится внутри многоугольника тогда и только тогда, когда все уравнения дают положительное значение.

Поскольку проекция Робинсона (Robinson, 1974) не основана на математических формулах[2], многие ученые пытались воспроизвести его оригинальную работу (Richardson, 1989; Snyder, 1990; Ipbuker, 2004). Предполагалось также, что Робинсон использовал в своей работе схему Эйткена (Richardson, 1989), похожую на алгоритм Невилла, приведенный в этой книге, но считающуюся устаревшей (Press et al., 2002, стр. 111). Математические детали многих проекций, в т. ч. проекции Мольвейде, см. в книге Snyder (1987).

2.10. Упражнения

1. Напишите на Python программу для вычисления манхэттенского расстояния между двумя точками.
2. Обсуждая вычисление площади многоугольника, мы ничего не сказали о многоугольниках, содержащих дырки. Остается ли приведенная формула площади справедливой для таких многоугольников? Если нет, напишите на Python программу, учитывающую эту возможность. Рекомендуем заглянуть в раздел приложения В.1.4, где обсуждается, как проецировать и отображать сложные многоугольники.
3. Вручную подготовьте набор многоугольников и протестируйте на нем алгоритмы принадлежности точки многоугольнику, рассмотренные в этой главе. Многоугольники могут быть выпуклыми или невыпуклыми, иметь разную форму и быть самопересекающимися.
4. Еще один способ протестировать алгоритмы – взять что-нибудь «реальное», т. е. реальный набор данных. В приложении В описана библиотека GDAL/OGR, которая умеет преобразовывать файлы данных существующих ГИС в простые структуры данных, используемые в этой главе, и тестировать алгоритмы принадлежности точки многоугольнику.

---

[1] PNPOLY – Point Inclusion in Polygon Test (http://www.ecse.rpi.edu/Homepages/wrf/Research/Short_Notes/pnpoly.html).

[2] В статье в газете New York Times (Wilford, 1988) приводится высказывание самого Робинсона: «Я выбрал своего рода артистический подход. Я представил себе формы и размеры, наиболее приятные глазу. И настраивал параметры, пока не достиг состояния, когда изменение любого из них ничего не улучшало. Тогда я вывел математическую формулу, дающую такой эффект. Большинство картографов начинают с математики».

5. Мы упомянули, что версия алгоритма на основе числа оборотов с тригонометрическими функциями работает долго. А так ли это? И снова воспользуйтесь функциями из библиотеки GDAL/OGR, рассматриваемой в приложении B.
6. Мы решили проблему среднего меридиана в коде, реализующем проекцию Мольвейде. Но если вы действительно захотите воспользоваться средним меридианом, долгота которого отлична от 0°, то многие береговые линии будут неправильно вытянуты по горизонтали. Дело в том, что наши данные привязаны к сетке, в которой долготы изменяются от –180° (к востоку) до 180° (к западу). Кроме того, если долгота среднего меридиана не кратна 10°, то мы будем иметь пару меридианов, проецируемых в другой интервал. Найдите способ решить эту проблему, так чтобы можно было правильно проецировать данные при любом среднем меридиане.
7. Программа на Python для проекции Робинсона не содержит кода для обработки других средних меридианов. Если вы справились с упражнением 6, то теперь самое время включить возможность задания среднего меридиана пользователем в программу для проекции Робинсона.
8. Теперь, разобравшись с проекцией Робинсона, мы можем поэкспериментировать с другими проекциями. Например, сможете ли вы применить аналогичный подход (как в табл. 2.1) для проецирования карты мира на треугольную сетку, где южный полюс представляется отрезком той же длины, что средний меридиан, а северный полюс – точкой? Разумеется, есть и другие формы сетки, которые было бы интересно исследовать.
9. В большинстве проекций масштаб в разных точках различен. Теперь, когда мы знаем, как получить данные для сетки, и умеем вычислять расстояния в разных системах координат (сферической и декартовой), можно систематически изучить вопрос о поведении коэффициента масштабирования на картографической проекции. Напишите соответствующую программу на Python и отобразите результаты на карте. Каковы закономерности искажения масштаба на проекциях Робинсона и Мольвейде?
10. На рис. 2.11 мы показали ряд карт в проекции Мольвейде, не приведя кода. Для построения этого рисунка использовались те же данные об очертании материков, что для других карт, и функция `transform2` из листинга 2.14. Напишите на Python программу, генерирующую карты на этом рисунке.

Глава 3

Наложение многоугольников

А за стенами фабрики на полмили вокруг воздух был пропитан густым запахом шоколада!

Роальд Дал «Чарли и шоколадная фабрика»

В ГИС операция вычисления пересечения отрезков встречается очень часто. Например, интересно знать, где траектория урагана пересекает береговую линию страны и какова площадь затронутой им области. В этой главе рассматриваются фундаментальные алгоритмы для решения подобных задач.

3.1. Пересечение отрезков

Алгоритм вычисления точки пересечения двух отрезков можно обобщить на произвольное множество отрезков. Прямолинейный подход – в цикле проверить каждую пару отрезков. Этот подход работает, но занимает много времени, особенно если количество отрезков велико и многие из них вообще не пересекаются, как, например, при вычислении точек пересечения двух составленных из многоугольников карт, когда отрезки одной карты пересекаются лишь с немногими отрезками другой.

Ниже мы познакомимся с более эффективным алгоритмом, когда рассматриваются только вероятные пересечения. Задумаемся, когда может возникнуть событие пересечения. Понятно, что шансы на пересечение есть только у отрезков, расположенных рядом. Рассмотрим четыре отрезка на рис. 3.1. В позициях e0, e1, e2 и e3 больше шансов пересечься у отрезков S0 и S1, потому что их концы расположены к этим прямым ближе, чем концы других отрезков (S0 и S2 или S1 и S2). При движении вправо в сторону от e3 ситуация меняется. Например, в позиции e4 отрезок S1 ближе к S2, и его пересечение с S2 более вероятно (они действительно пересекаются), чем с S0. Развивая эту мысль, представим себе заметающую прямую (пунктирную линию), которая движется слева направо, пересекая по пути все интересующие нас отрезки. В любой момент времени имеет смысл рассматривать только отрезки, пересекающие заметающую прямую. Точнее, мы рассматриваем лишь события, когда заметающая прямая встречает концы отрезков или точки их пересече-

ния. Поскольку заметающая прямая движется слева направо, мы можем быть уверены, что все точки пересечения слева от нее уже рассмотрены. В случае, показанном на рис. 3.1, заметающая прямая пересекает S0 и S1, а это значит, что нужно проверить, пересекаются ли эти отрезки. В данном случае пересекаются, поэтому мы включим их точку пересечения в состав событий и рассмотрим его, когда заметающая прямая дойдет до этого места. Важно отметить, что другие отрезки справа от заметающей прямой тоже могут пересекаться с S0 и S1, но мы рассмотрим эти случаи в свое время.

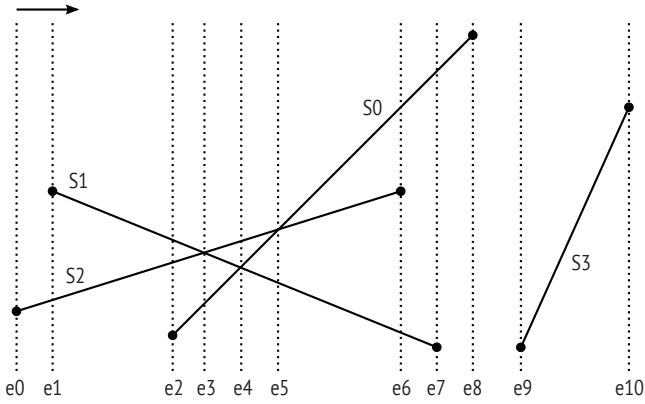

Рис. 3.1 ❖ Исследование пересечения отрезков с помощью заметающей прямой

Мы храним список отрезков, связанных с точками событий. Если точка события – это левый конец отрезка, то отрезок добавляется в список. Достигнув правого конца отрезка, мы удаляем его из списка. Отрезки, связанные с каждой точкой события, отсортированы по «возрастанию», снизу вверх. Так, для события e1 на рис. 3.1 отрезки хранятся в порядке S0, S1. Если два отрезка пересекаются, то создается новое внутреннее событие. Когда заметающая прямая встречает такую точку события, порядок следования отрезков в списке изменяется на противоположный. Например, в точке события e3 отрезки расположены в порядке S2, S0, S1. Поскольку это внутреннее событие, соответствующее пересечению S1 и S0, в следующей точке события e4 порядок будет S2, S1, S0 – именно в таком порядке отрезки следуют друг за другом в направлении снизу вверх в промежутке между событиями e3 и e4. Почему так важно поддерживать правильный порядок? Потому что от порядка зависит, как проверяется возможность пересечения. Например, в точке события e1 нужно проверить только, пересекается ли S1 с ближайшим (соседним) отрезком в списке, каковым является S0. То же самое относится и к внутренним событиям, например e3, в которых мы проверяем, пересекаются ли отрезки, встречающиеся в этой точке (т. е. S0 и S1), с соседними с ними отрезками (т. е. S2).

События, хранящиеся в списке, отсортированы так, что левые события всегда предшествуют правым. Чтобы эффективно организовать отрезки в каждой точке события, мы используем сбалансированное дерево, в котором

каждый узел представляет отрезок. В левой ветви узла находятся отрезки, расположенные ниже того, который представлен данным узлом, а в правой – отрезки, расположенные выше него. В правой части рис. 3.2 показано сбалансированное двоичное дерево для конфигурации отрезков в левой части этого рисунка в точке события, обозначенного штрихпунктирной линией.

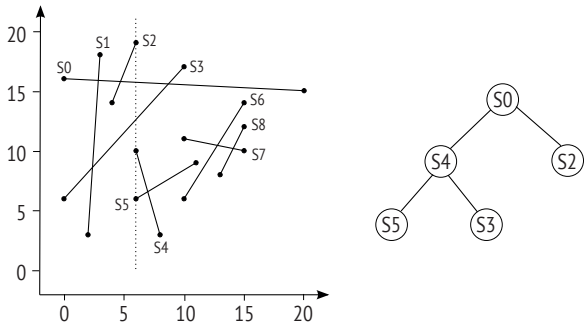

Рис. 3.2 ❖ Заметающая прямая.
Слева: набор прямолинейных отрезков, штрихпунктирной линией обозначено событие.
Справа: сбалансированное двоичное дерево для хранения отрезков в точке события

В листинге 3.1 описанные выше действия формально представлены в виде алгоритма Бентли–Оттмана, называемого также алгоритмом заметающей прямой. Здесь мы сначала описали процесс на псевдокоде, а ниже обсудим реализацию алгоритма на Python.

Листинг 3.1 ❖ Алгоритм Бентли–Оттмана нахождения точек пересечения прямолинейных отрезков

```
Инициализировать очередь событий EQ, включающую концы всех
отрезков. С каждым левым концом также ассоциируется идентификатор
сегмента. Инициализированная очередь EQ будет храниться
отсортированной по координатам X концов.
Инициализировать пустое двоичное дерево T для хранения отрезков
в точке события.
Пока EQ не пуста
  e = следующее событие в EQ и удалить его из очереди
  L = множество отрезков с левым концом e
  R = множество отрезков с правым концом e
  I = множество отрезков, для которых e - внутренняя точка
  Если в объединении L, R и I больше одного отрезка
      Сообщить о пересечении в точке события e
  Удалить отрезки, принадлежащие R и I, из T
  Вставить отрезки, принадлежащие L и I, в T
  Если L и I пусты
      s = первый отрезок в R, а sr и sl - правый и левый
          потомки s в T
      Если sr и sl пересекаются в точке, не принадлежащей EQ
          Добавить точку пересечения в EQ
```

```
16  Иначе
17      s' = самый левый отрезок из принадлежащих L и I в T
18      s'' = самый правый отрезок из принадлежащих L и I в T
19      sl = узел слева от s' в T
20      sr = узел справа от s'' в T
21      Если s' и sl пересекаются в точке, не принадлежащей EQ
22          Добавить точку пересечения в EQ
23      Если s'' и sr пересекаются в точке, не принадлежащей EQ
24          Добавить точку пересечения в EQ
```

Для реализации алгоритма заметающей прямой нам понадобятся структуры данных для хранения событий и очереди событий (листинг 3.2). Класс `Event` содержит точку и ассоциированные с ней отрезки. Но отрезки хранятся, только если событие соответствует левому концу, как следует из строки 1 псевдокода. Класс `EventQueue` инициализируется списком объектов, представляющих отрезки. Заметим, что все точки в очереди различны и что отрезки, ассоциированные с левым концом, добавляются в список `edges`, ассоциированный с событием (строка 35). В классе `EventQueue` имеются также дополнительные функции для выполнения различных операций. Так, функция `add` вставляет новое событие в отсортированную очередь. Функция `find` возвращает индекс первого вхождения события или точки в очередь; если точка отсутствует в очереди, возвращается –1.

Листинг 3.2 ❖ Модуль очереди событий (line_seg_eventqueue.py)

```
 1 from point import *
 2
 3 class Event:
 4     """
 5     Событие в очереди алгоритма заметающей прямой.
 6     В каждом объекте Event хранится точка события и
 7     ассоциированные с ней отрезки.
 8     """
 9     def __init__(self, p=None):
10         self.edges = []              # отрезки, ассоциированные с событием
11         self.p = p # event point
12     def __repr__(self):
13         return "{0}{1}".format(self.p,self.edges)
14
15 class EventQueue:
16     """
17     Очередь событий в алгоритме заметающей прямой.
18     """
19     def __init__(self, lset):        # инициализировать очередь событий
20         """
21         Конструктор EventQueue.
22         Вход
23           lset: список объектов Segment. По левому концу каждого отрезка
24                 создается событие
25         Выход
26           Отсортированный список событий, который является членом класса
```

```
        """
        if lset == None:
            return
        self.events = []
        for l in lset:
            e0 = Event(l.lp)
            inx = self.find(e0)
            if inx == -1:
                e0.edges.append(l)
                self.events.append(e0)
            else:
                self.events[inx].edges.append(l)
            e1 = Event(l.rp)
            if self.find(e1) == -1:
                self.events.append(e1)
        self.events.sort(key=lambda e: e.p)

    def add(self, e):
        """
        Добавляет событие e в очередь, обновляет список событий
        """
        self.events.append(e)
        self.events.sort(key=lambda e: e.p)

    def find(self, t):
        """
        Возвращает индекс события t, если оно присутствует в очереди.
        В противном случае возвращает -1.
        """
        if isinstance(t, Event):
            p = t.p
        elif isinstance(t, Point):
            p = t
        else: return -1
        for e in self.events:
            if p == e.p:
                return self.events.index(e)
        return -1

    def is_empty(self):
        return len(self.events) == 0
```

Поскольку операции над отрезками производятся часто – в каждой точке события, удобнее и эффективнее хранить отрезки в древовидной структуре, а не в списке. Подойдет любое сбалансированное двоичное дерево (например, AVL[1] или красно-черное), поддерживающее операции вставки, удаления и преобразования в отсортированный список. Мы воспользовались реализацией AVL-дерева из Python-модуля bintrees[2].

---

[1] AVL – инициалы ученых Адельсона–Вельского и Ландиса, придумавших самобалансирующееся двоичное дерево, в котором разность высот любых двух поддеревьев не превосходит 1.

[2] https://pypi.python.org/pypi/bintrees/2.0.1.

Для реализации алгоритма Бентли–Оттмана (листинг 3.3) нам понадобится несколько функций для извлечения важной информации из AVL-дерева. Функция `get_edges` возвращает все хранящиеся в дереве отрезки, для которых указанная точка является внутренней или правым концом. Внутренняя точка возможна в двух случаях: когда точка является ранее вычисленным пересечением или когда она является левым концом другого отрезка. Функция `get_lr` возвращает левого и правого соседа отрезка в дереве; если левого или правого соседа не существует, то вместо него возвращается сам отрезок. Наконец, функция `get_lrmost` возвращает самый левый и самый правый отрезок в переданном списке.

Листинг 3.3 ❖ Реализация алгоритма Бентли–Оттмана нахождения точек пересечения прямолинейных отрезков (line_seg_intersection.py)

```
from bintrees import AVLTree
from point import *
from intersection import *
from line_seg_eventqueue import *

def get_edges(t, p):
    """
    Возвращает отрезки, для которых точка p является
    внутренней или правым концом.
    """
    lr = []
    lc = []
    for s in AVLTree(t):
        if s.rp == p:
            lr.append(s)
        elif s.lp == p and s.status == INTERIOR:
            lc.append(s)
        elif sideplr(p, s.lp, s.rp) == 0:
            lc.append(s)
    return lr, lc

def get_lr(T, s):
    """
    Возвращает левого и правого соседа (ветви) узла p в дереве T
    """
    try:
        sl = T.floor_key(s)
    except KeyError:
        sl = None
    try:
        sr = T.ceiling_key(s)
    except KeyError:
        sr = None
    return sl, sr

def get_lrmost(T, segs):
    """
```

```
    Находит в дереве T самый левый и самый правый отрезок из присутствующих в списке segs
    """
    l = []
    for s in list(T):
        if s in segs:
            l.append(s)
        if len(l) < 1:
            return None, None
        return l[0], l[-1]

def find_new_event(s1, s2, p, q):
    ip = intersectx(s1, s2)
    if ip is None:
        return False
    if q.find(ip) is not -1:
        return False
    if ip.x>p.x or (ip.x==p.x and ip.y >= p.y):
        e0 = Event()
        e0.p = ip
        e0.edges = [s1, s2]
        q.add(e0)
    return True

def intersectx(s1, s2):
    """
    Проверяет, пересекаются ли два отрезка, и возвращает
    точку их пересечения.
    """
    if not test_intersect(s1, s2):
        return None
    p = getIntersectionPoint(s1, s2)     # точка пересечения
    return p

def intersections(psegs):
    """
    Реализация алгоритма Бентли-Оттмана.
    Вход
      psegs: список отрезков
    Выход
      intpoints: список точек пересечения
    """
    eq = EventQueue(psegs)
    intpoints = []
    T = AVLTree()
    L=[]
    while not eq.is_empty():             # для каждого события
        e = eq.events.pop(0)             # удалить событие
        p = e.p                          # получить точку события
        L = e.edges                      # отрезки, для которых p - левый конец
        R,C = get_edges(T, p)            # p: правый конец (R) или внутренняя (C)
        if len(L+R+C) > 1:               # пересечение в p принадлежит L+R+C
            for s in L+R+C:
                if not s.contains(p):    # если p внутренняя
```

```
                    s.lp = p             # изменить lp и
                    s.status = INTERIOR  # статус
            intpoints.append(p)
            R,C = get_edges(T, p)
            for s in R+C:
                T.discard(s)
            for s in L+C:
                T.insert(s, str(s))
            if len(L+C) == 0:
                s = R[0]
                if s is not None:
                    sl, sr = get_lr(T, s)
                    find_new_event(sl, sr, p, eq)
            else:
                sp, spp = get_lrmost(T, L+C)
                try:
                    sl = T.prev_key(sp)
                except KeyError:          # только для первого ключа
                    sl = None
                try:
                    sr = T.succ_key(spp)
                except KeyError:          # только для последнего ключа
                    sr = None
                find_new_event(sl, sp, p, eq)
                find_new_event(sr, spp, p, eq)
    return intpoints
```

Для демонстрации алгоритма мы воспользуемся отрезками на рис. 3.2 (листинг 3.4).

Листинг 3.4 ❖ Тестирование алгоритма Бентли–Оттмана (test_line_seg_intersection.py)

```
from line_seg_intersection import *

s = [ [[20,15],[0,16]], [[3,18],[2,3]], [[4,14],[6,19]],
      [[10,17],[0,6]], [[8,3],[5,10]], [[6,6],[11,9]],
      [[16,14],[10,6]], [[16,10],[10,11]],
      [[14,8],[16,12]] ]

psegs = [Segment(i, Point(s[i][0][0], s[i][0][1]),
                 Point(s[i][1][0], s[i][1][1]))
        for i in range(len(s))]

ints = intersections(psegs)
print "Число точек пересечения: ", len(ints)"
print ints
```

Эта программа печатает следующий результат:

```
Число точек пересечения: 7
[(2.4, 8.6), (2.9, 15.9), (4.7, 15.8), (6.6, 6.3),
 (8.7, 15.6), (13.3, 10.4), (15.1, 10.2)]
```

В табл. 3.1 показаны последовательные состояния очереди событий и отсортированные отрезки в дереве, а также сведения о каждой точке пересечения. Чтобы нагляднее представить точки события, мы используем префиксы L и R для обозначения левого и правого концов соответственно, а X – для обозначения внутренних точек или точек пересечения.

Таблица 3.1. Результат работы алгоритма заметающей прямой

Событие	Очередь событий	Отрезки в дереве	Точка пересечения
–	L3,L0,L1,R1,L2,L4,L5,R2,R4,L6,L7,R3,R5,L8,R7,R8,R6,R0	[]	
L3	L0,L1,R1,L2,L4,L5,R2,R4,L6,L7,R3,R5,L8,R7,R8,R6,R0	[3]	
L0	L1,R1,L2,L4,L5,R2,R4,X3X0,L6,L7,R3,R5,L8,R7,R8,R6,R0	[3,0]	
L1	X3X1,R1,L2,L4,L5,R2,R4,X3X0,L6,L7,R7,R5,L8,R7,R8,R6,R0	[1,3,0]	X3X1
X3X1	X0X1,R1,L2,L4,L5,R2,R4,X3X0,L6,L7,R7,R5,L8,R7,R8,R6,R0	[3,1,0]	X0X1
X0X1	R1,L2,L4,L5,R2,R4,X3X0,L6,L7,R7,R5,L8,R7,R8,R6,R0	[3,0,1]	
R1	L2,L4,L5,R2,R4,X3X0,L6,L7,R7,R5,L8,R7,R8,R6,R0	[3,0]	
L2	X0X2,L4,L5,R2,R4,X3X0,L6,L7,R7,R5,L8,R7,R8,R6,R0	[3,2,0]	X0X2
X0X2	L4,L5,R2,R4,X3X0,L6,L7,R7,R5,L8,R7,R8,R6,R0	[3,0,2]	
L4	L5,R2,R4,X3X0,L6,L7,R7,R5,L8,R7,R8,R6,R0	[4,3,0,2]	
L5	R2,X4X5,R4,X3X0,L6,L7,R7,R5,L8,R7,R8,R6,R0	[5,4,3,0,2]	
R2	X4X5,R4,X3X0,L6,L7,R7,R5,L8,R7,R8,R6,R0	[5,4,3,0]	X4X5
X4X5	R4,X3X0,L6,L7,R7,R5,L8,R7,R8,R6,R0	[4,5,3,0]	
R4	X3X0,L6,L7,R7,R5,L8,R7,R8,R6,R0	[5,3,0]	X3X0
X3X0	L6,L7,R7,R5,L8,R7,R8,R6,R0	[5,0,3]	
L6	L7,R7,R5,L8,R7,R8,R6,R0	[6,5,0,3]	
L7	R7,R5,L8,R7,R8,R6,R0	[6,5,7,0,3]	
R7	R5,L8,R7,R8,R6,R0	[6,5,7,0]	
R5	X6X7,L8,R7,R8,R6,R0	[6,7,0]	X6X7
X6X7	L8,R7,R8,R6,R0	[7,6,0]	
L8	X7X8,R7,R8,R6,R0	[8,7,6,0]	X7X8
X7X8	R7,R8,R6,R0	[7,8,6,0]	
R7	R8,R6,R0	[8,6,0]	
R8	R6,R0	[6,0]	
R6	R0	[0]	
R0	–	[]	

3.2. Наложение

Описанный выше алгоритм Бентли–Оттмана эффективен для вычисления точек пересечения заданного множества отрезков. Однако он не понимает, что отрезки взяты из разных карт, поэтому обращается со всеми отрезками одинаково. Во многих приложениях ГИС необходимо вычислять пересечение двух множеств отрезков, взятых из разных карт. Этот процесс называется наложением карт и, пожалуй, является самой распространенной из опера-

ций ГИС. Пусть, например, у нас есть карта плотности населения для каждого округа в штате и карта распределения лесов в штате. Наложение этих двух карт позволит определить, сколько леса приходится на человека в каждом округе.

Важнейшей задачей, которую нужно решить при наложении двух карт, является классификация возникающих многоугольников и правильное хранение информации из обеих исходных карт в каждом результирующем многоугольнике. В отличие от простого нахождения точек пересечения, на этот раз нам придется иметь дело с ломаными, ограничивающими многоугольники, и с самими многоугольниками. Чтобы лучше организовать эту процедуру, мы будем использовать специальную структуру данных – двусвязный список ребер (ДСР) (doubly connected edge list – DCEL), в котором учитываются связи между тремя видами геометрических объектов: точками, прямыми и многоугольниками. Рассмотрим множество многоугольников на рис. 3.3. В ДСР каждый отрезок представлен двумя полуребрами, имеющими противоположные направления. Полуребро из вершины i в вершину j обозначается $e_{i,j}$. Если задано, например, полуребро $e_{1,6}$, то парное ему (в данном случае $e_{6,1}$) называется его близнецом. Каждый многоугольник в ДСР называется гранью. Для каждой грани определено множество полуребер, составляющих ее границу, и эти ребра образуют цикл. Условимся, что грань всегда находится слева от своего ориентированного граничного полуребра. Например, граница грани f_2 состоит из полуребер $e_{2,1}$, $e_{1,6}$, $e_{6,7}$, $e_{7,3}$ и $e_{3,2}$. Кроме того, у каждого полуребра имеется начальная точка. Будем называть предыдущим полуребром `prev` то полуребро, которое заканчивается в начальной точке данного и принадлежит границе той же грани, что данное. Аналогично определяется следующее полуребро `next`.

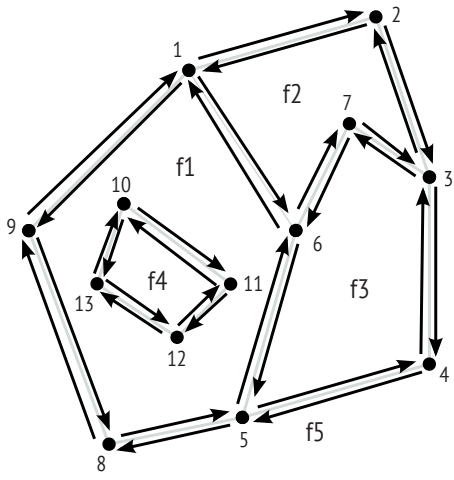

Рис. 3.3 ❖ Пример двусвязного списка ребер

Для хранения всей информации о множестве многоугольников в ДСР нам понадобится три структуры данных. Во-первых, для каждой вершины хранится ее индекс и исходящие из нее полуребра (см. табл. 3.2). Во-вторых, для

каждого полуребра хранится его индекс, начальная точка, близнец, грань, лежащая слева от него (инцидентная грань), а также предыдущее и следующее полуребро (см. табл. 3.3, в которой показана лишь часть полуребер). Наконец, для каждой грани (см. табл. 3.4) хранится ее индекс, одно из полуребер, составляющих ее границу (внешняя компонента), и полуребра, находящиеся внутри грани (внутренние компоненты). Внутренние компоненты грани важны для учета дырок внутри многоугольника. Для каждой дырки необходимо полуребро. Если в многоугольнике несколько дырок, то потребуется несколько полуребер. В нашем примере внутри грани f_1 есть одна дырка, поэтому необходима только одна внутренняя компонента (например, ребро $e_{10,11}$). Грань f_5 представляет область, внешнюю для всех многоугольников, у нее нет внешней компоненты.

Таблица 3.2. Вершины двусвязного списка ребер

Вершина	Список ребер
1	e1,2, e1,6, e1,9
2	e2,1, e2,3
3	e3,2, e3,7, e3,4
4	e4,3, e4,5
5	e5,6, e5,8, e5,4
6	e6,1, e6,7, e6,5
7	e7,3, e7,6
8	e8,5, e8,9
9	e9,1, e9,8
10	e10,11, e10,13
11	e11,10, e11,12
12	e12,13, e12,11
13	e13,10, e13,12

Таблица 3.3. Ребра двусвязного списка ребер

Полуребро	Начало	Близнец	Инцидентная грань	Следующее	Предыдущее
e1,2	1	e2,1	f5	e2,3	e9,1
e1,6	1	e6,1	f2	e6,7	e2,1
e1,9	1	e9,1	f1	e9,8	e6,1
e2,1	2	e1,2	f2	e1,6	e3,2
e2,3	2	e3,2	f1	e3,4	e1,2
e3,2	3	e2,3	f2	e2,1	e7,3
e3,7	3	e7,3	f3	e7,6	e4,3
e3,4	3	e4,3	f5	e4,5	e2,3
…					
e6,1	6	e1,6	f1	e1,9	e5,6
…					
e9,1	9	e1,9	f5	e1,2	e8,9
…					
e13,12	13	e12,13	f4	e12,11	e10,13

Таблица 3.4. Грани двусвязного списка ребер

Грань	Внешняя компонента	Внутренняя компонента
f1	e1,9	e10,11
f2	e2,1	None
f3	e7,6	None
f4	e11,10	None
f5	None	e1,2

Алгоритм наложения многоугольников реализован в листинге 3.5, где функция `overlay` возвращает список точек пересечения двух полигональных карт. Эта функция принимает два входных параметра: список отрезков (`psegs`) и неполный ДСР (`D`), содержащий только информацию о вершинах и полуребрах, непосредственно скопированную из обеих исходных карт (ниже мы опишем, как подготовить эти данные). Тело функции похоже на функцию `intersections` из листинга 3.3, что и неудивительно. Дополнительно проверяется, принадлежит ли точка пересечения отрезкам из разных карт. Мы можем быть уверены, что отрезки из каждой карты получают одно и то же значение атрибута `c`, но для разных карт оно различно. Таким образом, два пересекающихся отрезка принадлежат разным картам, если значение `c` в них различно (строка 88). Если это не так, то в `D` ничего обновлять не нужно. Если же имеет место пересечение двух карт, то возможно три случая: ребро одной карты, проходит через вершину другой, ребро одной карты пересекает ребро другой во внутренней точке и вершина одной карты совпадает с вершиной другой. При выбранном способе представления многоугольников и отрезков других случаев в точках пересечения карт нет. В этой книге мы покажем только, как обрабатывается первый случай (строка 93), остальные два рассматриваются аналогично.

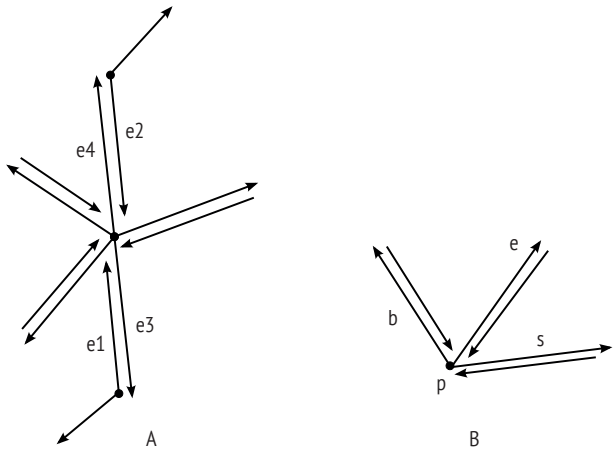

Рис. 3.4 ❖ Обработка случая,
когда ребро одной карты проходит через вершину другой

Случай, когда ребро одной карты проходит через вершину (или вершины) другой, обрабатывается в функции `handle_edge_vertex` в листинге 3.5. Этот

случай не требует создания новых вершин, но нам придется обновить информацию о вершинах и полуребрах (в коде полуребро обозначается идентификатором `hedge`). Грани будут обновлены позже. Пусть `p` – точка пересечения, являющаяся вершиной в одной карте, а `e1` и `e2` – два полуребра того исходного ребра другой карты, которое проходит через `p` (строка 30). Как показано на рис. 3.4A, мы можем сохранить исходные начала этих двух полуребер, но должны изменить следующее и предыдущее полуребра `e1` и `e2`. С этой целью мы создадим два новых полуребра `e3` и `e4` с начальной точкой `p` (строки 31 и 32). Нужно позаботиться о том, чтобы у `e3` и `e4` были правильные следующее и предыдущее полуребра (строки 34–37). В концах исходного ребра мы теперь имеем две пары близнецов: `e1` и `e3`, `e2` и `e4`. Мы также знаем, что для `e3` и `e4` следующими полуребрами являются следующие полуребра исходных `e2` и `e1` соответственно. Чтобы обновить новое полуребро (обозначенное `e`), исходящее из точки пересечения `p`, мы отсортируем ребра с начальной точкой `p` по углам и определим два полуребра, находящихся слева (`b`) и справа (`s`) от `e` (рис. 3.4B). Функция `update_intersect_dcel` (строка 6) содержит детали процедуры обновления ДСР в том, что касается этих полуребер. После обновления (строки 44 и 45) два новых полуребра добавляются в `D`.

Листинг 3.5 ❖ Наложение двух карт (overlay.py)

```
 1 from line_seg_intersection import *
 2 from dcel import *
 3
 4 class OvearlayError(Exception): pass
 5
 6 def update_intersect_dcel(hl, e):
 7     """
 8     Обновляет полуребра, связанные с e, зная список полуребер hl
 9     """
10     l = len(hl)
11     if l<2:
12         raise OvearlayError(
13             "Overlay/DCEL error: single edge for vertex")
14     big, small = l-1, 0
15     for i in range(l):
16         if e.angle > hl[i].angle:
17             big,small = i-1,i
18             break
19     b,s = hl[big], hl[small]
20     e.prevhedge = b.twin
21     e.twin.nexthedge = s
22     b.prevhedge = e.twin
23     b.twin.nexthedge = e
24     s.prevhedge = e.twin
25
26 def handle_edge_vertex(L, R, C, p, D):
27     e = C[0].edge
28     v1 = Vertex(C[0].lp0.x ,C[0].lp0.y) # получить полуребра для e
29     v2 = Vertex(C[0].rp.x ,C[0].rp.y)
```

```
30      e1,e2 = D.findhedges(v1,v2)          # начало и конец of e
31      e3 = Hedge(v1, Vertex(p.x,p.y))      # hedge,p в качестве начала
32      e4 = Hedge(v2 ,Vertex(p.x,p.y))      # hedge,p в качестве начала
33      # обновление в окрестности концов e:
34      e1.twin = e3
35      e3.twin = e1
36      e2.twin = e4
37      e4.twin = e2
38      e3.nexthedge = e2.nexthedge
39      e4.nexthedge = e1.nexthedge
40
41      v = D.findvertex(p)                  # обновление в окрестности p
42      v.sortincident()
43      hl = v.hedgelist
44      update_intersect_dcel(v.hedgelist, e3)
45      update_intersect_dcel(v.hedgelist, e4)
46      # добавить два новых ребра с началом p
47      v.hedgelist.append(e3)
48      v.hedgelist.append(e4)
49      # e1, e2 обновлены в D благодаря ссылкам
50      D.hedges.append(e3)
51      D.hedges.append(e4)
52
53  def handle_edge_edge(L, R, C, p, D):
54      Pass
55
56  def handle_vertex_vertex(L, R, C, p, D):
57      Pass
58
59  def overlay(psegs, D):
60      """
61      Вычисляет результат наложения двух полигональных карт.
62      Вход
63        psegs: список объектов Segment. Атрибут с в каждом отрезке
64        указывает на исходную карту
65        D: частичный ДСР с исходными полуребрами и вершинами
66      Выход
67        intpoints: список точек пересечения
68      """
69      eq = EventQueue(psegs)
70      intpoints = []
71      T = AVLTree()
72      L=[]
73      while not eq.is_empty():             # для всех событий
74          e = eq.events.pop(0)             # удалить событие
75          p = e.p                          # точка события
76          L = e.edges                      # отрезки, для которых p – левый конец
77          R,C = get_edges(T, p)            # пересечение в p отрезков из L+R+C
78          if len(L+R+C) > 1:
79              for s in L+R+C:
80                  if not s.contains(p):
81                      s.lp = p
```

```
                    s.status = INTERIOR
                intpoints.append(p)
                R,C = get_edges(T, p)
                c1 = (L+R+C)[0].c
                cross = False
                for l in L+R+C:
                    if c1 is not l.c:
                        cross = True
                        break
                # обновить списки вершин и ребер
                if cross is True:
                    if len(C) == 1:     # СЛУЧАЙ 1: ребро проходит через вершину
                        handle_edge_vertex(L, R, C, p, D)
                    if len(C) > 1:      # СЛУЧАЙ 2: два ребра пересекаются
                        handle_edge_edge(L, R, C, p, D)
                    if len(C) == 0:     # СЛУЧАЙ 3: вершина на вершине
                        handle_vertex_vertex(L, R, C, p, D)
            for s in R+C:
                T.discard(s)
            for s in L+C:
                T.insert(s, 1)
            if len(L+C) == 0:
                s = R[0]
                if s is not None:
                    sl, sr = get_lr(T, s)
                    y = find_new_event(sl, sr, p, eq)
            else:
                lp, lpp = get_lrmost(T, L+C)
                try:
                    sl = T.prev_key(lp)
                except KeyError:        # только для первого ключа
                    sl = None
                try:
                    sr = T.succ_key(lpp)
                except KeyError:        # только для последнего ключа
                    sr = None
                find_new_event(sl, lp, p, eq)
                find_new_event(sr, lpp, p, eq)
    return intpoints
```

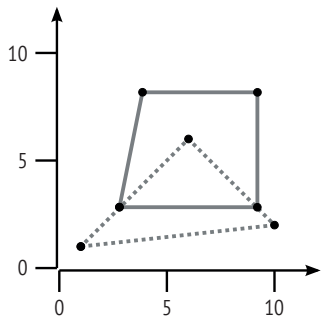

Рис. 3.5 ❖ Тестирование алгоритма наложения

Для демонстрации того, как алгоритм Бентли–Оттмана можно применить для вычисления результата наложения двух карт, мы воспользуемся примером на рис. 3.5. Здесь предполагается, что картам соответствуют многоугольники, показанные сплошной и пунктирной линиями. Листинг 3.6 содержит код инициализации и вычисления наложения. Вершины карт представлены переменными `pgon1` и `pgon2`, а их ребра – переменными `edges1` и `edges2` соответственно. Карты хранятся в ДСР `d[0]` и `d[1]`. Для реализации структуры данных ДСР мы возьмем за основу существующий Python-модуль[1]. Функции `load` модуля DCEL передаются списки точек и ребер, необходимые для создания объекта ДСР. Еще один объект ДСР понадобится для хранения объекта `D` с вычисленным наложением. Мы инициализируем `D` объединением вершин и полуребер из обоих источников (строки 13 и 14). Чтобы построить наложение карт, нам еще нужно подсчитать количество отрезков в обеих картах, они хранятся в списках отрезков `ps1` и `ps2`, а атрибуту `c` присваивается значение 0 или 1, чтобы различить карты (строки 28 и 31). Затем эти отрезки объединяются в один список `s` (строка 32), который является одним из двух параметров, передаваемых функции `overlay`. Второй параметр – объект `D`, который функция обновляет.

После возврата из функции `overlay` все связи между вершинами и полуребрами в `D` должны быть корректны. Теперь настало время зарегистрировать грани. Вместо того чтобы повторно использовать грани исходных карт, мы построим грани в `D` с чистого листа. В цикле `while`, начинающемся в строке 38, перебираются отрезки для каждой грани. Первая наша задача – понять, сколько граней будет в наложении и какие полуребра ограничивают эти грани. Коль скоро множества полуребер и вершин правильны, мы можем взять любое полуребро и, следуя по хранящимся в ДСР ссылкам на следующее, построить граничные циклы (строка 43).

После построения циклов все полуребра, хранившиеся в `D`, будут удалены. Но это не проблема, поскольку все они по-прежнему хранятся, только теперь в переменной `cycles`. Следующий шаг – сопоставить правильную грань каждому полуребру (строка 57). Чтобы проверить, является ли граничный цикл внешней компонентой грани, нужно лишь вычислить знак площади. Из предыдущей главы мы знаем, что площадь будет отрицательна, когда ребра обходятся против часовой стрелки.

Листинг 3.6 ❖ Тестирование алгоритма наложения (test_overlay_1.py)

```
1 from overlay import *
2
3 pgon1 = [ [3,3], [9,3], [8,8], [4,8] ]
4 edges1 = [ [0,1], [1,2], [2,3], [3,0] ]
5 pgon2 = [ [1,1], [6,6], [10,2] ]
6 edges2 = [ [0,1], [1,2], [2,0] ]
7
8 d=[ Dcel(), Dcel() ]
```

[1] https://pypi.python.org/pypi/dcel/0.1.1. Модифицированный код имеется на сайте этой книги на Github.

```
d[0].load(pgon1, edges1)
d[1].load(pgon2, edges2)

D = Dcel()
D.vertices = d[0].vertices + d[1].vertices
D.hedges = d[0].hedges + d[1].hedges

s1 = []
for i in range(len(pgon1)-1):
    s1.append([pgon1[i], pgon1[i+1]])
s1.append([pgon1[-1], pgon1[0]])

s2 = []
for i in range(len(pgon2)-1):
    s2.append([pgon2[i], pgon2[i+1]])
s2.append([pgon2[-1], pgon2[0]])

ps1 = [Segment(i, Point(s1[i][0][0],s1[i][0][1]),
               Point(s1[i][1][0],s1[i][1][1]), 0)
       for i in range(len(s1))]
ps2 = [Segment(i+len(s1), Point(s2[i][0][0],s2[i][0][1]),
               Point(s2[i][1][0],s2[i][1][1]), 1)
       for i in range(len(s2))]
s = ps1+ps2

ints = overlay(s, D)

hl = D.hedges
cycles = []
while len(hl) is not 0:           # построить все граничные циклы
    c = []
    e0 = hl.pop()
    e = e0
    c.append(e)
    while True:
        print e,
        e1 = e.nexthedge
        if e1 is not e0:
            c.append(e1)
            hl.remove(e1)
            e = e1
        else:
            break
    cycles.append(c)
    print

# создать грани и связать их с полуребрами
faces = []
for c in cycles:
    f = Face()
    f.wedge = c[0]
    for e in c:
```

```
61          e.face = f
62          D.hedges.append(e)
63      if f.area() < 0:
64          f.external = True
65      else:
66          f.external = False
67      faces.append(f)
68  D.faces = faces
```

Для примера на рис. 3.5 получаются следующие полуреберные циклы, ограничивающие многоугольные грани после наложения карт:

```
(9,3)->(10,2) (10,2)->(1,1) (1,1)->(3,3) (3,3)->(4,8)
  (4,8)->(8,8) (8,8)->(9,3)
(9,3)->(6,6) (6,6)->(3,3) (3,3)->(9,3)
(3,3)->(6,6) (6,6)->(9,3) (9,3)->(8,8) (8,8)->(4,8)
  (4,8)->(3,3)
(3,3)->(1,1) (1,1)->(10,2) (10,2)->(9,3) (9,3)->(3,3)
```

3.3. Примечания

В литературе, посвященной ГИС, уже давно осознано, как важно хранить данные из обеих участвующих в наложении карт, чтобы в результирующей карте присутствовала исходная информация. Например, в работе Frank (1987) описано три типа информации, которую следует хранить в результате наложения: негеометрические свойства, пространственные идентификаторы, например ключи всех многоугольников, и геометрические описания объектов, в частности что они собой представляют: точки, линии или многоугольники. Описанное выше использование ДСР позволяет эффективно достичь всех этих целей. В этой книге нас будет интересовать только геометрический аспект. Но в изложенные ранее алгоритмы нетрудно включить и другие аспекты наложения. Дополнительные сведения о наложении многоугольников см. в великолепной книге de Berg et al. (1998)[1].

Насколько эффективен алгоритм заметающей прямой (Bentley and Ottmann, 1979) и алгоритм наложения? Алгоритм нахождения точек пересечения отрезков имеет сложность $O(n \log n + k \log n)$, где n – число отрезков, а k – число пересечений. В некоторых частных случаях можно ожидать более высокой производительности. Например, если мы хотим всего лишь проверить, существует ли хотя бы одна точка пересечения отрезков, то алгоритм, описанный в работе Shamos and Hoey (1976), позволит вернуть ответ за время $O(n \log n)$. Если имеется два множества отрезков, например красные и синие, и отрезки в каждом множестве не пересекаются друг с другом, то алгоритм заметания трапеций, разработанный Ченом (Chan, 1994), сможет вычислить все пересечения за время $O(n \log n + k)$. В литературе также описаны способы

[1] *Берг М., Чеонг О., Кревельд М., Овермарс М.* Вычислительная геометрия. 3-е изд. М.: ДМК-Пресс, 2017.

распараллеливания алгоритма нахождения пересечения (Franklin et al., 1989; Healey et al., 1998).

3.4. Упражнения

1. Мы говорили, что алгоритм Бентли–Оттмана теоретически эффективнее полного перебора. Настало время проверить это эмпирически. Напишите на Python программу, которая будет случайным образом генерировать различные множества отрезков, выполнять для них оба алгоритма и сравнивать время работы.
2. Мы знаем, что файл фигур (shapefile) стал самым популярным форматом хранения геопространственных данных. Однако он не содержит никакой топологической информации. Иными словами, из файла фигур нельзя узнать, как связаны между собой различные элементы. Структура данных ДСР, описанная в этой главе, безусловно, содержит такую информацию. Прочитайте справку по пакету GDAL/OGR в приложении В и напишите на Python программу, которая будет конвертировать файл фигур для многоугольника в формат ДСР.
3. Разработанная в этой главе функция наложения неполна. Мы не включили код для двух из трех возможных случаев. Восполните этот пробел и протестируйте свою реализацию на более сложных входных данных.

Часть II

ИНДЕКСИРОВАНИЕ ПРОСТРАНСТВЕННЫХ ДАННЫХ

Глава 4

Индексирование

Как у наших у ворот
Чудо-дерево растёт.
Корней Чуковский

Есть много ситуаций, когда нам нужно найти некое подмножество пространственных данных. Например, чтобы интерполировать значение переменной на точку, в которой нет наблюдений, мы должны использовать имеющиеся значения в близких точках. Вообще, такая задача возникает, когда требуется что-то узнать о точках вблизи данной. Другой пример – чтобы извлечь информацию о многоугольнике на карте, по которому пользователь щелкнул мышью, требуется найти многоугольник, содержащий местоположение мыши, игнорируя остальные многоугольники, не имеющие ничего общего с этой точкой. Так ли важно делать это быстро? Иными словами, играет ли быстродействие большую роль в этом случае? Давайте рассмотрим аналогичную ситуацию – поиск слова «индексирование» в этой книге. Кто-то скажет – да вот же оно, на этой самой странице. Это правильно. Но откуда нам это знать? Если мы закроем книгу, то сможем ли потом найти его так же быстро? А как насчет других слов? Если нам известно, что слово «индексирование» встречается только в этой главе, то задача упрощается, потому что на предыдущих страницах его быть не может (и мы можем их игнорировать). Тем самым мы индексируем информацию: ассоциируем со словом «индексирование» номер главы, а номера глав храним в отсортированном виде, начиная с 1. Точно так же можно отсортировать по алфавиту все слова (или хотя бы самые важные) и с каждым из них ассоциировать номер страницы. Именно для этого и предназначен предметный указатель в конце книги, который позволяет быстро найти слово «индексирование», потому что можно пропустить слова, начинающиеся с букв «А», «Б», «В» и т. д. Если речь идет о событиях, происходивших в прошлом, то можно связать с ними год или век и тем самым рассматривать события как временную последовательность и индексировать их по времени.

Индексирование лежит в основе стратегии, которую люди используют уже тысячи лет: разделяй и властвуй. Например, когда мы впервые ищем слово «индексирование» в указателе к этой книге, мы еще не знаем, где находится раздел, относящийся к букве И. Но мы можем раскрыть указатель посередине и посмотреть, есть ли там раздел И. Если вместо И мы нашли К, то знаем, что нет смысла искать на следующих страницах, потому что И там точно не

будет. Затем можно перейти в конец первой четверти списка и посмотреть, расположен ли находящийся там раздел до или после И. Каждый раз, совершая такое действие, мы сужаем область поиска. По сути дела, мы делим всю область поиска пополам и отбрасываем ту половину, в которой искомого точно нет. Оставшуюся половину снова делим пополам и продолжаем так поступать, пока не найдем то, что ищем.

Индексирование можно выполнять по-разному. Телефонная книга отсортирована по алфавиту, а события – по времени. Иногда, если имеется много данных, сведенных в таблицу, мы можем выбирать способ сортировки. Например, сайт expedia.com позволяет сортировать данные об авиарейсах по цене, по времени вылета и по времени прибытия – в порядке возрастания или убывания. Чтобы реализовать столь разнообразные требования на компьютере, мы абстрагируем различные виды сортировки и индексирования, применяя древовидную структуру. Рассматривая нахождение точек пересечения в главе 3, мы видели, как дерево можно использовать для сортировки прямолинейных отрезков. В этой главе мы введем некоторые базовые свойства деревьев, а затем обсудим, чем деревья могут быть полезны для индексирования пространственных данных.

На рис. 4.1 показана древовидная структура. Обычно у дерева имеется корень, который является основной, а чаще единственной точкой доступа к дереву. Иными словами, при поиске информации в дереве мы всегда начинаем с корня. В нашем случае корнем является узел, содержащий число 4. Дерево состоит из множества узлов, в которых хранится информация, и эти узлы организованы иерархически. Узел А называется родителем узла В, если А расположен в иерархии прямо над В, а узел В в этом случае называется дочерним узлом А. У корня дерева, естественно, нет родителя. В дереве имеются узлы, не имеющие дочерних, они называются листовыми. В примере на рис. 4.1 листовыми являются узлы 1, 3, 5 и 7. Узел вместе со всеми его потомками (дочерними узлами, их дочерними узлами и т. д.) образует поддерево исходного дерева.

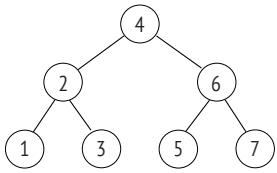

Рис. 4.1 ❖ Двоичное дерево для хранения чисел 1, 2, ..., 7 в отсортированном виде

Дерево на рис. 4.1 – пример специальной древовидной структуры, называемой двоичным деревом, потому что у каждого узла может быть не более двух дочерних. В данном конкретном случае дерево представляет последовательность чисел. Для каждого узла гарантируется, что число в левом дочернем узле меньше числа в самом узле, а число в правом дочернем узле – больше. Существует путь из корня к любому узлу, а количество узлов на этом пути называется глубиной, или высотой, узла.

Глубина всего дерева определяется количеством узлов на самом глубоком пути. На рис. 4.1 глубина всех листовых узлов одинакова. Это идеально сбалансированное дерево. Идеально сбалансированные деревья встречаются редко – для этого нужно, чтобы в дереве было ровно $2^{H+1} - 1$ узлов, где H – глубина (или высота) дерева. Однако можно попробовать создать сбалансированные деревья, в которых разность высот левого и правого поддеревьев любого узла не превышает 1.

Чтобы воспользоваться двоичным деревом для поиска числа, мы начинаем с корня. Если в корне уже хранится искомое число, то поиск останавливается. В противном случае мы решаем, по какой ветви идти дальше. Если искомое число меньше корня, то идем по левой ветви, а в правую заглядывать не нужно. Если же искомое число больше корня, то идем по правой ветви и игнорируем левую. Этот процесс сравнения с числом в узле и выбора следующей ветви продолжается, пока искомое число не будет найдено или не будет установлено, что его нет в дереве.

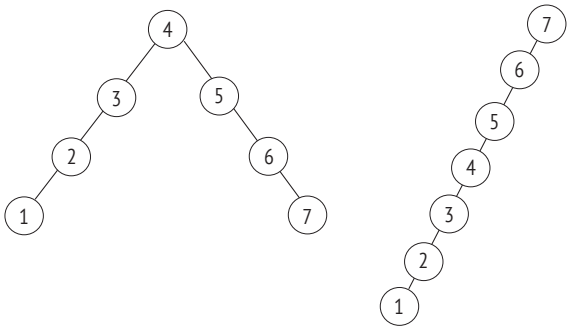

Рис. 4.2 ❖ Два несбалансированных дерева для хранения чисел 1, 2, ..., 7. Правое дерево, очевидно, несбалансированное. В левом дереве глубина левого поддерева узла 3 равна 3, а глубина его правого поддерева равна 0 (т. е. разность больше 1)

Очевидно, что процесс поиска займет мало времени, если глубина дерева невелика. Дерево на рис. 4.1 оптимально для хранения семи чисел, потому что не существует дерева меньшей глубины. На рис. 4.2 показаны два других дерева, в которых хранятся те же данные. Ни одно из них не сбалансировано. Для поиска числа в этих деревьях в общем случае потребуется больше шагов, а значит, и больше времени.

Можно ли применить идею индексирования с помощью дерева к пространственным данным? Да, но с некоторыми дополнениями: индексирование пространственных данных устроено иначе, чем привычное для нас индексирование одномерных данных. У объектов в пространстве больше измерений; обычно мы имеем дело с двумя измерениями, но можно добавить и третье – высоту или возвышение. Простейший способ индексирования пространственных данных – ограничиться только одним измерением. Например, семь точек на рис. 4.3 можно отсортировать по горизонтальной координате, что даст последовательность *B*, *A*, *E*, *C*, *D*, *F*, *G*. Если мы хотим

найти точки, находящиеся внутри некоторого прямоугольника, то можем с помощью такой сортировки сузить диапазон координат *X*. Поскольку координаты *X* отсортированы, это можно сделать быстро, воспользовавшись стратегией «разделяй и властвуй», описанной выше. Но это полезно, только если мы хотим искать точки по координате *X*. Координаты *Y* часто не зависят от *X*, поэтому поиск точек в каком-то диапазоне *X* не позволит исключить ни одной точки. Например, у точки *A* координата *X* меньше, чем у всех остальных точек, кроме *B*. Но это ничего не говорит нам о том, будет ли координата *Y* точки *A* меньше или больше, чем у остальных. Таким образом, либо придется использовать полный перебор всех точек с желаемыми координатами *X*, либо отсортировать точки по координате *Y*.

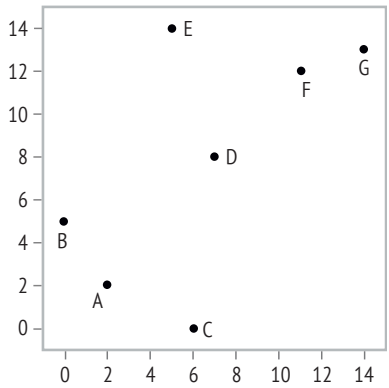

Рис. 4.3 ❖ Гипотетический набор данных из семи точек

Для реализации идеи двоичного дерева сначала определим структуру данных для хранения информации об узле дерева. В листинге 4.1 приведен код Python-класса для этой цели. Поскольку Python не проверяет типы, члены этого класса могут иметь любой тип, при условии что объекты этого типа можно сравнивать.

Листинг 4.1 ❖ Класс узла двоичного дерева

```
1 class node():
2     def __init__(self, data, left, right):
3         self.data = data
4         self.left = left
5         self.right = right
6     def __repr__(self):
7         return str(self.data)
```

Мы уже говорили, что все операции с двоичным деревом начинаются с корня. Иными словами, программа видит дерево как узел, из которого может добраться до любого другого узла. Раз так, то мы можем производить поиск в дереве рекурсивно, пользуясь Python-функцией в листинге 4.2. Дерево передается ей просто как узел `t`, а искомая информация – в параметре `d`. Булева переменная `is_find_only` сообщает, нужно ли искать значение (`True`)

или родителя узла, который содержал бы отсутствующее в дереве значение, если бы оно там было (False). Последнее полезно, если нужно решить, в какое место дерева вставить новый узел – он должен быть дочерним узлом найденного родителя. Поиск продолжается по левой или по правой ветви в зависимости от данных, хранящихся в текущем узле, и искомых данных (строки 4–7). Функция search вызывает саму себя (строка 15), пока не произойдет одно из трех: достигнут конец дерева (строка 2), найден узел, содержащий данные (строка 8), или выяснилось, что данных в дереве нет, но нужно вернуть место, куда их можно будет вставить (строка 13).

Листинг 4.2 ❖ Поиск в двоичном дереве

```
 1 def search_bt(t, d, is_find_only=True):
 2     if t is None:
 3         return
 4     if d<t.data:
 5         next = t.left
 6     else:
 7         next = t.right
 8     if t.data==d:
 9         if is_find_only:
10             return t
11         else:
12             return
13     if not is_find_only and next is None:
14         return t
15     return search_bt(next, d, is_find_only)
```

Теперь можно написать функцию, которая будет вставлять новый узел в дерево (листинг 4.3). Сначала она находит родительский узел для узла, который мы собираемся вставить (строка 2), пользуясь показанной выше функцией поиска. Если поиск вернул None, т. е. такой узел в дереве уже есть, то делать ничего не нужно (строки 2 и 3). В противном случае мы создаем новый узел (строка 5) и размещаем его слева (строка 7) или справа (строка 9) от родителя в зависимости от данных.

Листинг 4.3 ❖ Функция вставки в двоичное дерево

```
1 def insert(t, d):
2     n = search_bt(t, d, False)
3     if n is None:
4         return
5     n0 = node(d, left=None, right=None)
6     if d<n.data:
7         n.left = n0
8     else:
9         n.right = n0
```

Теперь мы можем построить дерево по набору данных. Будем предполагать, что каждое значение встречается в наборе только один раз, потому что в дереве одно значение нельзя сохранить дважды. В листинге 4.4 приведен код, который создает двоичное дерево по заданному списку целых чисел.

Сначала мы определяем функцию bt, которая возвращает корень вновь созданного дерева. Затем определяем функцию, которая распечатывает хранящиеся в дереве данные в порядке возрастания (строка 7). В конце листинга мы случайным образом перемешиваем элементы списка (строка 22) и создаем дерево по рандомизированным данным (строка 23). Функция bt_print должна вывести такой же результат, как и раньше.

Листинг 4.4 ❖ Тестирование двоичного дерева

```
 1 def bt(data):
 2     root = node(data=data[0], left=None, right=None)
 3     for d in data[1:]:
 4         insert(root, d)
 5     return root
 6
 7 def bt_print(t):
 8     if t.left:
 9         bt_print(t.left)
10     print t
11     if t.right:
12         bt_print(t.right)
13
14 data = range(10)
15 t = bt(data)
16 search_bt(t, 0)
17 bt_print(t)
18 search_bt(t, 10)
19 insert(t, 10)
20 bt_print(t)
21 import random
22 random.shuffle(data)
23 t1 = bt(data)
24 bt_print(t1)
```

Это код не предназначен для создания сбалансированного двоичного дерева – для этого необходимы дополнительные операции после вставки узла. Однако описанная схема будет полезна при обсуждении деревьев для хранения пространственных данных ниже. Далее в этой части книги мы познакомимся с несколькими древовидными структурами для эффективного индексирования пространственных данных, в основном точек и многоугольников. Во всех этих методах несколько измерений обрабатываются равноправно, так что все они вносят вклад в уменьшение времени поиска. Стоит отметить, что, хотя пространственные данные отличаются от одномерных, описанные принципы по-прежнему действуют. В следующей главе мы увидим, что код напоминает тот, что разработан выше.

4.1. Упражнения

1. Спроектируйте серию вычислительных экспериментов, позволяющих сравнить поиск в двоичном дереве с линейным поиском.

Глава 5

*k*D-деревья

> Никакое дело не покажется невыполнимым, если разбить его на мелкие части.
>
> *Генри Форд*

*k*D-деревья – это семейство древовидных структур данных, специально предназначенное для работы с многомерными данными. Буква *k* в названии обозначает размерность данных. В данном случае, поскольку мы имеем дело с пространственными данными в двумерной области, нас будут интересовать 2D-деревья. Хотя все пространственные структуры данных, обсуждаемые в этой части книги, двумерные, идея *k*D-дерева обобщается и на более высокие размерности. Существует много видов *k*D-деревьев, в этой книге мы будем заниматься двумя основными типами: точечными и точечно-регионными.

5.1. Точечные *k*D-деревья

Для иллюстрации работы точечного *k*D-дерева будем использовать семь точек, перечисленных в табл. 5.1. Эти же точки отмечены в левой части рис. 5.1. В данном случае предполагается, что точки добавляются в дерево в порядке *A*, *B*, ..., *G* – мы увидим, что такой порядок имеет решающее значение. В табл. 5.1 показаны координаты этих точек. Сначала мы используем первую точку *A* для разбиения пространства, заполненного всеми точками, на две части – левую и правую – по координате *X* точки. На рис. 5.1 вертикальная прямая обозначает общую границу этих двух частей. Одновременно мы создаем корень 2D-дерева (поскольку *A* – первая из использованных точек) с двумя ветвями. Затем мы переходим ко второй точке, *B*. На этот раз для разбиения плоскости на две части – верхнюю и нижнюю – используется координата *Y* точки *B*; эти две части разделены на рисунке горизонтальной прямой, проходящей через *B*. Чтобы вставить точку *B* в дерево, мы начинаем с корня (в нем, напомним, находится первая точка *A*). Поскольку координата *X* точки *B* меньше, чем у точки *A*, мы идем от корня по левой ветви. В этот момент находящийся там узел пуст, и мы помещаем в него точку *B*. Из точки *B* исходят две пустые ветви, которые можно будет впоследствии использовать для размещения новых точек. Чтобы вставить точку *C*, мы снова начинаем

с корня. Поскольку координата *X* точки *C* больше, чем у точки *A*, мы идем от корня по правой ветви, которая пока заканчивается пустым узлом, и помещаем в него точку *C*. На этом рисунке через точку *C* проходит еще одна горизонтальная прямая, делящая плоскость на верхнюю и нижнюю части. Этот процесс повторяется, пока все точки не окажутся в дереве.

Таблица 5.1. Точки, используемые для индексирования пространственных данных

Точка	*X*	*Y*
A	2	2
B	0	5
C	8	0
D	9	8
E	7	14
F	13	12
G	14	13

Из приведенного выше описания ясно, что использование координат *X* и *Y* чередуется. В литературе принято соглашение – сначала использовать координату *X*, потом *Y*, затем снова *X* и т. д. Левая часть рис. 5.1 служит только для иллюстрации. Во время поиска в дереве производится сравнение с соответствующей координатой на каждом уровне дерева, причем в корне сравнивается координата *X*.

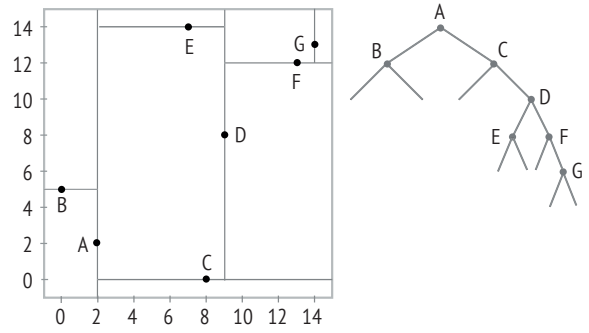

Рис. 5.1 ❖ 2D-дерево для множества семи точек, построенное в фиксированном порядке

Допустим, что требуется найти информацию о точке *F* на карте, где изображено семь точек, и что для этого мы щелкаем мышью по интересующей нас точке. Для поиска информации мы начинаем с корня *k*D-дерева. Поскольку координата *X* точки *F* больше, чем у корня, мы идем по правой ветви дерева, где находится точка *C*. Это не то, что мы ищем, поэтому мы продолжаем поиск и на этот раз сравниваем координаты *Y* искомой точки и точки *C*. Мы идем по правой ветви, потому что координата *Y* точки *F* больше, чем у точки *C*. Это приводит нас в узел, содержащий точку *D*, и теперь для определения

направления поиска мы сравниваем координаты *X*. Поиск продолжается по правой ветви и завершается в узле дерева *F*. Если считать, что глубина корня равна 0, то получается, что мы сравнивали координаты *X* на четных уровнях и координату *Y* на нечетных. На протяжении всего процесса мы на каждом шаге могли исключить потенциально половину ветвей, тем самым сделав поиск эффективным.

Код в листинге 5.1 реализует некоторые из рассмотренных выше идей[1]. Здесь мы впервые определили структуру данных, содержащую необходимую информацию об узле: хранящуюся в нем точку и левый и правый дочерние узлы. Функция `kdcompare` (строка 13) получает корень дерева, искомую точку, а также глубину, определяющую, какую координату использовать, и сообщает, по какой ветви следовать дальше.

Для вставки узла в дерево нам нужна функция, которая ищет в дереве путь к месту, в которое должна быть помещена новая точка. Это делает функция `query_kdtree` (строка 61). Она похожа на функцию поиска в двоичном дереве, рассмотренную в предыдущей главе, и находит узел, который должен стать родителем нового узла. Функция обходит дерево, вызывая в каждом узле `kdcompare`. Булев параметр `is_find_only` показывает, хотим ли мы найти пустой узел, в котором можно сохранить новую точку (`True`), или только узнать, существует ли в дереве точно такая же точка, как искомая (`False`). Чтобы найти пустой узел, функция проверяет, пуст ли дочерний узел текущего узла (строка 71). Если да, она возвращает текущий узел и ветвь, в которой следует сохранить новую точку (строка 73). В противном случае, если наша цель – воспользоваться этой функцией для поиска точки в дереве, то она возвращает узел, содержащий искомую точку, если таковой существует, и `None` в противном случае (строка 75).

Определив эти функции, мы теперь опишем, как создать *k*D-дерево по фиксированной последовательности точек. Для этого служит функция `kdtree` (строка 30). Сначала мы создаем экземпляр класса `KDTreeNode`, который станет корнем дерева (строка 34). Затем для каждой входной точки создается новый узел, содержащий эту точку (строка 36). Далее находим узел с пустым дочерним узлом (строка 37) и размещаем его по правильную сторону от родительского узла (строк 38–41). Созданное таким образом дерево показано на рис. 5.1. С первого взгляда видно, что правая ветвь дерева гораздо выше левой. Из-за этого дерево не сбалансировано, что, в свою очередь, увеличивает время поиска, т. к. следование вдоль правой ветви этого *k*D-дерева исключает из просмотра мало точек.

Несбалансированность *k*D-дерева обусловлена тем, что порядок точек в последовательности неудачен. Чтобы решить эту проблему, мы могли бы вставлять точки в другом порядке, а именно так, что всякий раз в дерево помещается средняя из оставшихся точек, подлежащих вставке. Именно так поступает функция `kdtree2` (строка 45), которая перед каждой вставкой в де-

---

[1] Частично этот код заимствован со страницы http://en.wikipedia.org/wiki/K-d_tree, он используется и в других местах, например https://code.google.com/p/python-kdtree/. Более полный Python-пакет для *k*D-деревьев можно найти в библиотеке SciPy (http://www.scipy.org).

рево выбирает из последовательности медианную точку. Сначала мы определяем, какая ось соответствует текущей глубине (строка 49). А затем сортируем точки по этой оси (строка 50) и находим медианную точку (строка 51). Далее функция вызывается рекурсивно (строка 55), при этом медианная точка используется в качестве нового узла, а остальные точки разбиваются на два множества, так что точки с меньшими координатами, чем у медианной, попадут в левую ветвь текущего узла, а с большими – в правую ветвь. Процесс продолжается, пока не кончатся точки (строки 46–47). Применив этот алгоритм к нашим семи точкам, мы создадим идеально сбалансированное 2D-дерево (рис. 5.2).

Функция `query` в листинге 5.1 только ищет точку, присутствующую (или отсутствующую) в дереве. На практике к дереву можно предъявить гораздо больше запросов. Например, мы могли бы нарисовать на экране окружность или прямоугольник и спросить, какие точки попадают в него. Такие запросы часто называют запросами к (круговому или прямоугольному) диапазону. Интересно также найти несколько точек, ближайших к данной точке, в качестве которой может выступать наше местонахождение или точка, указанная нами на карте.

Листинг 5.1 ❖ Реализация *k*D-дерева (kdtree1.py)

```
 1 import sys
 2 sys.path.append('../geom')
 3 from point import *
 4
 5 class kDTreeNode():
 6   def __init__(self, point, left, right):
 7       self.point = point
 8       self.left = left
 9       self.right = right
10   def __repr__(self):
11       return str(self.point)
12
13 def kdcompare(r, p, depth):
14     """
15     Возвращает результат поиска в kd-дереве
16     Вход
17       r: корень
18       p: точка
19       depth : начальная глубина поиска
20     Выход
21       -1 (идти по левой ветви) или 1 (идти по правой ветви)
22     """
23     k = len(p)
24     dim = depth%k
25     if p[dim] <= r.point[dim]:
26         return -1
27     else:
28         return 1
29
```

```
def kdtree(points):
    """
    Создает kd-дерево, содержащее точки в заданном порядке
    """
    root = kDTreeNode(point=points[0], left=None, right=None)
    for p in points[1:]:
        node = kDTreeNode(point=p, left=None, right=None)
        p0, lr = query_kdtree(root, p, 0, False)
        if lr<0:
            p0.left = node
        else:
            p0.right = node
    return root

# всегда использует медианную точку для разбиения данных на две части
def kdtree2(points, depth = 0):
    if len(points)==0:
        return
    k = len(points[0])
    axis = depth % k
    points.sort(key=lambda points: points[axis])
    pivot = len(points)//2
    while pivot<len(points)-1 and\
            points[pivot][axis]==points[pivot+1][axis]:
        pivot += 1
    return kDTreeNode(point=points[pivot],
                      left=kdtree2(points[:pivot],
                                   depth+1),
                      right=kdtree2(points[pivot+1:],
                                    depth+1))

def query_kdtree(t, p, depth=0, is_find_only=True):
    if t is None:
        return
    if t.point == p and is_find_only:
        return t
    lr = kdcompare(t, p, depth)
    if lr<0:
        child = t.left
    else:
        child = t.right
    if child is None:
        if not is_find_only:
            return t, lr
        else:
            return
    return query_kdtree(child, p, depth+1, is_find_only)

if __name__ == '__main__':
    data1 = [ (2,2), (0,5), (8,0), (9,8),
              (7,14), (13,12), (14,13) ]
    points = [Point(d[0], d[1]) for d in data1]
```

```
82      p = points[0]
83      t1 = kdtree(points)
84      t2 = kdtree2(points)
85
86      print [ query_kdtree(t1, p) for p in points ]
87      print [ query_kdtree(t2, p) for p in points ]
```

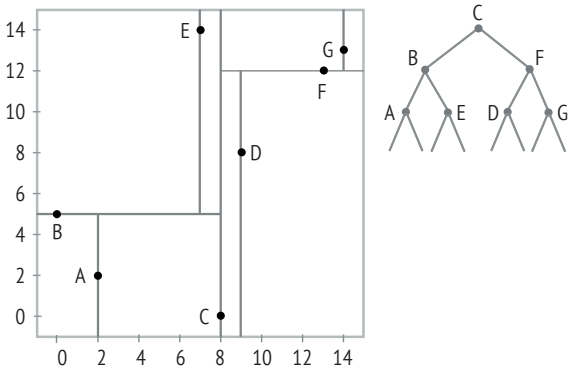

Рис. 5.2 ❖ 2D-дерево, построенное по семи точкам с использованием медианной точки на каждом уровне

5.1.1. Запрос к прямоугольному диапазону

Алгоритм запроса к прямоугольному диапазону ищет точки, попадающие в указанный пользователем прямоугольник. Он должен найти все точки, удовлетворяющие заданным требованиям. Для этого, возможно, придется обойти все дерево. Однако мы хотели бы все же не задерживаться на точках, которые заведомо не могут попасть в диапазон. В примере на рис. 5.3 мы хотим найти только точку *C*, находящуюся внутри пунктирного прямоугольника. Ясно, что все точки слева от корня (точки *A*) рассматривать не имеет смысла, потому что они лежат вне прямоугольника. Но точки справа от корня проверять нужно. Это наблюдение можно распространить на узлы на любом уровне. Например, находясь в узле *G*, мы можем игнорировать все левое поддерево, потому что у принадлежащих ему точек координата *Y* заведомо меньше, чем у точки с наименьшей координатой *Y* внутри прямоугольника. Но поддерево по правую сторону от узла *G* рассматривать нужно, хотя в нашем случае единственная принадлежащая ему точка (*I*) лежит вне прямоугольника. Вообще, если координата узла по просматриваемой в текущий момент оси меньше наименьшей границы прямоугольника по той же оси, то левое поддерево можно игнорировать, но правое необходимо просматривать. Если координата узла по текущей оси больше наибольшей границы прямоугольника по той же оси, то просматривать нужно только левое поддерево.

Этот алгоритм запроса к прямоугольному диапазону реализован функцией `range_query_orthogonal` в листинге 5.2. Для представления поискового прямоугольника нужно использовать двумерный список (строка 27), в котором

первый внутренний список содержит наименьшую и наибольшую координаты *X*, а второй – наименьшую и наибольшую координаты *Y*. В строке 8 точка в узле сравнивается с нижней границей прямоугольника. Явно проверять, находится ли точка внутри прямоугольника, нужно только после того, как все описанные выше условия уже проверены (строка 16). Мы используем прямоугольник, демонстрирующий случай в левой части рис. 5.4, в него попадают две точки:

```
[(2, 2), (9, 8)]
```

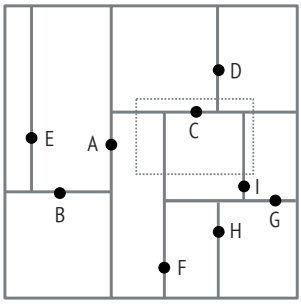

Рис. 5.3 ❖ Запрос к прямоугольному диапазону (пунктирный прямоугольник) с использованием *k*D-дерева

Листинг 5.2 ❖ Запрос к прямоугольному диапазону с использованием *k*D-дерева (kdtree2a.py)

```
 1 from kdtree1 import *
 2
 3 def range_query_orthogonal(t, rect, found, depth=0):
 4     if t is None:
 5         return
 6     k = len(t.point)
 7     axis = depth%k
 8     if t.point[axis] < rect[axis][0]:
 9         range_query_orthogonal(t.right, rect, found, depth+1)
10         return
11     if t.point[axis] > rect[axis][1]:
12         range_query_orthogonal(t.left, rect, found, depth+1)
13         return
14     x, y = t.point.x, t.point.y
15     if not (rect[0][0]>x or rect[0][1]<x or
16             rect[1][0]>y or rect[1][1]<y):
17         found.append(t.point)
18     range_query_orthogonal(t.left, rect, found, depth+1)
19     range_query_orthogonal(t.right, rect, found, depth+1)
20     return
21
22 def test():
23     data1 = [ (2,2), (0,5), (8,0), (9,8), (7,14),
24               (13,12), (14,13) ]
```

```
25      points = [Point(d[0], d[1]) for d in data1]
26      t1 = kdtree(points)
27      rect = [ [1, 9], [2, 9] ]
28      found = []
29      range_query_orthogonal(t1, rect, found)
30      print found
31
32 if __name__ == '__main__':
33      test()
```

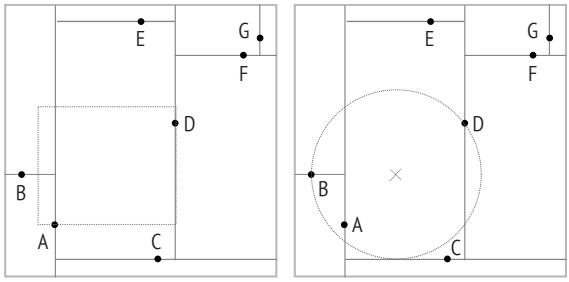

Рис. 5.4 ❖ Запрос к прямоугольному (слева) и круговому (справа) диапазонам с использованием *k*D-дерева, построенного по семи точкам

5.1.2. Запрос к круговому диапазону

Обработка запросов к круговому диапазону следует той же логике, что для прямоугольных диапазонов, только на этот раз мы должны найти точки, лежащие внутри окружности радиуса *r* с центром в указанной точке *p* (листинг 5.3). В каждом узле алгоритм проверяет, нужно ли спускаться дальше по дереву, стремясь при этом избежать просмотра всего дерева. Это делается рекурсивно в функции `range_query_circular`. Если наименьшая (в направлении *X* или *Y* в зависимости от текущего узла дерева) граница окружности находится справа или сверху от точки, представленной узлом (строка 7), то нужно переходить к правому дочернему узлу (строка 8) и игнорировать левую ветвь, потому что принадлежащие ей точки заведомо не попадают внутрь окружности. Аналогично, если наибольшая (в направлении *X* или *Y*) граница находится слева от текущего узла, то нужно переходить к левому дочернему узлу (строки 10 и 11). Если неверно ни то, ни другое, значит, представленная узлом точка находится внутри диапазона (в направлении *X* или *Y*), и мы должны проверить, действительно ли она лежит внутри окружности (строка 14) и, если да, включить ее в состав результата (строка 14). Затем мы удостоверяемся, что все дочерние узлы проверены (строки 15 и 16). Процесс продолжается, пока не окажется, что все точки либо проверены (строки 5 и 6), либо могут быть проигнорированы (строки 9 и 12). В результате выполнения функции `test` (строка 29) будут возвращены три точки, находящиеся на расстоянии, не большем 5, от точки (5, 5):

```
[(2, 2), (0, 5), (9, 8)]
```

Листинг 5.3 ❖ Запрос к круговому диапазону с использованием *k*D-дерева (kdtree2b.py)

```
from kdtree1 import *

# поиск точек внутри окружности радиуса r с центром в точке p
def range_query_circular(t, p, r, found, depth=0):
    if t is None:
        return
    if kdcompare(t, Point(p.x-r, p.y-r), depth)>0:
        range_query_circular(t.right, p, r, found, depth+1)
        return
    if kdcompare(t, Point(p.x+r, p.y+r), depth)<0:
        range_query_circular(t.left, p, r, found, depth+1)
        return
    if p.distance(t.point) <= r:
        found.append(t.point)
    range_query_circular(t.left, p, r, found, depth+1)
    range_query_circular(t.right, p, r, found, depth+1)
    return

def test():
    data1 = [ (2,2), (0,5), (8,0), (9,8), (7,14),
              (13,12), (14,13) ]
    points = [Point(d[0], d[1]) for d in data1]
    p = Point(5,5)
    t1 = kdtree(points)
    found = []
    range_query_circular(t1, p, 5, found)
    print found

if __name__ == '__main__':
    test()
```

5.1.3. Поиск ближайших соседей

Для поиска *n* ближайших соседей заданной точки *p* мы следуем той же логике, что при запросе к диапазону: мы хотим игнорировать точки, которые заведомо не могут быть ближайшими соседями, но не хотим пропустить потенциальных кандидатов. Иными словами, требуется найти *n* ближайших соседей *p* и только их. Конкретно, в каждом узле, не исключенном из рассмотрения, мы проверяем, верно ли, что хранящаяся в этом узле точка ближе к *p*, чем уже найденные. Если да, то мы обновляем множество соседей, вставляя в него текущую точку. При вставке новой точки мы следим за тем, чтобы множество найденных соседей оставалось отсортированным, и храним максимальное расстояние (`maxdist`) между найденными соседями и *p*. Если *n* соседей еще не найдено, то максимальное расстояние равно бесконечности. Затем мы решаем, какой из дочерних узлов ближе к *p*, и организуем логику так, чтобы более близкий узел был рассмотрен как один из потенциальных ближайших соседей. Если расстояние между точкой в узле и *p* меньше найденного

к этому моменту максимального расстояния, то нужно проверить и другую ветвь, чтобы не пропустить точки в этой части пространства. В любой момент поиска, достигнув листового узла, мы проверяем, следует ли включить хранящуюся в нем точку в состав ближайших соседей. Процесс продолжается, пока остаются точки, подлежащие просмотру.

В листинге 5.4 приведена реализация этой идеи. Для поддержания списка ближайших точек мы определяем константу, принимаемую за бесконечность. Поскольку настоящей бесконечности в компьютере быть не может (максимальное представимое число очень велико, но конечно), мы используем для этой цели встроенное в Python очень большое число (строка 3). Функция `update_neighbors` поддерживает список найденных до сих пор ближайших точек, проверяя, нужно ли вставить в него новую точку `p0`. В списке `neighbors` вместе с самой точкой хранится ее расстояние до `p`. Мы пользуемся встроенной в Python функцией `enumerate`, которая перебирает элементы списка и на каждом шаге возвращает индекс элемента и сам элемент (строка 8). Дойдя до конца списка (строка 9), функция возвращает расстояние от последнего элемента до `p` (строка 10). В противном случае, если расстояние между новой точкой `p0` и `p` меньше, чем расстояние, хранящееся в одном из элементов `neighbors` (строка 11), то мы вставляем ее в список (строка 12). Если количество найденных соседей еще меньше n, то в качестве максимального расстояния возвращается бесконечность (строка 14). Иначе возвращается расстояние до n-го элемента `neighbors` (строка 15). Если список пуст, мы просто добавляем в него новую точку и возвращаем бесконечность (строки 16–17).

Основная логика поиска сосредоточена в функции `nnquery` (строка 19)[1]. Движение вниз по дереву прекращается, когда узел пуст (строка 21) или является листовым (строка 23). В случае листового узла мы проверяем, следует ли добавлять хранящуюся в нем точку в список ближайших соседей (строка 24). Спускаясь по дереву, мы следим за глубиной, это позволяет решить, какую ось проверять при выборе следующей ветви (строка 26). Чтобы понять, какое поддерево ближнее, а какое дальнее, мы проверяем координату по соответствующей оси (строки 28 и 30). Мы спускаемся по ближнему поддереву (строка 31), чтобы узнать, существуют ли точки, более близкие к `p`, чем уже найденные. Но это не значит, что мы полностью игнорируем дальнее поддерево. Эта ситуация иллюстрируется на рис. 5.4, где целевой точкой p является (5, 5). В корне ближним является правое поддерево, и алгоритм исследует все точки в нем. Однако левое поддерево, возможно, тоже заслуживает внимания. Чтобы определить, так ли это, мы проверяем, верно ли, что расстояние между корнем этого поддерева и целевой точкой меньше текущего максимального расстояния. Если да, то мы заходим и в дальнее поддерево (строка 33).

Листинг 5.4 ❖ Поиск ближайших соседей с использованием *k*D-дерева (kdtree3.py)

```
1 from kdtree1 import *
2
3 INF = float('inf')
4 maxdist = INF
```

[1] Этот код заимствован из модуля https://code.google.com/p/python-kdtree/.

```
 5
 6 def update_neighbors(p0, p, neighbors, n):
 7     d = p0.distance(p)
 8     for i, x in enumerate(neighbors):
 9         if i == n:
10             return neighbors[n-1][1]
11         if d < x[1]:
12             neighbors.insert(i, [p0, d])
13         if len(neighbors) < n:
14             return INF
15         return neighbors[n-1][1]
16     neighbors.append([p0, d]) # первая точка
17     return INF
18
19 def nnquery(t, p, n, found, depth=0):
20     global maxdist
21     if t is None:
22         return
23     if t.left == None and t.right == None:
24         maxdist = update_neighbors(t.point, p, found, n)
25         return
26     axis = depth % len(p)
27     if p[axis] < t.point[axis]:
28         nearer_tree, farther_tree = t.left, t.right
29     else:
30         nearer_tree, farther_tree = t.right, t.left
31     nnquery(nearer_tree, p, n, found, depth+1)
32     maxdist = update_neighbors(t.point, p, found, n)
33     if (t.point.distance(p)) < maxdist:
34         nnquery(farther_tree, p, n, found, depth+1)
35     return
36
37 def kdtree_nearest_neighbor_query(t, p, n=1):
38     nearest_neighbors = []
39     nnquery(t, p, n, nearest_neighbors)
40     return nearest_neighbors[:n]
41
42 if __name__ == '__main__':
43     data1 = [ (2,2), (0,5), (8,0), (9,8), (7,14),
44               (13,12), (14,13) ]
45     points = [Point(d[0], d[1]) for d in data1]
46     p = Point(5,5)
47     t1 = kdtree(points)
48     n = 3
49     nearests = []
50     nnquery(t1, p, n, nearests)
51     print [x[0] for x in nearests[:n]]
52     print [x[1] for x in nearests[:n]]
53     print sorted([p.distance(x) for x in points])[:n]
```

Чтобы воспользоваться функцией `nnquery`, нам нужно подготовить пустой список (строка 49). Однако наши функции не предполагают ограничений на

общее число найденных элементов. Поэтому число точек в списке `nearests` может быть больше *n*. Мы пользуемся встроенной в Python операцией `slice`, чтобы вернуть не более *n* элементов (строки 51–53). В последней строке в листинге производится выборка из отсортированного списка всех расстояний. Из приведенных ниже результатов следует, что описанный алгоритм поиска ближайших соседей возвращает в точности то же самое, что поиск полным перебором.

```
[(2, 2), (9, 8), (0, 5), (8, 0)]
[4.242640687119285, 5.0, 5.0, 5.830951894845301]
[4.242640687119285, 5.0, 5.0, 5.830951894845301]
```

5.2. Точечно-регионные kD-деревья

Описанное в предыдущем разделе *k*D-дерево основано на разбиении пространства координатами точек из набора данных. В связи с этим возникает проблема поддержания правильной структуры дерева. Любое изменение в составе точек приводит к изменению структуры независимо от того, как было создано дерево. Например, дерево, основанное на разбиении с помощью медианных точек, придется корректировать после удаления точки или вставки новой точки. Точечно-регионное *k*D-дерево (point region – PR) решает эту проблему, заимствуя идею разбиения пространства у квадродеревьев, еще одного семейства методов индексирования пространственных данных (глава 6). В точечно-регионном *k*D-дереве мы по-прежнему пользуемся координатами *X* и *Y* при построении дерева. Но вместо того чтобы использовать координаты точки при каждом разбиении пространства, мы используем среднюю точку региона. Поэтому точечно-регионные *k*D-деревья отличаются от обычных *k*D-деревьев в одном важном отношении: данные хранятся только в листовых узлах (рис. 5.5 справа, ср. с рис. 5.1).

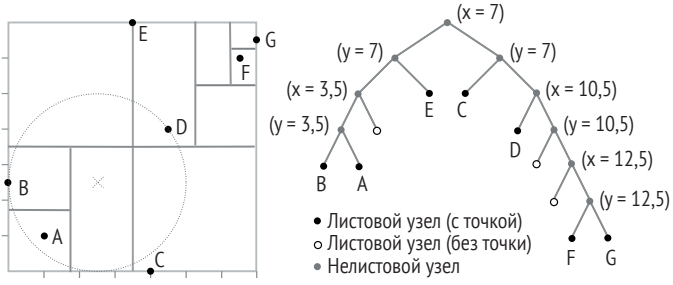

Рис. 5.5 ❖ Точечно-регионное *k*D-дерево для множества семи точек

На рис. 5.5 показано такое дерево, построенное по семи точкам. Как и раньше, дерево создается рекурсивно, начиная с корня, который определяется центром координат *X*. Конечно, для этого нам нужно знать прямоугольник, содержащий область, и здесь мы можем просто использовать ограничиваю-

щий прямоугольник всех точек. В листинге 5.5 приведен код, реализующий точечно-регионное *k*D-дерево. Ключ к построению такого дерева – явное запоминание информации о каждом регионе. Здесь мы приняли простой подход – хранение нижней и верхней границ, или диапазонов координат *X* и *Y* точек (строка 8). В строках 81–90 показано, как получаются и используются эти диапазоны. Мы также храним центр области (строка 9). Благодаря использованию центра легко определить, по какой ветви дерева должен продолжаться поиск; это делается в функции `prkdcompare` (строка 22).

Построение точечно-регионного *k*D следует той же логике, что и построение обычного *k*D-дерева с использованием медианы. Но вместо медианной точки, которая является элементом входных данных, здесь мы используем центр региона, т. е. точку, которая может и не присутствовать среди данных (а если и присутствует, мы все равно используем в качестве центра отдельную точку, а совпадающую с ней точку из набора данных храним в листовом узле). Функция `prkdtree` (начинающаяся в строке 27) строит точечно-регионное *k*D-дерево рекурсивно. Листовой узел создается, когда в рассматриваемом регионе есть всего одна точка или нет ни одной. В противном случае алгоритм продолжает разбивать регион на две части, пользуясь координатой *X* или *Y* его центральной точки (строки 41 и 42). Точки сортируются по соответствующей координате (строка 43), а для нахождения среднего значения, делящего множество точек на две половины, мы используем Python-модуль `bisect` (строки 44 и 45). Затем алгоритм должен вычислить границы и центры обоих подрегионов (строки 46–54), которые будут использованы для рекурсивного построения левой и правой ветвей текущего узла (строка 55).

Функция `query_prkdtree` ищет точку в точечно-регионном *k*D-дереве, следуя той же логике, что и в обычном *k*D-дереве, т. е. спускаясь по дереву (строки 71–75), пока не встретит пустой узел (строка 65), искомую точку (строка 67) или пустой листовой узел (строка 69). Для тестирования алгоритма мы построили простое точечно-регионное *k*D-дерево из семи точек и проверили, что все точки удается найти (строка 92).

Листинг 5.5 ❖ Создание и опрос точечно-регионного *k*D-дерева (prkdtree1.py)

```
 1 from bisect import *
 2 import sys
 3 sys.path.append('../geom')
 4 from point import *
 5
 6 class PRKDTreeNode():
 7     def __init__(self, xyrange, center, point, left, right):
 8         self.xyrange = xyrange       # диапазоны xy
 9         self.center = center         # координата центра
10         self.point = point           # точка
11         self.left = left             # левое поддерево
12         self.right = right           # правое поддерево
13     def is_leaf(self):
14         return (self.left == None and self.right == None)
15     def __repr__(self):
```

```
16          return str(self.center)
17
18  # r - корень поддерева, p - точка, depth - текущая глубина
19  def prkdcompare(r, p, depth):
20      k = len(p)
21      dim = depth%k
22      if p[dim] <= r.center:
23          return dim,-1               # налево
24      else:
25          return dim,1                # направо
26
27  def prkdtree(points, xylim, center=None, depth=0):
28      if len(points)==1:
29          return PRKDTreeNode(xyrange=xylim,
30                              center=center,
31                              point=points[0],
32                              left=None,
33                              right=None)
34      if len(points)==0:
35          return PRKDTreeNode(xyrange=xylim,
36                              center=center,
37                              point=None,
38                              left=None,
39                              right=None)
40      k = len(points[0])
41      axis = depth % k
42      pmid = (xylim[axis][1]+xylim[axis][0])/2.0
43      points.sort(key=lambda points:points[axis])
44      P = [p[axis] for p in points]
45      pivot = bisect(P, pmid) # includes all <=
46      xmin,xmax=xylim[0][0], xylim[0][1]
47      ymin,ymax=xylim[1][0], xylim[1][1]
48      rangel = [ [xmin, xmax], [ymin, ymax] ]
49      rangel[axis][1] = pmid
50      ranger = [ [xmin, xmax], [ymin, ymax] ]
51      ranger[axis][0] = pmid
52      axis2 = (depth+1) % k
53      pmidl = (rangel[axis2][0]+rangel[axis2][1])/2.0
54      pmidr = (ranger[axis2][0]+ranger[axis2][1])/2.0
55      return PRKDTreeNode(xyrange = xylim,
56                          center = pmid,
57                          point = None,
58                          left=prkdtree(points[:pivot],
59                                        rangel,pmidl,depth+1),
60                          right=prkdtree(points[pivot:],
61                                         ranger,pmidr,depth+1))
62
63  # запрос о нахождении p в t
64  def query_prkdtree(t, p, depth=0):
65      if t is None:
```

```
66          return
67      if t.point == p:
68          return p
69      if t.is_leaf():
70          return
71      if prkdcompare(t, p, depth)[1] < 0:
72          next = t.left
73      else:
74          next = t.right
75      p0 = query_prkdtree(next, p, depth+1)
76      if p0 is not None:
77          return p0
78      return
79
80  if __name__ == '__main__':
81      data1 = [ (2,2), (0,5), (8,0), (9,8), (7,14),
82                (13,12), (14,13) ]
83      points = [Point(d[0], d[1]) for d in data1]
84      px = [p.x for p in points]
85      py = [p.y for p in points]
86      xmin = min(px)
87      xmax = max(px)
88      ymin = min(py)
89      ymax = max(py)
90      xylim = [ [xmin, xmax], [ymin, ymax] ]
91      t = prkdtree(points, xylim)
92      print [query_prkdtree(t, p) for p in points]
```

Как и для *k*D-дерева, мы можем реализовать запросы к диапазону и поиск ближайших соседей в точечно-регионном *k*D-дереве. Здесь мы рассмотрим только запрос к круговому диапазону (листинг 5.6), а запрос к прямоугольному диапазону реализуется аналогично. Основной функцией в алгоритме запроса к диапазону является `rquery` (строка 5). Это рекурсивная функция, которая прекращает работу, встретив пустой узел. В каждом узле мы можем получить ширину и высоту представленного им региона (строки 8 и 9). Если левая (или нижняя) граница этого региона находится правее (или выше) текущего узла дерева (строка 11), то нужно рассматривать только правое поддерево узла, поскольку точки в левом поддереве заведомо не попадают в запрошенный диапазон. Аналогично, если правая (или верхняя) граница региона находится левее (или ниже) текущего узла дерева (строка 15), то нужно рассматривать только левое поддерево узла. В остальных случаях мы должны проверять, попадает ли представленная узлом точка в радиус поиска (строка 18), и рассматривать обе ветви узла (строки 21 и 22). Мы видим, что функция `rquery` вложена в функцию `range_query` (строка 4), смысл этого в том, чтобы сделать список точек, найденных в результате поиска по дереву, доступным непосредственно, а не передавать его каждый раз в качестве параметра функции выполнения запроса. Алгоритм тестируется на тех же данных, что для *k*D-дерева, и результат получается тот же самый:

```
[(2, 2), (0, 5), (9, 8)]
```

Листинг 5.6 ❖ Запрос к круговому диапазону с использованием точечно-регионного *k*D-дерева (prkdtree2.py)

```
 1 from prkdtree1 import *
 2
 3 # поиск точек в окружности радиуса r с центром в p
 4 def range_query(t, p, r):
 5     def rquery(t, p, r, found, depth=0):
 6         if t is None:
 7             return
 8         w = t.xyrange[0][1] - t.xyrange[0][0]
 9         h = t.xyrange[1][1] - t.xyrange[1][0]
10         if prkdcompare(t, Point(p.x-r-w, p.y-r-h),
11                        depth)[1] > 0:
12             rquery(t.right, p, r, found, depth+1)
13             return
14         if prkdcompare(t, Point(p.x+r+w, p.y+r+h),
15                        depth)[1] < 0:
16             rquery(t.left, p, r, found, depth+1)
17             return
18         if t.point is not None:
19             if p.distance(t.point) <= r:
20                 found.append(t.point)
21         rquery(t.left, p, r, found, depth+1)
22         rquery(t.right, p, r, found, depth+1)
23         return
24     found = []
25     if t is not None:
26         rquery(t, p, r, found)
27     return found
28
29 if __name__ == '__main__':
30     data1 = [ (2,2), (0,5), (8,0), (9,8), (7,14),
31               (13,12), (14,13) ]
32     points = [Point(d[0], d[1]) for d in data1]
33     px = [p.x for p in points]
34     py = [p.y for p in points]
35     xmin = min(px)
36     xmax = max(px)
37     ymin = min(py)
38     ymax = max(py)
39     xylim = [ [xmin, xmax], [ymin, ymax] ]
40     t = prkdtree(points, xylim)
41     p = Point(5,5)
42     print range_query(t, p, 5)
```

Алгоритм нахождения ближайших соседей заданной точки реализован функцией `prkdtree_nnquery` в листинге 5.7. Он очень похож на алгоритм для *k*D-дерева, но имеет два существенных отличия. Во-первых, в строке 10 нужно проверять, представляет ли узел точку, потому что не во всех листовых узлах (и вообще не во всех узлах) точечно-регионного *k*D-дерева хранятся точки. С другой стороны, в обычном *k*D-дереве *любой* узел соответствует точ-

ке исходного набора данных. В точечно-регионном же *k*D-дереве это не так, потому узел соответствует центру региона, а не точке данных. Во-вторых, строка 19 закомментирована, потому что список найденных точек обновляется только в листовых узлах. Функция `nearest_neighbor_query` обертывает основную функцию поиска и возвращает список найденных точек. Затем алгоритм тестируется на том же наборе данных и находит три ближайшие к (5, 5) точки:

```
[(2, 2), (9, 8), (0, 5)]
[4.242640687119285, 5.0, 5.0]
[4.242640687119285, 5.0, 5.0]
```

Листинг 5.7 ❖ Поиск ближайших соседей с использованием точечно-регионного *k*D-дерева (prkdtree3.py)

```
 1 ffrom prkdtree1 import *
 2 from kdtree3 import *               # используем update_neighbors и INF
 3
 4 prmaxdist = INF
 5
 6 def prkdtree_nnquery(t, p, n, found, depth=0):
 7     global prmaxdist
 8     if t is None:
 9         return
10     if t.is_leaf() and t.point is not None:
11         prmaxdist = update_neighbors(t.point, p, found, n)
12         return
13     axis,dir = prkdcompare(t, p, depth)
14     if dir<0:
15         nearer_tree, farther_tree = t.left, t.right
16     else:
17         nearer_tree, farther_tree = t.right, t.left
18     prkdtree_nnquery(nearer_tree, p, n, found, depth+1)
19     #prmaxdist = update_neighbors(t.point, p, found, n)
20     if (t.center-p[axis]) < prmaxdist:
21         prkdtree_nnquery(farther_tree, p, n, found, depth+1)
22     return
23
24 def nearest_neighbor_query(t, p, n=1):
25     nearest_neighbors = []
26     prkdtree_nnquery(t, p, n, nearest_neighbors)
27     return nearest_neighbors[:n]
28
29 if __name__ == '__main__':
30     data1 = [ (2,2), (0,5), (8,0), (9,8), (7,14),
31               (13,12), (14,13) ]
32     points = [Point(d[0], d[1]) for d in data1]
33     px = [p.x for p in points]
34     py = [p.y for p in points]
35     xmin = min(px)
36     xmax = max(px)
37     ymin = min(py)
```

```
    ymax = max(py)
    xylim = [ [xmin, xmax], [ymin, ymax] ]
    t = prkdtree(points, xylim)
    n = 3
    p = Point(5,5)
    nearests = nearest_neighbor_query(t, p, n)
    print [x[0] for x in nearests[:n]]
    print [x[1] for x in nearests[:n]]
    print sorted([p.distance(x) for x in points])[:n]
```

5.3. Тестирование *k*D-деревьев

Насколько эффективно индексирование пространственных данных с помощью *k*D-дерева или регионного *k*D-дерева? Проверим на наборах данных разного размера. В листинге 5.8 мы используем набор случайных точек и вычисляем время построения дерева и поиска 100 случайных точек в нем с помощью дерева и прямым перебором.

Листинг 5.8 ❖ Тесты производительности *k*D-деревьев (kdtree_performance.py)

```
from kdtree1 import *
from prkdtree1 import *

import random
import time
import sys
import string

if __name__ == '__main__':
    npts = 1000                     # число точек по умолчанию
    if len(sys.argv)==2:            # заданное пользователем число точек
        if sys.argv[1].isdigit() is True:
            npts = string.atoi(sys.argv[1])
    points = []
    for i in xrange(npts):
        p = Point(random.random(), random.random())
        points.append(p)

    time1 = time.time()
    kdt1 = kdtree(points)
    time2 = time.time()
    treet1 = time2-time1

    time1 = time.time()
    kdt2 = kdtree2(points)
    time2 = time.time()
    treet2 = time2-time1

    px = [p.x for p in points]
    py = [p.y for p in points]
```

```
    xmin = min(px)
    xmax = max(px)
    ymin = min(py)
    ymax = max(py)
    xylim = [ [xmin, xmax], [ymin, ymax] ]
    time1 = time.time()
    kdt3 = prkdtree(points, xylim)
    time2 = time.time()
    treet3 = time2-time1

    print npts, "|", treet1, treet2, treet3, "|",

    n = 100
    t1 = 0                          # время поиска 100 точек в kd-дереве
    t2 = 0                          # время поиска в сбалансированном kd-дереве
    t3 = 0                          # время поиска в точечно-регионном kd-дереве
    t4 = 0                          # время линейного поиска
    pp = random.sample(points, n)
    for p in pp:
        time1 = time.time()
        p1 = query_kdtree(kdt1, p)
        time2 = time.time()
        t1 = t1 + time2-time1

        time1 = time.time()
        p1 = query_kdtree(kdt2, p)
        time2 = time.time()
        t2 = t2 + time2-time1

        time1 = time.time()
        p1 = query_prkdtree(kdt3, p)
        time2 = time.time()
        t3 = t3 + time2-time1

        time1 = time.time()
        for i in range(len(points)):
            if p == points[i]:
                break
        time2 = time.time()
      t4 = t4 + time2-time1

print t1, t2, t3, t4
```

Мы проверяем точки в диапазоне от 10 000 до 1 млн, пользуясь обычными и точечно-регионными *k*D-деревьями, описанными в этой главе. Результаты приведены в табл. 5.2. Отметим, что абсолютное время (в секундах) не важно, поскольку зависит от конкретного компьютера[1] и языка программирования. Ясно, что время построения *k*D-дерева существенно больше, чем время поиска в нем. По мере увеличения количества точек в дереве время

[1] Мы использовали MacBook Air с процессором Intel Core i5 1.3 ГГц и памятью 4 ГБ.

построения возрастает (рис. 5.6). Что же касается запросов, то в левой части рис. 5.7 мы видим колоссальный эффект индексирования: для всех трех методов индексирования время запроса близко к нулю, тогда как в случае линейного поиска оно линейно возрастает с увеличением количества точек. В правой части рисунка показаны детальные графики для всех трех видов *k*D-деревьев.

Таблица 5.2. Производительность kD-деревьев

	Построение дерева			**Запрос**			
N	***k*D**	***k*D***	**PR *k*D**	***k*D**	***k*D***	**PR *k*D**	**Линейный**
10 000	0.43	0.16	0.56	0.005	0.003	0.009	0.495
20 000	0.94	0.47	1.33	0.006	0.004	0.011	1.075
30 000	2.02	0.70	2.42	0.008	0.005	0.014	2.025
40 000	2.49	0.92	3.21	0.007	0.004	0.011	1.978
50 000	2.96	1.05	3.56	0.007	0.004	0.011	2.550
60 000	3.19	1.27	4.56	0.008	0.004	0.012	3.300
70 000	3.88	1.58	5.52	0.007	0.004	0.012	3.458
80 000	4.96	2.37	8.46	0.010	0.006	0.015	5.032
90 000	4.89	2.03	6.78	0.007	0.004	0.011	4.452
100 000	5.19	2.29	8.10	0.007	0.004	0.011	4.931
200 000	11.20	5.40	16.63	0.008	0.005	0.012	10.553
300 000	17.95	8.64	25.37	0.010	0.006	0.015	13.693
400 000	25.27	12.43	37.33	0.011	0.006	0.015	22.594
500000	33.10	16.38	47.60	0.011	0.007	0.017	27.388
600 000	38.14	21.29	59.59	0.011	0.007	0.016	31.133
700 000	44.42	24.22	65.11	0.012	0.007	0.019	38.304
800 000	52.41	28.89	79.16	0.013	0.008	0.026	44.704
900 000	59.60	33.12	98.97	0.013	0.007	0.019	50.814
1 000 000	71.01	39.90	110.65	0.016	0.009	0.024	57.869

* Сбалансированное.

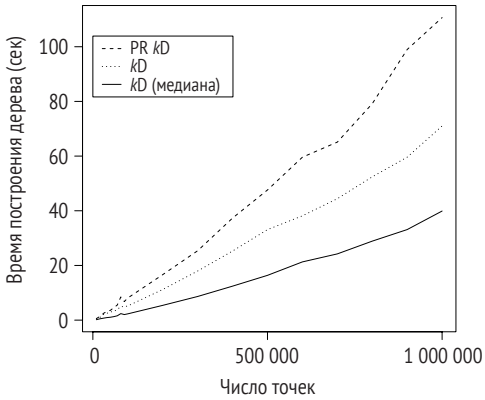

Рис. 5.6 ❖ Время построения *k*D-дерева

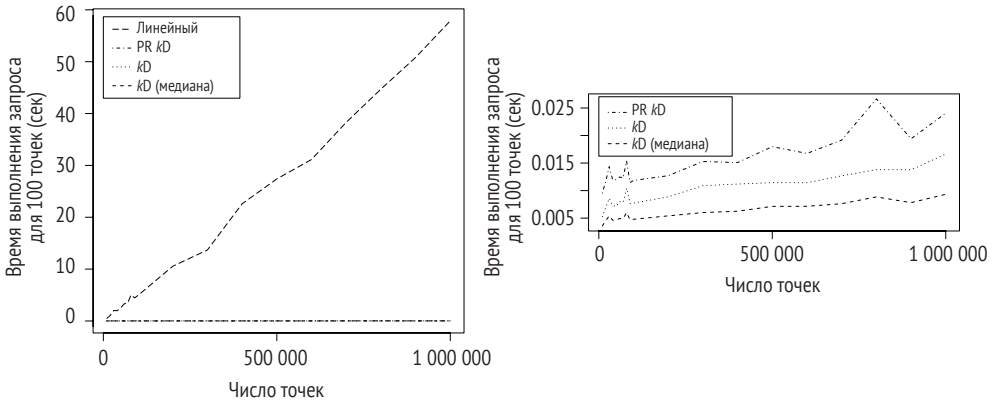

Рис. 5.7 ❖ Время выполнения запроса

Однако, сравнивая время запроса со временем построения дерева, мы можем задаться вопросом: а что толку от использования *k*D-деревьев, если время построения даже больше, чем время линейного поиска? Особенно это заметно, когда число точек велико. Следует отчетливо понимать, что индексирование пространственных данных не предназначено для разовых запросов. Например, вряд ли стоит этим заниматься, если нам нужно только один раз найти точку среди данных и больше мы никогда к этим данным не вернемся. Но существует много реальных приложений, в которых одни и те же данные используются снова и снова и, возможно, миллионами пользователей. Типичный пример – онлайновые карты; только представьте себе, сколько человек ежедневно используют Google Maps, например, чтобы найти ближайший ресторан. В некоторых исследованиях, особенно связанных с использованием данных переписей (Young et al., 2009), также необходим интенсивный опрос набора данных для нахождения местоположения объектов. Кроме того, не будем забывать, что есть и другие задачи, помимо описанных здесь простых примеров. Например, запрос к диапазонам методом полного перебора будет значительно медленнее из-за необходимости вычислять расстояния – нетривиальной операции.

Если сосредоточиться на трех видах *k*D-деревьев, то из приведенных выше результатов с очевидностью следует, что сбалансированные *k*D-деревья, построенные по медианным точкам, превосходят остальные по производительности. Это и неудивительно: сбалансированное дерево всегда лучше. Но почему сбалансированное дерево и строится быстрее? Внимательно присмотревшись к коду, мы заметим, что на каждом шаге построения сбалансированного *k*D-дерева применяется алгоритм сортировки, встроенный в Python по умолчанию, тогда как для двух других деревьев приходится использовать уже построенное дерево для поиска нужного узла; но эти деревья не сбалансированы, поэтому не так эффективны, как быстрая сортировка.

Построение любого *k*D-дерева опирается на какой-то метод сортировки. В наших реализациях мы либо используем встроенный в Python метод сортировки точек по той или иной координате, либо пользуемся тем порядком точек, который уже задан перед построением дерева. Сортировку можно

выполнить эффективно, но на добавление точек в дерево приходится большая часть времени построения. Полная временная сложность построения *k*D-дерева равна $O(n \log n)$. Но время поиска в *k*D-дереве в среднем составляет всего $O(\log n)$. Однако в худшем случае, когда все точки выстроились вдоль одной ветви узла, время поиска составляет $O(n)$ – так же, как при линейном поиске.

Каков теоретический объем памяти, занятой *k*D-деревом? Это зависит от того, сколько узлов придется создать для построения всего дерева. На рис. 5.1 и 5.2 показано, что количество узлов в точности равно количеству точек. Поэтому объем памяти равен $O(n)$, где n – число точек. Но в случае точечно-регионного *k*D-дерева нужно больше узлов, поскольку следует учитывать нелистовые узлы и листовые узлы без данных. В лучшем случае, когда во всех листовых узлах хранятся данные и все они расположены на одном и том же уровне, мы имеем идеально сбалансированное дерево высоты $\ln n$. Тогда общее число узлов равно $n \ln n$.

5.4. Примечания

Бентли (Bentley, 1975) первому пришла в голову идея *k*D-дерева как структуры данных для поиска информации. С тех пор было опубликовано много работ на эту тему. Ханан Самет написал три великолепные книги по индексированию пространственных данных (Samet, 1990a,b, 2006). В них описаны не только классические *k*D-деревья, рассмотренные в этой главе, но и многочисленные варианты. С *k*D-деревьями связано много интересных вопросов, которые мы не осветили (см. упражнения ниже). Одна из важных идей – использование блоков (bucket). В рассмотренных выше *k*D-деревьях каждый узел (или регион в случае регионных *k*D-деревьев) содержит не более одной точки. Это удобно при опросе дерева, потому что в каждом узле нужно сравнивать максимум с одной точкой. Но одновременно растет размер (глубина) дерева. Для решения этой проблемы мы могли бы хранить в одном узле несколько точек. Это позволило бы существенно уменьшить размер дерева (а значит, и время его построения). Если размер блока не слишком велик, то поиск точек в блоке не сильно увеличит время поиска. Кроме того, использование блоков уменьшает размер дерева, а вместе с ним и общее время выполнения запроса.

5.5. Упражнения

1. Модифицируйте код всех трех алгоритмов запроса к *k*D-дереву, так чтобы он сообщал обо всех точках, проверенных в процессе поиска. Действительно ли алгоритмы игнорируют некоторые узлы и точки, как задумано?
2. Протестируйте производительность методов запроса к диапазону и поиска ближайших соседей для *k*D-деревьев. В листинге 5.8 имеется не-

обходимый код, которым можно воспользоваться для создания наборов случайных точек для такого тестирования.

3. Рассмотрев первую технику индексирования пространственных данных, уместно задаться вопросом, действительно ли индексирование повышает скорость работы реального приложения. Воспользуйтесь *k*D-деревом в одном из методов интерполяции (см. главу 8) и посмотрите, имеется ли значимое изменение производительности.
4. Мы обсудили запросы к прямоугольному диапазону с использованием *k*D-дерева. Точечно-регионное *k*D-дерево имеет такую же структуру, но в процессе поиска возникают некоторые нюансы. Сумеете ли вы реализовать алгоритм запроса к прямоугольному диапазону с использованием точечно-регионного *k*D-дерева?
5. Докажите, что сбалансированное двоичное дерево с n листовыми узлами содержит $n \ln n$ узлов.
6. Что будет, если расстояние (или вообще пространственная информация) является не единственным критерием при поиске ближайших точек? Например, если с каждой точкой ассоциирована общая численность населения и требуется найти ближайшие населенные пункты, в которых проживает более 100 человек. Модифицируйте код, так чтобы он возвращал правильные результаты.
7. Модифицируйте код в файле `kdtree1.py`, так чтобы в каждом узле можно было хранить не более двух точек. Как это отразится на общей производительности *k*D-дерева для наборов данных разного размера?
8. В главе 1 мы обсуждали алгоритм нахождения кратчайшего расстояния между парами точек из списка. Его время работы составляет $O(n^2)$, где n – количество точек в списке. Можно ли воспользоваться *k*D-деревом, чтобы существенно повысить эффективность решения этой задачи? Напишите программу на Python и подумайте о теоретической временной сложности своего кода. Сможете ли вы доказать свои выводы эмпирически?

Глава 6

Квадродеревья

В квадродереве (или дереве квадрантов) для индексирования геопространственных данных используются обе координаты *X* и *Y*. Эта стратегия отличается от *k*D-деревьев, где на каждом шаге используется только одно направление. Чтобы воспользоваться обеими координатами, узел квадродерева разбивает плоскость на четыре части, которые в зависимости от различных проектных соображений и правил могут быть разбиты еще на четыре меньшие части. Идею квадродеревьев можно применить не только к точкам, но и к геометрическим объектам большей размерности, например прямым и многоугольникам, и к другим моделям данных, например растровой. В литературе, посвященной ГИС, применение квадродеревьев к растровым данным, вероятно, исследовано наиболее полно, быть может, потому что растровая модель полезна в самых разных приложениях цифровых моделей рельефа и изображений дистанционного зондирования. В этой главе мы начнем разговор с регионных квадродеревьев, разработанных специально для растровых данных. Затем обсудим квадродеревья для точечных данных, или точечные квадродеревья. В следующей главе мы продолжим обсуждение квадродеревьев для прямолинейных отрезков и многоугольников.

6.1. Регионные квадродеревья

Если в растровых данных встречается однородная область, все элементы которой имеют одно и то же значение, то не имеет смысла хранить значение каждого элемента, создавая тем самым избыточность. Достаточно было бы сохранить одно значение, представляющее всю область. Именно эта идея лежит в основе регионных квадродеревьев: мы продолжаем разбивать область на четыре равные части, пока каждая часть не окажется однородной. В этом контексте каждая часть называется квадрантом, поскольку является одной четвертью большего региона, для которого и построена. В процессе разбиения региона мы создаем дерево. Корень дерева представляет весь регион до начала разбиения. Если весь регион однороден, то корень также является листовым узлом, и с ним ассоциируется значение элементов. В противном случае корень не является листом и содержит четыре ветви, ведущие к четырем новым узлам. Узел становится листовым, когда представляет однородный регион. С нелистовыми узлами ассоциируются еще четыре узла – и так до тех пор, пока мы не дойдем до листьев.

Проиллюстрируем идею регионного квадродерева на примере гипотетических растровых данных на сетке 16×16, изображенных на рис. 6.1A. Без ограничения общности будем считать, что возможных значений только два: 0 (белое) и 1 (серое). В реальном приложении значений может быть больше. Здесь область неоднородна и может быть разбита на меньшие части.

На первом шаге мы создаем нелистовой узел, являющийся корнем дерева (верхний узел на рис. 6.1) и имеющий четыре дочерних узла. Каждый из четырех дочерних узлов указывает на один из квадрантов на рис. 6.1B. Ради согласованности мы всегда нумеруем квадранты по часовой стрелке, начиная с северо-западного (правый верхний): на рис. 6.1 они обозначены NW, NE, SE, SW. Этот порядок подразумевается во всех квадродеревьях в данной книге. Поскольку ни один из квадрантов не однороден, каждый представляется нелистовым узлом, т. е. будет иметь еще четыре дочерних узла. Чтобы не разбрасываться, мы сосредоточимся только на северо-западном квадранте. Разбив этот квадрант на четыре части, мы замечаем, что квадранты NW, SE и SW теперь однородны. Поэтому они представлены листовыми узлами дерева. Однако квадрант NE нуждается в дальнейшем разбиении. Данный процесс продолжается, пока все части не станут однородными, в этот момент построение дерева завершается.

Программа в листинге 6.1 реализует идею регионного квадродерева. Прежде чем обратиться к коду, разберемся с хранением растровых данных. И на этот раз попытаемся применить простой и прямолинейный подход к представлению данных. В данном случае мы используем двумерный массив NumPy, как показано в строках 70 и 76, где во втором примере используются данные, изображенные на рис. 6.1. Того же результата можно было бы достичь с помощью списков Python. Однако NumPy предоставляет очень удобные инструменты для вырезания прямоугольных «блоков» двумерного массива (см. ниже). Это замечательный инструмент, который позволяет сосредоточиться на идее, а не на программных приемах, необходимых для решения задачи.

Для заданного двумерного массива функция `homogeneity` (строка 14) проверяет, все ли его элементы принимают одно и то же значение. Если да, то мы можем сказать, что процесс разбиения дошел до однородного квадранта и можно создавать листовой узел. Однородность проверяется в лоб: мы перебираем все элементы массива и возвращаем `False`, если находим значение, не равное значению первого элемента.

Мы определяем структуру данных `Quad` (строка 3) для хранения информации о каждом квадранте на каждом шаге разбиения. Квадрант также представляет собой узел дерева. В объекте `Quad` хранится три важных элемента информации: значение квадранта, если он однороден, размер квадранта и четыре его дочерних узла. Мы не помечаем явно, является узел листовым или нет, потому что это легко проверить, посмотрев, все ли его дочерние узлы пустые (равны `None` в Python). Нелистовой узел также будет иметь значение `None`, потому что он неоднороден (и, стало быть, ему нельзя присвоить определенное значение).

После этих подготовительных пояснений мы можем создать регионное квадродерево по заданному набору данных. Для этого служит функция

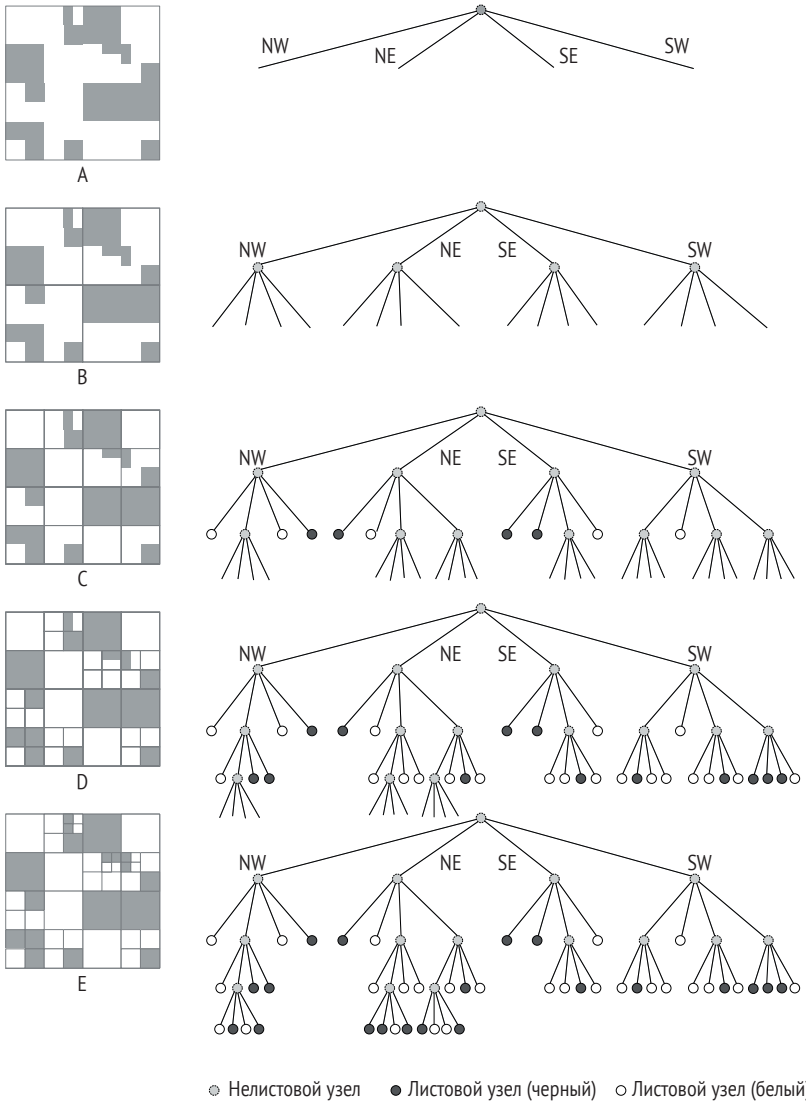

Рис. 6.1 ❖ Разбиение региона растровых данных на сетке 16×16 на квадранты. Дерево справа от каждого рисунка иллюстрирует построение регионного квадродерева на данном шаге разбиения. Внизу показано конечное дерево, ассоциированное с шагом (E)

`quadtree` (строка 22). Здесь мы сначала находим размер данных, переданных функции, с помощью метода `shape` массива `numpy`. Он возвращает число элементов по указанному измерению, мы с его помощью получим количество строк данных. В переменной `dim` хранится текущий размер, а в переменной `dim2` – размер дочерних узлов, который, в силу самой природы квадродерева, в два раза меньше текущего размера. Как следует из рис. 6.1 и обсуждения выше, данные, используемые для создания квадродерева, должны иметь одинаковое количество строк и столбцов. Кроме того, количество строк (а значит,

и столбцов) должно быть равно 2^n, где n – константа. Если текущие данные однородны (строка 25), то алгоритм останавливается и возвращает листовой узел в виде экземпляра класса `Quad`, все четыре дочерних узла которого равны `None`. В противном случае возвращается текущий узел без значения, и функция вызывает саму себя, чтобы создать четыре дочерних узла, данные для которых вырезаны из текущих данных. Например, квадрант NW в результате такого вырезания получает данные в строках от 0 до `dim2` и в столбцах от 0 до `dim2`, исключая саму строку и столбец `dim2` (строка 30). Рекурсия останавливается, когда все меньшие квадранты станут однородными.

Листинг 6.1 ❖ Регионное квадродерево (quadtree1.py)

```
 1 import numpy as np
 2
 3 class Quad():
 4     def __init__(self, value, dim, nw, ne, se, sw):
 5         self.value = value
 6         self.dim = dim
 7         self.nw = nw
 8         self.ne = ne
 9         self.se = se
10         self.sw = sw
11     def __repr__(self):
12         return str(self.value)
13
14 def homogeneity(d):
15     v = d[0,0]
16     for i in d:
17         for j in i:
18             if j <> v:
19                 return False
20     return True
21
22 def quadtree(data):
23     dim = data.shape[0]
24     dim2 = dim/2
25     if homogeneity(data) is True:
26         return Quad(value=data[0,0], dim=dim,
27                     nw=None, ne=None, se=None, sw=None)
28     return Quad(value=None,
29         dim = dim,
30         nw = quadtree(data[0:dim2, 0:dim2]),
31         ne = quadtree(data[0:dim2, dim2:,]),
32         se = quadtree(data[dim2:,dim2:,]),
33         sw = quadtree(data[dim2:,0:dim2]))
34
35 ##############################################################
36 # Примечание: x, y отсчитываются от левого верхнего угла
37 #
38 # Квадранты в дочерних узлах расположены в порядке
39 # [q.nw, q.ne, q.sw, q.se]. Ячейка, в которую попадает (x, y) в
40 # каждом регионе, определяется значениями dx и dy. Тогда квадрант,
```

```
# в котором расположена (x, y), вычисляется по формуле dx + dy * 2.
#
#         dx
#
#   ----+-----+------
#       |  0  |  1
#   ----+-----+------
# dy  0 |  0  |  1
#   ----+-----+------
#     1 |  3  |  2
#   ----+-----+------
#
#############################################################
def query_quadtree(q, x, y):
    if q.value is not None:       # это листовой узел
        return q.value            # вернуть значение
    dim = q.dim
    dim2 = dim/2
    dx,dy = 0, 0
    if x>=dim2:
        dx = 1
        x = x-dim2
    if y>=dim2:
        dy = 1
        y = y-dim2
    qnum = dx+dy*2
    return query_quadtree([q.nw,q.ne,q.sw,q.se][qnum], x, y)

def test():
    data0 = np.array(
        [[0, 1, 2, 2],
         [2, 2, 2, 2],
         [0, 0, 2, 0],
         [0, 0, 1, 0]])

    data1 = np.array(
        [[0,0,0,0,0,0,1,0,1,1,1,1,0,0,0,0],
         [0,0,0,0,0,0,1,0,1,1,1,1,0,0,0,0],
         [0,0,0,0,0,0,1,1,1,1,1,1,0,0,0,0],
         [0,0,0,0,0,0,1,1,1,1,1,1,0,0,0,0],
         [1,1,1,1,0,0,0,0,0,0,1,1,1,0,0,0],
         [1,1,1,1,0,0,0,0,0,0,0,0,1,0,0,0],
         [1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,1],
         [1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,1],
         [0,0,1,1,0,0,0,0,1,1,1,1,1,1,1,1],
         [0,0,1,1,0,0,0,0,1,1,1,1,1,1,1,1],
         [0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1],
         [0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1],
         [1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0],
         [1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0],
         [0,0,1,1,0,0,1,1,0,0,0,0,0,0,1,1],
         [0,0,1,1,0,0,1,1,0,0,0,0,0,0,1,1]])

```

```
94      q = quadtree(data1)
95      for y in range(q.dim):
96          for x in range(q.dim):
97              print query_quadtree(q, x, y),
98          print
99      return q
100
101 if __name__ == '__main__':
102      test()
```

Если задано регионное квадродерево, то мы можем запросить значение в столбце x и строке y с помощью функции query_quadtree (строка 54). Важно отметить, что нумерация строк и столбцов начинается в левом верхнем углу сетки. Задача в том, чтобы преобразовать точку, заданную значениями x и y, в правильный квадрант на каждом шаге. Преобразование необходимо, если используется другое начало координат. Чтобы выполнить его, мы при запросе к квадродереву корректируем номера строки и столбца на каждом уровне дерева, принимая во внимание уровень. Переменные dx и dy показывают, в какой части относительно центра региона находится запрашиваемая точка: dx равно 0 (1), если она находится в левой (правой) половине по горизонтали, а dy равно 0 (1), если в верхней (нижней) половине по вертикали. Комментарий в строках 35–53 подробно описывает, как определяются значения dx и dy. Если x находится в правой половине региона, то нужно модифицировать значение x для опроса следующего уровня дерева (строка 62), и то же самое относится к значению y. В строке 66 вычисляется номер квадранта (qnum), в который попадает запрашиваемая точка; он принимает значения от 0 до 3. Для продолжения поиска мы рекурсивно вызываем query_quadtree, передавая ей дочерний квадрант с индексом qnum из списка, в котором квадранты перечислены в правильном порядке (строка 67).

Функция test в листинге 6.1 использует данные, изображенные на рис. 6.1, для построения квадродерева. Затем мы в цикле вызываем query_quadtree, чтобы получить значения в каждой позиции сетки. Должны быть напечатаны в точности те значения, на основе которых строилось дерево.

```
0 0 0 0 0 0 1 0 1 1 1 1 0 0 0 0
0 0 0 0 0 0 1 0 1 1 1 1 0 0 0 0
0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0
0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0
1 1 1 1 0 0 0 0 0 0 1 1 1 0 0 0
1 1 1 1 0 0 0 0 0 0 0 0 1 0 0 0
1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1
1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1
0 0 1 1 0 0 0 0 1 1 1 1 1 1 1 1
0 0 1 1 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 0 0 1 1 0 0 0 0 0 0 1 1
0 0 1 1 0 0 1 1 0 0 0 0 0 0 1 1
```

6.2. Точечные квадродеревья

Идею квадродеревьев можно распространить на индексирование точек: пространство разбивается на квадранты до тех пор, пока не окажется, что в каждом квадранте не более одной точки. Возьмем в качестве примера рис. 6.2, где пространство разбивается только в местах расположения точек. Это именно то, что называется точечным квадродеревом; в данном случае используются те же семь точек, что в предыдущем разделе. Мы вставляем в дерево точки *A*, *B*, ..., *G*, помещая точку *A* в корень. Вертикальная и горизонтальная прямые, проходящие через *A*, разбивают пространство на четыре квадранта. Поскольку в северо-западный квадрант попадает только точка *B*, узел в ветви NW корня становится листовым и содержит точку *B*. Узел является листовым, если все четыре исходящие из него ветви пусты. Вставляя новую точку, мы находим подходящую ветвь, в которой есть пустой узел. В данном случае точка *C* должна быть вставлена в ветвь SE, где соответствующий узел пуст. Точно так же точка *D* будет вставлена в ветвь NE, исходящую из корня. Точку *E* тоже следовало бы вставить в ветвь NE корня. Однако, поскольку находящийся в ней узел уже занят, мы сравниваем ее с точкой *D* и вставляем в узел, находящийся в ветви NW узла *D*. Этот процесс продолжается, пока все точки не будут помещены в дерево. Очевидно, никто не гарантирует, что дерево получится сбалансированным, потому что мы используем ту последовательность точек, которая задана извне.

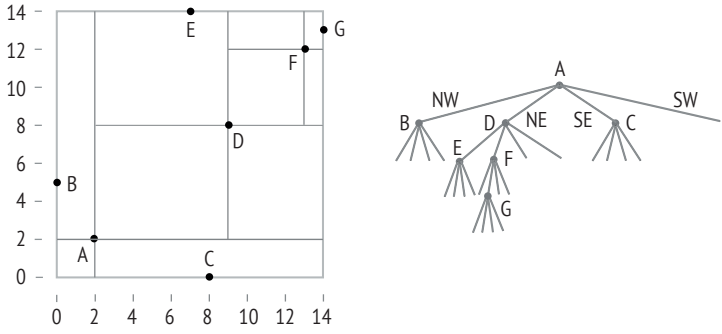

Рис. 6.2 ❖ Точечное квадродерево

Теперь определим структуру данных, которую можно использовать для хранения узла точечного квадродерева. Для этой цели предназначен класс `PQuadTreeNode` в листинге 6.2 (строка 5): мы храним точку и четыре ветви узла. В состав класса входит функция-член `is_leaf`, которая проверяет, является ли узел листовым; для этого нужно, чтобы все четыре дочерних узла были пусты, т. е. равны `None`. Функция `insert_pqtree` вставляет точку `p` в точечное квадродерево `q` (строка 37). Сначала мы находим узел, который станет родителем нового узла, а затем решаем, какую ветвь использовать для хранения нового узла. Для поиска подходящего узла служит функция `search_pqtree` (строка 18), которой передается параметр `is_find_only`, равный `False`. Реали-

зованный в ней алгоритм поиска спускается вниз по дереву (q), ориентируясь на соотношение между искомой точкой (p) и каждым узлом на пути поиска. Функция pointquadtree (строка 49) строит точечное квадродерево – сначала создается корень, а затем последовательно вставляются остальные точки. В конце программы данные, изображенные на рис. 6.2, используются для тестирования алгоритма.

Листинг 6.2 ❖ Точечное квадродерево (pointquadtree1.py).

```
 1 import sys
 2 sys.path.append('../geom')
 3 from point import *
 4
 5 class PQuadTreeNode():
 6     def__init__(self,point,nw=None,ne=None,se=None,sw=None):
 7         self.point = point
 8         self.nw = nw
 9         self.ne = ne
10         self.se = se
11         self.sw = sw
12     def __repr__(self):
13         return str(self.point)
14     def is_leaf(self):
15         return self.nw==None and self.ne==None and \
16                self.se==None and self.sw==None
17
18 def search_pqtree(q, p, is_find_only=True):
19     if q is None:
20         return
21     if q.point==p and is_find_only:
22         return q
23     dx,dy=0,0
24     if p.x>q.point.x:
25         dx=1
26     if p.y>q.point.y:
27         dy=1
28     qnum = dx+dy*2
29     child = [q.sw, q.se, q.nw, q.ne][qnum]
30     if child is None:
31         if not is_find_only:
32             return q
33         else:                          # q не является точкой, и больше нечего искать
34             return
35     return search_pqtree(child, p, is_find_only)
36
37 def insert_pqtree(q, p):
38     n = search_pqtree(q, p, False)
39     node = PQuadTreeNode(point=p)
40     if p.x<n.point.x and p.y<n.point.y:
41         n.sw = node
42     elif p.x<n.point.x and p.y>=n.point.y:
43         n.nw = node
```

```
44      elif p.x>=n.point.x and p.y<n.point.y:
45          n.se = node
46      else:
47          n.ne = node
48
49  def pointquadtree(data):
50      root = PQuadTreeNode(point = data[0])
51      for p in data[1:]:
52          insert_pqtree(root, p)
53      return root
54
55  if __name__ == '__main__':
56      data1 = [ (2,2), (0,5), (8,0), (9,8), (7,14),
57                (13,12), (14,13) ]
58      points = [Point(d[0], d[1]) for d in data1]
59      q = pointquadtree(points)
60      print [search_pqtree(q, p) for p in points]
```

Алгоритм поиска в круговом диапазоне очень похож на алгоритм для *k*D-дерева: мы стараемся игнорировать половину дерева, если эта половина содержит точки за пределами ближайшей границы квадрата, описанного вокруг окружности поиска. Разница, однако, в том, что теперь из каждого узла квадродерева исходит четыре ветви, мы должны объединить ветви по одну сторону от точки в узле, чтобы образовать правильную половину квадранта. Затем мы решаем, нужно ли продолжать поиск в каждой половине. Например, если левая граница круга больше координаты *X* точки в узле, то нужно продолжить рассмотрение всех точек к востоку от этого узла (правее), т. е. включить ветви NE и SE, тогда как точки к западу от узла (ветви NW и SW) заведомо не могут оказаться внутри окружности, так что их можно игнорировать. Мы проверяем подобные ситуации для западной, южной и северной сторон ограничивающего окружность прямоугольника. Точки, хранящиеся во всех остальных узлах, могут попасть внутрь окружности, поэтому нужно продолжить поиск во всех четырех квадрантах. Реализация этого алгоритма на Python приведена в листинге 6.3, где строка 10 относится к ситуации, в которой нужно просматривать только точки к востоку от узла. В конце кода производится тестирование на тех же данных для окружности радиуса 5 с центром в точке (5, 5) – поиск возвращает три точки, как и в случае *k*D-дерева.

Листинг 6.3 ❖ Поиск в круговом диапазоне с помощью точечного квадродерева (pointquadtree2.py)

```
1 import sys
2 sys.path.append('../geom')
3 from point import *
4 from pointquadtree1 import *
5
6 def range_query(t, p, r):
7     def rquery(t, p, r, found):
8         if t is None:
```

```
 9            return
10        if p.x-r > t.point.x :
11            rquery(t.ne, p, r, found)   # только точки справа
12            rquery(t.se, p, r, found)
13            return
14        if p.y-r > t.point.y:
15            rquery(t.ne, p, r, found)   # только точки выше
16            rquery(t.nw, p, r, found)
17            return
18        if p.x+r < t.point.x:
19            rquery(t.nw, p, r, found)   # только точки слева
20            rquery(t.sw, p, r, found)
21            return
22        if p.y+r < t.point.y:
23            rquery(t.se, p, r, found)   # только точки ниже
24            rquery(t.sw, p, r, found)
25            return
26        if p.distance(t.point) <= r:
27            found.append(t.point)
28        rquery(t.nw, p, r, found)
29        rquery(t.ne, p, r, found)
30        rquery(t.se, p, r, found)
31        rquery(t.sw, p, r, found)
32        return
33    found = []
34    if t is not None:
35        rquery(t, p, r, found)
36    return found
37
38 if __name__ == '__main__':
39    data1 = [ (2,2), (0,5), (8,0), (9,8), (7,14),
40              (13,12), (14,13) ]
41    points = [Point(d[0], d[1]) for d in data1]
42    q = pointquadtree(points)
43    p = Point(5, 5)
44    print range_query(q, p, 5)
```

Наконец, применим алгоритм поиска ближайшего соседа к точечному квадродереву (листинг 6.4). Алгоритм реализован в функции `pq_nnquery`, его идея в том, чтобы спускаться по дереву максимально глубоко, до тех пор пока остаются точки, подлежащие проверке. Для поддержания списка найденных точек мы используем ту же функцию `update_neighbors`, что в коде для *k*D-дерева (строка 2). Как и раньше, найдя точку рядом с целевой, мы сохраняем ее саму и расстояние до целевой точки в списке. Дойдя до листового узла, мы сравниваем его с найденными к этому моменту ближайшими точками (строка 24). Для нелистовых узлов мы вызываем функцию `pqcompare`, которая находит квадрант, ближайший к целевой точке (строка 26), затем идем по соответствующей ветви для продолжения поиска (строка 28) – и так до тех пор, пока не дойдем до листьев. Затем, если точка, хранящаяся в нелистовом узле, оказалась ближе к целевой, чем самая дальняя из уже найденных, мы должны проверить и другие квадранты (код, начинающийся в строке 30). Для

удобства мы пользуемся функцией-оберткой `nearest_neighbor_query` (строка 36), которая возвращает список найденных точек вместе с расстояниями до целевой точки. В конце программы мы тестируем алгоритм на тех же данных, что и раньше, и при поиске трех точек, ближайших к (5, 5), получаем в точности такой же результат:

```
[[(2, 2), 4.242640687119285], [(9, 8), 5.0], [(0, 5), 5.0]]
[4.242640687119285, 5.0, 5.0]
```

Листинг 6.4 ❖ Поиск ближайшего соседа с использованием точечного квадродерева (pointquadtree3.py)

```
 1 from pointquadtree1 import *
 2 from kdtree3 import update_neighbors
 3
 4 INF = float('inf')
 5 pqmaxdist = INF
 6
 7 # возвращает квадрант t, в котором находится p
 8 # 0-NW, 1-NE, 2-SE, 3-SW
 9 def pqcompare(t, p):
10     if p.x<t.point.x and p.y<t.point.y:
11         return 3                    # sw
12     elif p.x<t.point.x and p.y>=t.point.y:
13         return 0
14     elif p.x>=t.point.x and p.y<t.point.y:
15         return 2
16     else:
17         return 1
18
19 def pq_nnquery(t, p, n, found):
20     global pqmaxdist
21     if t is None:
22         return
23     if t.is_leaf():
24         pqmaxdist = update_neighbors(t.point, p, found, n)
25         return
26     quad_index = pqcompare(t, p)
27     quads = [t.nw, t.ne, t.se, t.sw]
28     pq_nnquery(quads[quad_index], p, n, found)
29     pqmaxdist = update_neighbors(t.point, p, found, n)
30     if (t.point.distance(p)) < pqmaxdist:
31         for i in range(4):
32             if i <> quad_index:
33                 pq_nnquery(quads[i], p, n, found)
34     return
35
36 def nearest_neighbor_query(t, p, n=1):
37     nearest_neighbors = []
38     pq_nnquery(t, p, n, nearest_neighbors)
39     return nearest_neighbors[:n]
40
```

```
41 if __name__ == '__main__':
42     data1 = [ (2,2), (0,5), (8,0), (9,8), (7,14),
43               (13,12), (14,13) ]
44     points = [Point(d[0], d[1]) for d in data1]
45     q = pointquadtree(points)
46     p = Point(5,5)
47     n = 3
48     print nearest_neighbor_query(q, p, n)
49     print sorted([p.distance(x) for x in points])[:n]
```

6.3. Примечания

В работе Finkel and Bentley (1974) впервые развита идея квадродерева как структуры данных для поиска информации. Заметим, что фамилия Бентли уже трижды встречалась в этой книге: алгоритм Бентли–Оттмана для проверки пересечения отрезков, *k*D-дерево и вот теперь эта глава о квадродеревьях. Как и в случае *k*D-деревьев, множество полезных деталей о квадродеревьях можно почерпнуть в книгах Samet (1990a,b, 2006).

Стоимость построения точечного квадродерева из случайного набора точек составляет $O(n \log_4 n)$. Простой поиск в сбалансированном точечном квадродереве имеет временную сложность $O(n \log_4 n)$, в худшем случае, когда на каждом уровне дерева находится только один узел, сложность равна $O(n)$. Показано, что стоимость поиска в диапазоне равна $O(2n^{1/2})$ в худшем случае (Bentley et al., 1977; Lee and Wong, 1977). В работе Lee and Wong (1977) также показано, что поиск s точек занимает время $O(sn^{1/2})$ в худшем случае.

Как создать сбалансированное точечное квадродерево? По сравнению к *k*D-деревом, это более трудная задача. Напомним, что в *k*D-дереве мы попеременно используем координаты X и Y для разбиения данных на две половины с применением медианной точки. Но в случае точечного квадродерева сделать это трудно, потому что координаты X и Y используются одновременно. Идея в том, чтобы добиться хотя бы примерно одинакового количества точек слева и справа или сверху и снизу, но не во всех четырех квадрантах одновременно. Чтобы сделать это, мы сначала находим медианную точку по координатам X и используем ее для разбиения пространства. На следующем уровне разбиения точки сортируются по координатам Y. Направления чередуются, пока дерево не будет построено полностью.

Еще один способ справиться с несбалансированностью точечных квадродеревьев – разбивать пространство по центру, а не по точке данных. Это называется точечно-регионным квадродеревом (point region quadtree), и в предыдущей главе мы видели, как эта идея применяется к *k*D-деревьям. Если точки распределены в пространстве случайно или равномерно, то этот подход оказывается очень эффективен, потому что с большой вероятностью дерево будет иметь сбалансированные ветви. Но если в множестве точек имеются выраженные кластеры, то может получиться сильно несбалансированное квадродерево. В следующей главе мы исследуем квадродеревья такого типа.

6.4. Упражнения

1. Мы уделили не слишком много внимания введению в регионные квадродеревья, ограничившись лишь построением дерева и простыми запросами. Но нетрудно разработать алгоритм запроса к диапазонам с использованием таких деревьев. Проанализируйте алгоритм запроса к диапазону с использованием *k*D-деревьев из предыдущей главы и спроектируйте аналогичный алгоритм для регионных квадродеревьев. При решении могут быть полезны идеи реализации запроса для точечных квадродеревьев.
2. Для нелистового узла регионного квадродерева мы в качестве значения в этом узле использовали `None`. Однако было бы очень полезно знать значения в регионе, представленном этим узлом. Напишите на Python программу, которая будет сообщать значения в каждом узле регионного квадродерева. Разумеется, для этого необходим исчерпывающий поиск во всех ветвях, исходящих из узла.
3. В предыдущей главе мы сравнивали различные методы индексирования с помощью *k*D-дерева, пользуясь наборами данных разного размера. Напишите на Python программу оценки эффективности точечных квадродеревьев на случайных наборах данных из предыдущей главы и сравните точечные квадродеревья с другими методами индексирования пространственных данных.
4. Напишите на Python программу, которая оптимизирует точечные квадродеревья, попеременно используя точки с медианными значениями координат *X* и *Y*. Также было бы интересно сравнить сбалансированное и несбалансированное деревья.
5. Модифицируйте код в файле `pointquadtree1.py`, так чтобы он печатал глубину квадродерева.
6. Спроектируйте алгоритм запросов к прямоугольным диапазонам и реализуйте его на Python.

Глава 7

Индексирование отрезков и многоугольников

В двух предыдущих главах мы занимались индексированием точек (а регионные квадродеревья предназначены для растровых данных, которые, конечно, можно рассматривать как регулярно расположенные точки). Мы удостоверились, что индексирование пространственных данных существенно ускоряет пространственные запросы, хотя за это приходится расплачиваться стоимостью начального построения дерева и накладными расходами на хранение данных в дереве. Впрочем, достигаемое ускорение с лихвой окупает затраты при многократном использовании дерева. В этой главе мы рассмотрим еще два подхода к индексированию, предназначенных для прямых и многоугольников.

7.1. Квадродеревья полигональных карт

Мы видели, как квадродеревья можно использовать для индексирования точек путем разбиения пространства до тех пор, пока все точки не будут вставлены в дерево. Здесь же мы воспользуемся другим видом квадродерева, точечно-регионным (PR), когда пространство разбивается по центру каждого региона, а не по точке данных. Мы видели, как это работает в другом контексте, для точечно-регионных *k*D-деревьев (см. раздел 5.2), а на рис. 7.1 приведен соответствующий пример. Сначала пространство разбивается на четыре квадранта, используя центральную точку всей области. Затем мы разбиваем каждый квадрант на меньшие, если выполнено некоторое условие. Первое условие заключается в том, что разбиение продолжается, пока в квадранте больше одной точки. Затем регионы разбиваются далее, если выполняется некоторое условие, связанное с отрезками. На рис. 7.1 показана ситуация, когда такое разбиение имеет место – регион содержит больше одного отрезка, и при этом отрезки не сходятся в одной вершине, которая принадлежит тому же региону. Например, правый нижний регион не разбивается, потому что содержит два отрезка, соединяющихся в точке, принадлежащей тому

же региону. То же самое относится к региону над ним, где в одной точке сходятся четыре отрезка. Такой вид квадродерева называется квадродеревом полигональной карты, или PM-квадродеревом (polygonal map – PM), потому что его можно использовать для индексирования прямолинейных отрезков и многоугольников. В зависимости от способа разбиения на основе отрезков выделяют три варианта PM-квадродеревьев. На рис. 7.1 показано PM1-квадродерево.

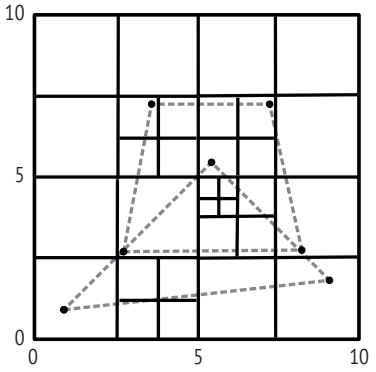

Рис. 7.1 ❖ Разбиение пространства для PM1-квадродерева

Прежде чем переходить к алгоритмам создания PM-квадродеревьев, необходимо решить два важных вопроса. Во-первых, поскольку мы имеем дело с отрезками и многоугольниками, нужна хорошая структура данных для хранения такой информации. Для точек мы пользовались классом `Point`, содержащим всю необходимую информацию, в частности координаты и некоторые операции для работы с точками. Для отрезков и многоугольников можно было бы воспользоваться двусвязным списком ребер (ДСР), введенным в главе 3. Но в ДСР применяется концепция полуребра для сохранения топологической информации о связях между точками, отрезками и многоугольниками. Для этой цели ДСР вполне пригоден, но не дает в явном виде информацию о ребрах как таковых, потому что каждое ребро представлено двумя полуребрами, что неудобно, если нам всего-то и нужно, что хранить ребра в PM-квадродереве. Поэтому мы расширим класс DCEL и создадим новый класс XDCEL (листинг 7.1, буква X означает «extended» – расширенный). При этом мы определим отдельный класс `Edge`, в котором хранятся входящий и исходящий узлы, а также левый и правый многоугольники для каждого ребра (строки 10–13); эту структуру данных можно встретить в любом стандартном учебнике ГИС[1]. Функция `eq`, начинающаяся в строке 14, определяет, являются ли два ребра эквивалентными (т. е. совпадают ли их концы), не обращая внимания на порядок концов. Функция `extent` возвращает минимальные и максимальные координаты *X* и *Y* ребра в виде экземпляра класса `Extent` (см. ниже). Функция `is_endpoint` будет полезна для определения того, обладают ли несколько ребер общей концевой точкой.

Сам класс `Xdcel` (строка 30) наследует классу `Dcel`. Мы берем полуребра, вершины и грани из `Dcel` и добавляем в структуру данных список ребер (стро-

[1] Однако следует отметить, что ребро в ДСР – это просто прямолинейный отрезок, тогда как в строгом смысле топологического отношения в ГИС ребро – это дуга, соединяющая два узла, которые являются точками пересечения дуг. В контексте нашего ДСР все вершины, являющиеся пересечениями или нет, считаются концами ребер. В ДСР может храниться избыточная информация о нескольких ребрах между двумя точками пересечения, потому что у этих ребер в точности одинаковые левый и правый многоугольники. Но такая избыточность удобна, поскольку мы можем сохранить все вершины, присутствующие в данных, а не поддерживать отдельный набор данных о точках, соответствующих вершинам, для которых нет узлов.

ка 35). Ребра строятся по данным из Dcel функцией-членом build_xdcel, которая перебирает все полуребра и определяет начальный и конечный узлы и левый и правый многоугольники для каждого ребра.

Листинг 7.1 ❖ Расширенный двусвязный список ребер (xdcel.py)

```
import sys
sys.path.append('../geom')
from dcel import *
from extent import *

class Edge:
    """Ребро в топологически связанном пространственном наборе данных"""
    def __init__(self, v1, v2, leftpoly, rightpoly):
        # Начало определяется как вершина, в которую ведет ребро
        self.fr = v1
        self.to = v2
        self.left = leftpoly
        self.right = rightpoly
    def __eq__(self, other): # v1->v2 is eq. to v2->v1
        return (self.fr == other.fr and self.to == other.to)\
            or (self.fr == other.to and self.to == other.fr)
    def extent(self):
        xmin = min(self.fr.x, self.to.x)
        xmax = max(self.fr.x, self.to.x)
        ymin = min(self.fr.y, self.to.y)
        ymax = max(self.fr.y, self.to.y)
        return Extent(xmin, xmax, ymin, ymax)
    def is_endpoint(self, p):
        if p == self.fr or p==self.to:
            return True
        return False
    def __repr__(self):
        return "{0}->{1}".format(self.fr, self.to)

class Xdcel(Dcel):
    def __init__(self, D):
        Dcel.hedges = D.hedges
        Dcel.vertices = D.vertices
        Dcel.faces = D.faces
        self.edges=[]
        self.build_xdcel()
    def build_xdcel(self):
        """Построить объекты Edge по информации, хранящейся в ДСР"""
        if not len(self.vertices) or not len(self.hedges):
            return
        for h in self.hedges:
            v1 = h.origin
            v2 = h.nexthedge.origin
            lf = h.face
            rf = h.twin.face
            e = Edge(v1, v2, lf, rf)
```

```
47              try:
48                  i = self.edges.index(e)
49              except ValueError:
50                  i = None
51              if i is None:
52                  self.edges.append(e)
```

Второй вопрос – это экстент ребра. Мы используем класс `Extent`, чтобы сохранить границы координат геометрического объекта (листинг 7.2). Это общий класс, который пригоден не только для ребер, но и для многоугольников, как мы увидим в следующем разделе. Он содержит данные, необходимые для определения экстента: границы координат *X* и *Y*. Экстент – это, по существу, ограничивающий прямоугольник объекта. Переопределив функцию `__getitem__` (строка 12), мы сможем использовать объект экстента как список с индексами 0, 1, 2 и 3, по которым можно обращаться к четырем границам прямоугольника. Имена остальных функций-членов этого класса говорят сами за себя. Остановимся разве что на функции `distance`, которая возвращает кратчайшее расстояние между двумя экстентами, вычисляемое по двум ближайшим углам.

Листинг 7.2 ❖ Класс Extent (extent.py)

```
 1 from math import sqrt
 2 import sys
 3 sys.path.append('../geom')
 4 from point import *
 5
 6 class Extent():
 7     def __init__(self, xmin=0, xmax=0, ymin=0, ymax=0):
 8         self.xmin = xmin
 9         self.xmax = xmax
10         self.ymin = ymin
11         self.ymax = ymax
12     def __getitem__(self, i):
13         if i==0: return self.xmin
14         if i==1: return self.xmax
15         if i==2: return self.ymin
16         if i==3: return self.ymax
17         return None
18     def __repr__(self):
19         return "[{0}, {1}, {2}, {3}]".format(
20             self.xmin, self.xmax ,self.ymin, self.ymax)
21     def __eq__(self, other):
22         return self.xmin==other.xmin and\
23             self.xmax==other.xmax and \
24             self.ymin==other.ymin and \
25             self.ymax==other.ymax
26     def touches(self, other):
27         return not (self.xmin>other.xmax or\
28             self.xmax<other.xmin or\
29             self.ymin>other.ymax or\
```

```
            self.ymax<other.ymin)
    def contains(self, point):
        return not (self.xmin>point.x or\
            self.xmax<point.x or\
            self.ymin>point.y or\
            self.ymax<point.y)
    def getcenter(self):
        return Point(self.xmin+(self.xmax-self.xmin)/2.0,
            self.ymin+(self.ymax-self.ymin)/2.0)
    def getwidth(self):
        return self.xmax-self.xmin
    def getheight(self):
        return self.ymax-self.ymin
    def is_minimal(self):
        return (self.xmax-self.xmin)<0.1 and\
            (self.ymax-self.ymin)<0.1
    def area(self):
        return (self.xmax-self.xmin)*(self.ymax-self.ymin)
    def intersect(self, other):
        if not self.touches(other):
            return 0
        xmin = max(self.xmin, other.xmin)
        xmax = min(self.xmax, other.xmax)
        ymin = max(self.ymin, other.ymin)
        ymax = min(self.ymax, other.ymax)
        return (xmax-xmin)*(ymax-ymin)
    def distance(self, other):
        x1 = self.xmin+(self.xmax-self.xmin)/2.0
        y1 = self.ymin+(self.ymax-self.ymin)/2.0
        x2 = other.xmin+(other.xmax-other.xmin)/2.0
        y2 = other.ymin+(other.ymax-other.ymin)/2.0
        return sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2))
```

7.1.1. PM1-квадродеревья

Общая процедура создания квадродерева полигональной карты состоит в том, чтобы сначала создать точечно-регионное дерево, а затем посмотреть, какие квадранты нуждаются в дальнейшем разбиении. Прежде всего определим структуру данных для хранения информации об узле PM-квадродерева. Класс `PMQuadTreeNode` представлен в листинге 7.3. Бросается в глаза сходство со структурой узла точечно-регионного *k*D-дерева, в которой также есть член `center`, описывающий центр квадранта, представленного узлом. Здесь мы явно используем класс `Extent` для представления границ квадранта, и этот же класс будет еще использован ниже в данной главе. При желании мы могли бы использовать класс `Extent` и в точечно-регионном *k*D-дереве (см. упражнение 1 в конце этой главы). В узле PM-квадродерева четыре квадранта упакованы в список `quads`. Как станет ясно при рассмотрении последующего кода, такое изменение оказывается очень удобным для реализации алгоритмов. Узел может содержать ребра, которые мы храним в списке `edges`. Если пред-

ставленный узлом регион содержит точку данных, то она хранится в члене `vertex`. Переменная-член `type` описывает, является ли данный узел нелистовым (серый узел), узлом, регион которого содержит вершину или пересекается с отрезками (черный узел), или узлом, который не содержит вершины и не пересекается с отрезками (белый узел). Очевидно, что черные и белые узлы являются листовыми.

Первый шаг создания PM1-квадродерева – создание точечно-регионного квадродерева. Это делает функция `split_by_points` (строка 30), которая разбивает пространство рекурсивно. В процессе разбиения пространства основная задача алгоритма заключается в том, чтобы вычислять экстенты новых квадрантов, зная границы и центр текущего экстента (строка 38–47). Имея следующие экстенты, мы можем создать четыре новых узла (строка 49). Затем вычисляется подмножество точек, принадлежащих каждому из меньших экстентов (строка 52). После этого подмножества точек и новые узлы используются для дальнейшего разбиения пространства – и так до тех пор, пока в каждом квадранте останется не более одной точки (строки 31 и 35). Функция `search_pmquadtree` (строка 59) реализует алгоритм поиска региона, содержащего целевую точку, который продолжается, пока не дойдет до листового (не серого) узла. Наконец, функция `is_intersect` проверяет, верно ли, что хотя бы одна из четырех сторон экстента пересекает ребро. Здесь мы используем рассмотренные в разделе 3.1 алгоритмы проверки пересечения двух отрезков.

Листинг 7.3 ❖ Код, общий для всех PM-квадродеревьев (pmquadtree.py)

```
 1 # Используется всеми PM-квадродеревьями
 2 import sys
 3 sys.path.append('../geom')
 4 from extent import *
 5 from point import *
 6 from linesegment import Segment
 7 from intersection import test_intersect
 8
 9 BLACK = 2   # узлы, содержащие точку или отрезок
10 WHITE = 1   # узлы, не содержащие точек и не пересекающиеся с отрезком
11 GREY = 0    # промежуточные узлы
12
13 class PMQuadTreeNode():
14     def __init__(self, point, extent,
15                  nw=None, ne=None, se=None, sw=None):
16         self.point = point # center
17         self.extent = extent
18         self.quads = [nw, ne, se, sw]
19         self.vertex = None
20         self.edges = []
21         self.type = GREY
22   def __repr__(self):
23         return str(self.point)
24   def __getitem__(self, i):
25         if i<4: return self.quads[i]
```

```
26          return None
27      def is_leaf(self):
28          return sum([ q is None for q in self.quads])==4
29
30  def split_by_points(points, pmq):
31      if len(points) == 1:
32          pmq.vertex = points[0]
33          pmq.type = BLACK
34          return
35      if len(points) ==0:
36          pmq.type = WHITE
37          return
38      xmin = pmq.extent.xmin
39      xmax = pmq.extent.xmax
40      ymin = pmq.extent.ymin
41      ymax = pmq.extent.ymax
42      xmid = xmin + (xmax-xmin)/2.0
43      ymid = ymin + (ymax-ymin)/2.0
44      exts = [ Extent(xmin, xmid, ymid, ymax), # nw
45               Extent(xmid, xmax, ymid, ymax), # ne
46               Extent(xmid, xmax, ymin, ymid), # se
47               Extent(xmin, xmid, ymin, ymid)  # sw
48      ]
49      pmq.quads = [PMQuadTreeNode(exts[i].getcenter(),exts[i])
50                   for i in range(4)]
51      subpoints = [[], [], [], []]      # четыре пустых пункта списка
52      for p in points:
53          for i in range(4):
54              if exts[i].contains(p):
55                  subpoints[i].append(p)
56      for i in range(4):
57          split_by_points(subpoints[i], pmq.quads[i])
58
59  def search_pmquadtree(pmq, x, y):
60      if pmq.type is not GREY:
61          return pmq
62      for q in pmq.quads:
63          if q.extent.contains(Point(x, y)):
64              return search_pmquadtree(q, x, y)
65      return None
66
67  def is_intersect(extent, edge):
68      if not extent.touches(edge.extent()):
69          return False
70      # четыре угла по часовой стрелке
71      p1 = Point(extent.xmin, extent.ymin)
72      p2 = Point(extent.xmin, extent.ymax)
73      p3 = Point(extent.xmax, extent.ymax)
74      p4 = Point(extent.xmax, extent.ymin)
75      segs = [ Segment(0, p1, p2), Segment(1, p2, p3),
76               Segment(2, p3, p4), Segment(3, p4, p1) ]
77      s0 = Segment(4, edge.fr, edge.to)
```

```
78      for s in segs:
79          if test_intersect(s, s0):
80              return True
81      return False
```

Функция `split_by_edges_pm1` в листинге 7.4 реализует рекурсивный алгоритм, завершающий построение PM1-квадродерева. Эта функция принимает два параметра: список объектов `Edge` и узел точечно-регионного квадродерева, который содержит все вершины ребер. Алгоритм начинает работу с нахождения ребер, которые пересекают экстент региона входного узла (строка 8). Это может произойти в двух случаях: когда ребро действительно пересекает границы региона (строка 9) или когда оно целиком содержится внутри региона (строка 12). Если таких ребер не существует (строка 14), то мы приписываем узлу тип `WHITE`, показывающий, что это листовой узел, не содержащий ни точек, ни ребер, так что его дальнейшее разбиение не требуется (поэтому функция возвращает управление). Для узла, регион которого содержит вершину (строка 17), мы проверяем, что все ребра, пересекающие его границу, исходят из одной и той же вершины внутри данного региона (цикл `for`, начинающийся в строке 19). Если все ребра внутри региона действительно исходят из одной вершины (строка 22), то мы добавляем ребра в список `edges`, являющийся членом класса узла, присваиваем узлу тип `BLACK` и прекращаем обработку этого узла (`return`), поскольку разбивать регион дальше не нужно. Если ребра исходят из разных вершин, то разбиение региона продолжается.

Если узел не содержит ни одной вершины (строка 27), то мы прекращаем разбиение пространства, только когда существует одно ребро, пересекающее регион. В этом случае (строка 28) узлу присваивается тип `BLACK`, ребра добавляются в список `edges`, и функция возвращает управление. Прежде чем двигаться дальше, нужно выполнить код в строке 32, потому что мы хотим избежать ситуаций, когда разбиение продолжается до бесконечности или регионы становятся слишком малы. Такое возможно, когда регион пересекают два отрезка, которые очень близки друг к другу и к углам региона. Если эти отрезки не исходят из вершины, находящейся внутри региона, то разбиение зайдет очень далеко, прежде чем мы сможем разделить отрезки по разным квадрантам.

Проверив все особые случаи, мы оказываемся в точке, где должны физически разбить регион на четыре новых квадранта. Осталось сделать две вещи. Во-первых, нужно определить экстенты этих меньших квадрантов. Это мы уже делали раньше в функции `split_by_points function` при создании начального точечно-регионного квадродерева (листинг 7.3). Сейчас мы используем такую же стратегию для получения новых экстентов и создания новых квадрантов (строки 38–47). Конечно, мы будем этим заниматься, только в листовом узле точечно-регионного квадродерева (строка 37). Не имеет никакого смысла разбивать нелистовой узел (поскольку он уже разбит – потому-то он и нелистовой). Во-вторых, нужно решить, какой из четырех новых квадрантов получит вершину (строка 52). После этого мы стираем признак наличия вершины в текущем узле (строка 56), потому что она теперь находится в одном из четырех его дочерних узлов.

Проделав все описанное выше, мы рекурсивно вызываем себя четыре раза, передавая каждый из новых квадрантов, который либо станет листовым узлом, либо будет разбит на более мелкие (строка 58 листинга 7.4). Поскольку мы рассмотрели все возможные случаи, рекурсивный вызов успешно завершается.

Листинг 7.4 ❖ PM1-квадродерево (pm1quadtree.py)

```
 1 import sys
 2 sys.path.append('../geom')
 3 from xdcel import *
 4 from pmquadtree import *
 5
 6 def split_by_edges_pm1(edges, pmq):
 7     subedges = []
 8     for e in edges:
 9         if is_intersect(pmq.extent, e):
10             subedges.append(e)
11         elif pmq.extent.contains(e.fr) and\
12               pmq.extent.contains(e.to):
13             subedges.append(e)
14     if len(subedges) == 0:
15         pmq.type = WHITE
16         return
17     if pmq.vertex is not None:
18         is_same_source = True
19         for e in subedges:
20             if not e.is_endpoint(pmq.vertex):
21                 is_same_source = False
22         if is_same_source:
23             for e in subedges:
24                 pmq.edges.append(e)
25             pmq.type = BLACK
26             return
27     else:                          # pmq не содержит вершин
28         if len(subedges) == 1:
29             pmq.type = BLACK
30             pmq.edges.append(subedges[0])
31             return
32     if pmq.extent.is_minimal():
33         for e in subedges:
34             pmq.edges.append(e)
35         pmq.type = BLACK
36         return
37     if pmq.is_leaf():              # теперь разбиваем регион, если необходимо
38         xmin = pmq.extent.xmin
39         xmax = pmq.extent.xmax
40         ymin = pmq.extent.ymin
41         ymax = pmq.extent.ymax
42         xmid = xmin + (xmax-xmin)/2.0
43         ymid = ymin + (ymax-ymin)/2.0
44         exts = [ Extent(xmin, xmid, ymid, ymax),  # nw
```

```
45                 Extent(xmid, xmax, ymid, ymax),  # ne
46                 Extent(xmid, xmax, ymin, ymid),  # se
47                 Extent(xmin, xmid, ymin, ymid)   # sw
48               ]
49         pmq.quads = [ PMQuadTreeNode(exts[i].getcenter(),
50                                      exts[i])
51                       for i in range(4) ]
52     if pmq.vertex:
53         for q in pmq.quads:
54             if q.extent.contains(pmq.vertex):
55                 q.vertex = pmq.vertex
56         pmq.vertex = None
57     for i in range(4):
58         split_by_edges_pm1(subedges, pmq.quads[i])
```

Для тестирования алгоритма PM1 нам нужны данные ДСР. Мы будем использовать многоугольники, изображенные на рис. 7.1, т. е. результат операции наложения, описанной в разделе 3.2. На практике можно сохранить объект Python в файле, сериализовав двоичную информацию. Конкретно, мы можем добавить следующие две строки в конец листинга 3.6. В результате будет создан ASCII-файл `mydcel.pickle`, в котором хранится вся информация из объекта `D` класса `Dcel`.

```
import pickle
pickle.dump(D, open('mydcel.pickle', 'w'))
```

Теперь этим файлом можно воспользоваться, десериализовав содержащуюся в нем информацию. В следующем фрагменте показано, как восстановить данные в объект `D` (строка 4), а затем использовать этот объект для создания объекта `XD` класса `Xdcel` (строка 5). В строках 7–12 создается наибольший экстент, который используется для конструирования корневого узла (строка 14). В строках 15 и 16 создается PM1-квадродерево, а в строке 17 ищется узел в точке (1, 1). На рис. 7.2 показано PM1-квадродерево, построенное по тестовым данным.

```
 1 import pickle
 2 from pm1quadtree import *
 3
 4 D = pickle.load(open('mydcel.pickle'))
 5 XD = Xdcel(D)
 6
 7 X = [v.x for v in D.vertices]
 8 Y = [v.y for v in D.vertices]
 9 xmin,xmax,ymin,ymax = min(X)-1, max(X)+1, min(Y)-1, max(Y)+1
10 maxmax = max(xmax,ymax)
11 xmax=ymax=maxmax
12 extent = Extent(xmin, xmax, ymin, ymax)
13
14 pmq = PMQuadTreeNode(extent.getcenter(), extent)
15 split_by_points(XD.vertices, pmq)
16 split_by_edges_pm1(XD.edges, pmq)
17 print search_pmquadtree(pmq, 1, 1)
```

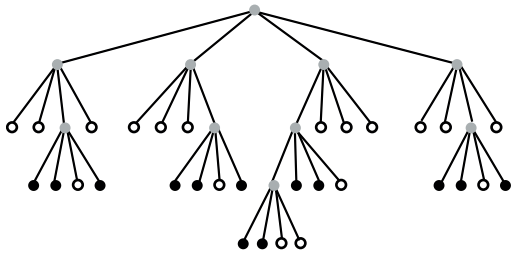

Рис. 7.2 ❖ PM1-квадродерево.
Серые узлы представляют промежуточные квадранты,
подлежащие дальнейшему разбиению, белые узлы – регионы, не содержащие ни вершин,
ни ребер, а черные – регионы, которые содержат вершину или пересекаются с ребрами

7.1.2. PM2-квадродеревья

Недостаток PM1-квадродеревьев – большая глубина: они могут оказаться слишком глубокими, потому что регион нужно разбивать, если он пересекается с отрезками, вершины которых находятся в других регионах. Поэтому для хранения дерева нужно много памяти. В PM2-квадродереве мы допускаем несколько ребер в регионе, если только у них общая вершина, пусть даже она находится в другом регионе. Иными словами, регион не разбивается, если все отрезки, которые он содержит, пересекаются в одной вершине, находящейся вне региона. Это может значительно уменьшить количество и глубину разбиений при построении дерева. На рис. 7.3 приведен пример PM2-квадродерева. По сравнению с PM1-квадродеревом на рис. 7.1 можно заметить, что квадранты в левой нижней половине всей области не нужно разбивать дальше, потому что отрезки в каждом квадранте имеют общую вершину.

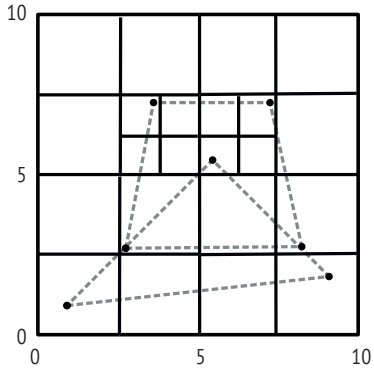

Рис. 7.3 ❖ Разбиение пространства
для PM2-квадродерева

Алгоритм построения PM2-квадродерева приведен в листинге 7.5. Хотя он похож на алгоритм для PM1-квадродеревьев, некоторые части ориентиро-

ваны специально на PM2-дерево. Во-первых, если регион содержит только одно ребро (строка 16), то узел помечается как листовой (типа BLACK). Если в регионе находится несколько отрезков (строка 20) с общей вершиной (строки 23–31), то независимо от положения этой вершины узел также помечается как листовой типа BLACK (строка 35). Для тестирования в функции test используется объект Xdcel с теми же данными, что для PM1-квадродерева.

Листинг 7.5 ❖ PM2-квадродерево (pm2quadtree.py)

```
 1 from xdcel import *
 2 from pmquadtree import *
 3 import pickle
 4
 5 def split_by_edges_pm2(edges, pmq):
 6     subedges = []
 7     for e in edges:
 8         if is_intersect(pmq.extent, e):
 9             subedges.append(e)
10         elif pmq.extent.contains(e.fr) and\
11              pmq.extent.contains(e.to):
12             subedges.append(e)
13     if len(subedges) == 0:
14         pmq.type = WHITE
15         return
16     elif len(subedges) == 1:
17         pmq.type = BLACK
18         pmq.edges.append(subedges[0])
19         return
20     else:
21         p1,p2 = subedges[0].fr, subedges[0].to
22         common_vertex = None
23         if subedges[1].is_endpoint(p1):
24             common_vertex = p1
25         elif subedges[1].is_endpoint(p2):
26             common_vertex = p2
27         if common_vertex is not None:
28             for e in subedges[2:]:
29                 if not e.is_endpoint(common_vertex):
30                     common_vertex = None
31                     break
32         if common_vertex is not None:
33             for e in subedges:
34                 pmq.edges.append(e)
35             pmq.type = BLACK
36             return
37         if pmq.extent.is_minimal():
38             for e in subedges:
39                 pmq.edges.append(e)
40             pmq.type = BLACK
41             return
42         if pmq.is_leaf():
43             xmin = pmq.extent.xmin
```

```
            xmax = pmq.extent.xmax
            ymin = pmq.extent.ymin
            ymax = pmq.extent.ymax
            xmid = xmin + (xmax-xmin)/2.0
            ymid = ymin + (ymax-ymin)/2.0
            exts = [ Extent(xmin, xmid, ymid, ymax), # nw
                     Extent(xmid, xmax, ymid, ymax), # ne
                     Extent(xmid, xmax, ymin, ymid), # se
                     Extent(xmin, xmid, ymin, ymid)  # sw
                   ]
            pmq.quads = [ PMQuadTreeNode(exts[i].getcenter(),
                                         exts[i])
                          for i in range(4) ]
        if pmq.vertex:
            for q in pmq.quads:
                if q.extent.contains(pmq.vertex):
                    q.vertex = pmq.vertex
            pmq.vertex = None
        for i in range(4):
            split_by_edges_pm2(subedges, pmq.quads[i])

def test():
    D = pickle.load(open('../data/mydcel.pickle'))
    XD = Xdcel(D)

    X = [v.x for v in D.vertices]
    Y = [v.y for v in D.vertices]
    xmin,xmax,ymin,ymax = min(X)-1, max(X)+1,\
                          min(Y)-1, max(Y)+1
    maxmax = max(xmax,ymax)
    xmax=ymax=maxmax
    extent = Extent(xmin, xmax, ymin, ymax)

    pm2q = PMQuadTreeNode(extent.getcenter(), extent)
    split_by_points(XD.vertices, pm2q)
    split_by_edges_pm2(XD.edges, pm2q)
    print search_pmquadtree(pm2q, 10, 10)

if __name__ == '__main__':
    test()
```

7.1.3. PM3-квадродеревья

С помощью PM3-квадродерева мы можем еще уменьшить вероятность разбиения регионов, потому что теперь разбиваются только регионы, содержащие более одной вершины. В результате PM3-квадродерево оказывается таким же, как точечно-регионное квадродерево. На рис. 7.4 показано разбиение на примере тех же данных, что и выше. Реализация PM3-квадродерева не вызывает затруднений, поскольку дополнительного разбиения региона просто нет. Мы только должны поместить в узлы информацию о ребрах, что и делает функция `split_by_edges_pm3` в листинге 7.6.

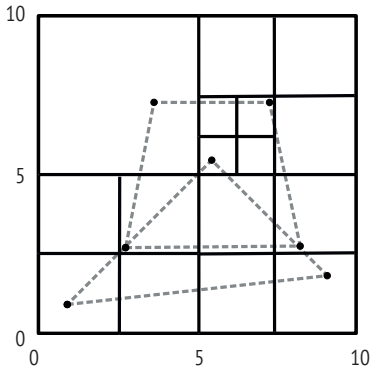

Рис. 7.4 ❖ Разбиение пространства для PM3-квадродерева

Листинг 7.6 ❖ PM3-квадродерево (pm3quadtree.py)

```
from xdcel import *
from pmquadtree import *
import pickle

def split_by_edges_pm3(edges, pmq):
    subedges = []
    for e in edges:
        if is_intersect(pmq.extent, e):
            subedges.append(e)
        elif pmq.extent.contains(e.fr) and\
             pmq.extent.contains(e.to):
            subedges.append(e)
    if not pmq.is_leaf():
        for i in range(4):
            split_by_edges_pm3(subedges, pmq.quads[i])
        return
    if len(subedges) == 0:
        pmq.type = WHITE
        return
    else:
        pmq.type = BLACK
        for e in subedges:
            pmq.edges.append(e)
        return

def test():
    D = pickle.load(open('../data/mydcel.pickle'))
    XD = Xdcel(D)

    X = [v.x for v in D.vertices]
    Y = [v.y for v in D.vertices]
    xmin,xmax,ymin,ymax = min(X)-1, max(X)+1,\
                          min(Y)-1, max(Y)+1
    maxmax = max(xmax,ymax)
```

```
35      xmax=ymax=maxmax
36      extent = Extent(xmin, xmax, ymin, ymax)
37
38      pm3q = PMQuadTreeNode(extent.getcenter(), extent)
39      split_by_points(XD.vertices, pm3q)
40      split_by_edges_pm3(XD.edges, pm3q)
41      print search_pmquadtree(pm3q, 1, 1)
42
43 if __name__ == '__main__':
44      test()
```

Все PM-квадродеревья можно использовать для поиска точек и отрезков. Например, можно найти отрезок, ближайший к данной точке, спускаясь вниз по дереву и исключая по пути узлы, отстоящие далеко от точки. Но для поиска многоугольников необходима информация, связывающая каждое ребро со смежными с ним многоугольниками, которая хранится в объекте `Xdcel`. Мы не приводим здесь детали этих механизмов, но интересующийся читатель сможет найти полезные сведения в литературе, упоминаемой в конце главы.

7.2. R-деревья

PM-квадродеревья можно использовать для индексирования отрезков и косвенно многоугольников, ассоциированных с отрезками. Что же касается R-деревьев, то они специально предназначены для индексирования многоугольников с целью ускорить поиск и другие операции. Интересно, что ключевым понятием в описании R-деревьев является не сам многоугольник, а минимальный ограничивающий прямоугольник (minimal bounding rectangle – MBR), который можно использовать для представления многоугольника. На рис. 7.5 показаны MBR для трех многоугольников, *A*, *B* и *C* (соответственно R3, R4 и R5). Хотя MBR не содержит подробной геометрической информации о многоугольнике, он очень полезен для определения места расположения многоугольника, а также помогает снизить стоимость вычислений при выполнении таких операций, как поиск и наложение. Например, чтобы найти многоугольники, имеющие общую границу с данным, можно сравнить прямоугольники и понять, могут ли многоугольники в принципе соприкасаться. Сравнивать MBR быстрее, чем более сложные многоугольники. Такое сравнение, конечно, не дает точных результатов – например, прямоугольники R3 и R5 пересекаются, а их многоугольники (*A* и *C*) – нет. Но оно позволяет значительно сузить область поиска, сведя ее к очень небольшому количеству многоугольников, так что для получения окончательного ответа понадобится не слишком много вычислений. Благодаря использованию MBR для абстрагирования многоугольников мы получаем возможность игнорировать детали, связанные со сложной формой многоугольников, и сосредоточиться на существенных операциях.

Для индексирования MBR на рис. 7.5 с помощью R-дерева мы должны сначала определить, как выглядит узел такого дерева. В отличие от предыдущих

деревьев, в которых каждый узел содержал один объект, здесь узел может содержать несколько объектов, каждый из которых называется записью. Еще до создания R-дерева мы должны задать максимальное число записей в узле, оно обозначается *M*. В нашем простом примере для иллюстрации идей мы положили *M* равным 2. К R-дереву предъявляется еще одно требование – в каждом узле должно храниться некоторое минимальное число записей во избежание пустых узлов. Минимальное число записей обычно определяется как наименьшее целое число, большее или равное *M*/2, в нашем случае 1. Наконец, R-дерево сбалансировано, все исходные MBR должны храниться в его листьях, а глубина всех листьев должна быть одинакова.

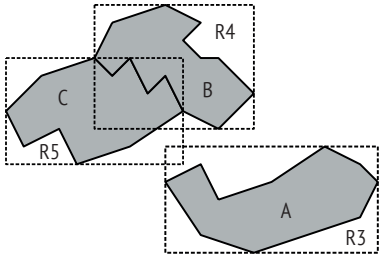

Рис. 7.5 ❖ Многоугольники
и их минимальные ограничивающие прямоугольники

Вот теперь мы готовы построить R-дерево для MBR (рис. 7.6). Создадим корневой узел дерева с двумя пустыми записями. Затем начнем вставлять MBR в корень. Предположим, что MBR вставляются в порядке R3, R4, R5. Первыми будут вставлены R3 и R4, и оба будут помещены в корень. Но при вставке R5 корневой узел переполнится, поэтому его придется разбить на два новых узла. При этом необходимо соблюсти два условия. Во-первых, оба новых узла должны иметь глубину, на единицу большую, чем у корня. В этот момент нужно будет поместить обе записи в два новых узла и решить, куда какую отправить. Оригинальный алгоритм R-деревьев говорит, что худшей комбинацией является пара прямоугольников, образующих наибольший новый ограничивающий прямоугольник, поэтому их следует помещать в разные узлы. Следовательно, когда в узлах еще имеются свободные записи, мы всегда помещаем MBR в тот узел, где он приведет к появлению наименьшего совокупного прямоугольника. В нашем примере разбиваемый узел содержит всего две записи, R3 и R4, так что каждая отправляется в новый узел. Во-вторых, необходимо правильно прописать связи между новыми узлами и корнем, а в записях в корне подкорректировать ограничивающие прямоугольники, потому что после разбиения в них может оказаться несколько меньших MBR. Каждая запись в корне теперь имеет новый MBR, который содержит объединение всех записей в одном из новых узлов. Мы ассоциируем с каждой записью указатель, который описывает такую связь между записью и дочерним узлом. С другой стороны, каждому дочернему узлу необходим указатель, который сообщает, какая запись в родительском узле охватывает этот дочерний узел. На рис. 7.6 эти указатели обозначены стрелками. Для

удобства две новые записи в корне обозначены R1 и R2, причем R1 содержит MBR R3, а R2 – R4.

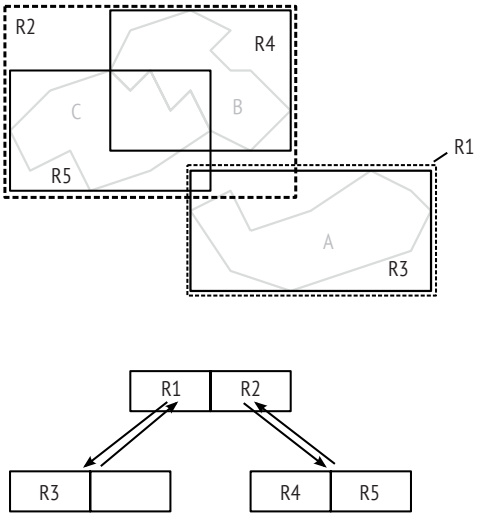

Рис. 7.6 ❖ Узлы и записи в R-дереве

После разбиения в дереве освобождается место для новых MBR. Теперь мы можем вставить наш последний MBR, R5, в дерево. На уровне корня имеется две записи, и мы должны решить, к какой из них отнести MBR. Для этого используется сформулированный выше принцип: мы хотим минимизировать увеличение ограничивающих прямоугольников на уровне корня после добавления нового MBR. Очевидно, что увеличение будет наименьшим, если вставить R5 в запись, содержащую R4. Мы спускаемся вниз по ветви, исходящей из R2, которая ведет в узел, где есть место для вставки новой записи. Помещаем туда R5 и обновляем ограничивающий прямоугольник R2, чтобы отразить добавление нового MBR. Теперь R-дерево принимает окончательный вид, показанный на рис. 7.6.

Для реализации алгоритма построения R-дерева нам нужны структуры данных для записи и узла. В классе `Entry` в листинге 7.7 (строка 3) имеется три члена: MBR записи в виде экземпляра класса `Extent`, ссылка на дочерний узел, записи которого охватываются этой записью, и ссылка на узел, содержащий эту запись. Класс узла называется `RTreeNode` (строка 12) и содержит четыре члена: список экземпляров класса `Entry`, максимальное число записей в этом узле, ссылку на запись в родительском узле, содержащую этот узел, и экстент узла в виде объекта `Extent`. Помимо данных-членов, в класс включено несколько полезных функций-членов. Функция `__getitem__` переопределяет встроенную функцию Python и обеспечивает возможность итерирования по записям в узле. Вспомогательные функции `is_leaf` и `is_root` удобны при написании кода. Наиболее важны функции `update` и `update_up`. Функция `update` отвечает за то, чтобы экстент узла был равен объединению MBR всех его записей; она вызывает другую функцию `union_extent`, которая определена

в строке 52. Функция update_up гарантирует, что изменение в данном узле распространится по всему пути до корня.

Листинг 7.7 ❖ Структуры данных записи и узла R-дерева (rtree1.py)

```
 1 from extent import *
 2
 3 class Entry():
 4     def __init__(self, extent=None, child=None,
 5                  parent=None, node=None):
 6         self.MBR = extent
 7         self.child = child           # дочерний узел
 8         self.node = node             # узел, содержащий данную запись
 9     def __repr__(self):
10         return str(self.MBR)
11
12 class RTreeNode():
13     def __init__(self, M, parent=None):
14         self.entries = []
15         self.M = M
16         self.parent = parent         # запись в родительском узле
17         self.extent = None
18     def __getitem__(self, i):
19         if i>=self.M or i>=len(self.entries):
20             return None
21         return self.entries[i]
22     def __repr__(self):
23         return str(self.extent)
24     def is_leaf(self):
25         for e in self.entries:
26             if e.child is not None:
27                 return False
28         return True
29     def is_root(self):
30         return self.parent is None
31     def update(self):
32         if not len(self.entries):
33             return
34         if self.entries[0] is not None:
35             self.extent = self.entries[0].MBR
36         for e in self.entries[1:]:
37             self.extent = union_extent(self.extent, e.MBR)
38     def update_up(self):
39         self.update()
40         if self.is_root():
41             return
42         self.parent.MBR = self.extent
43         self.parent.node.update_up()
44     def get_all_leaves(self, depth=0):
45         if not self.is_leaf():
46             for e in self.entries:
47                 e.child.get_all_leaves(depth+1)
```

```
48         else:
49             print depth, "-", self, self.entries
50             return
51
52 def union_extent(e1, e2):
53     xmin = min(e1.xmin, e2.xmin)
54     xmax = max(e1.xmax, e2.xmax)
55     ymin = min(e1.ymin, e2.ymin)
56     ymax = max(e1.ymax, e2.ymax)
57     return Extent(xmin, xmax, ymin, ymax)
```

В листинге 7.8 приведена функция `insert`, которая вставляет новый MBR (`e`) в R-дерево. Сначала проверяется, что вставляемого экстента еще нет в дереве (строка 5). Если в узле еще есть место для нового MBR (строка 9), то мы просто добавляем новую запись в список записей, обновляем все узлы дерева, связанные с этой записью, и на этом вставка завершается. Если же узел уже заполнен, то мы создаем два новых узла (строки 18 и 19), а затем в строках 20–34 находим в текущем узле два MBR, которые дальше всего отстоят друг от друга. Эти два MBR станут родоначальниками в новых узлах (строки 39 и 40). В цикле `while`, начинающемся в строке 43, мы продолжаем вставлять остальные MBR в каждый из новых узлов, так чтобы увеличение общей площади MBR в каждом узле было минимально. После того как оба новых узла будут готовы, нужно расщепить старый узел, воспользовавшись функцией `split`.

Функция `split` принимает текущий узел и оба только что созданных новых узла. Сначала мы создаем две пустые записи, пользуясь экстентами новых узлов (строки 106 и 107). Если расщепляется корневой узел (строка 108), то мы удаляем записи в корне и повторно используем его с новыми записями. В противном случае один из новых узлов заменяет текущий, а другой новый узел вставляется в родительский с помощью функции `insert`. Эта вставка может привести к следующему расщеплению узлов. На протяжении данного процесса мы следим за тем, чтобы все MBR находились в листьях дерева.

Есть много способов поиска в R-дереве. Ниже приведен пример поиска листового узла, для которого коэффициент перекрытия с заданным экстентом максимален. Алгоритм, реализованный в функции `search_rtree_extent`, спускается по дереву, выбирая запись с наибольшей площадью пересечения, пока не дойдет до листового узла. Позже мы воспользуемся этой функцией для построения R-дерева.

Листинг 7.8 ❖ Вставка и расщепление узла в R-дереве (rtree2.py)

```
1 from rtree1 import *
2
3 # e - экстент
4 def insert(node, e, child=None):
5     for ent in node.entries:              # уже присутствует в дереве
6         if ent.MBR == e:
7             return True
8     entry = Entry(extent=e, child=child)  # создать новую запись
9     if len(node.entries) < node.M:        # место есть
```

```
        entry.node = node
        if entry.child is not None:
            entry.child.parent = entry
        node.entries.append(entry)
        node.update_up()
        return True
    M = node.M                          # узел заполнен и должен быть расщеплен
    m = math.ceil(float(M)/2)
    L1 = RTreeNode(M)
    L2 = RTreeNode(M)
    maxi, maxj = -1, -1
    maxdist = 0.0
    tmpentries = [ent for ent in node.entries]
    tmpentries.append(entry)
    M1 = len(tmpentries)
    # найти два MBR, дальше всего отстоящих друг от друга
    for i in range(M1):
        for j in range(i+1, M1):
            d = tmpentries[i].MBR.distance(tmpentries[j].MBR)
            if d>maxdist:
                maxdist = d
                maxi = i
                maxj = j
    e1 = tmpentries[maxi]
    e2 = tmpentries[maxj]
    allexts = []                        # остальные MBR
    for ext in tmpentries:
        if ext is not e1 and ext is not e2:
            allexts.append(ext)
    L1.entries.append(e1)
    L2.entries.append(e2)
    L1.update()
    L2.update()
    while len(allexts):
        numremained = len(allexts)
        gotonode = None
        if len(L1.entries) == m-numremained:
            gotonode = L1
        elif len(L2.entries) == m-numremained:
            gotonode = L2
        if gotonode is not None:
            while len(allexts):
                ext = allexts.pop()
                gotonode.entries.append(ext)
        else:
            minarea = union_extent(L1.extent,L2.extent).area()
            minext = -1
            gotonode = None
            for i in range(len(allexts)):
                tmpext1 = union_extent(L1.extent,
                                       allexts[i].MBR)
                tmparea1 = tmpext1.area() - L1.extent.area()
```

```
                tmpext2 = union_extent(L2.extent,
                                       allexts[i].MBR)
                tmparea2 = tmpext2.area() - L2.extent.area()
                if min(tmparea1, tmparea2) > minarea:
                    continue
                minext = i
                if tmparea1 < tmparea2:
                    if tmparea1 < minarea:
                        tmpgotonode = L1
                        minarea = tmparea1
                elif tmparea2 < tmparea1:
                    if tmparea2 < minarea:
                        tmpgotonode = L2
                        minarea = tmparea2
                else:
                    minarea = tmparea1
                    if L1.extent.area() < L2.extent.area():
                        tmpgotonode = L1
                    elif L2.extent.area() < L1.extent.area():
                        tmpgotonode = L2
                    else:
                        if len(L1.entries) < len(L2.entries):
                            tmpgotonode = L1
                        else:
                            tmpgotonode = L2
            if minext <> -1 and tmpgotonode is not None:
                ext = allexts.pop(minext)
                gotonode = tmpgotonode
            gotonode.entries.append(ext)
        gotonode.update()
        for ent in L1.entries:
            ent.node = L1
            if ent.child is not None:
                ent.child.parent = ent
        for ent in L2.entries:
            ent.node = L2
            if ent.child is not None:
                ent.child.parent = ent
    split(node, L1, L2)
    L1.update_up()
    L2.update_up()
    return True

def split(node, L1, L2):
    entry1 = Entry(L1.extent)
    entry2 = Entry(L2.extent)
    if node.is_root():
        node.entries = []
        entry1.node = node
        entry2.node = node
        entry1.child = L1
        entry2.child = L2
        node.entries.append(entry1)
```

```
115         node.entries.append(entry2)
116         L1.parent = entry1
117         L2.parent = entry2
118         return
119     else:
120         entry1.node = L1
121         L1.parent = node.parent
122         L1.parent.child = L1
123         del node
124         insert(L1.parent.node, L2.extent, L2)
125         return
126 
127 def search_rtree_extent(node, e):
128     if node.is_leaf():
129         return node
130     best_entry = None
131     intersect_area = -1
132     for ent in node.entries:
133         tmp_area = ent.MBR.intersect(e)
134         if tmp_area > intersect_area:
135             intersect_area = tmp_area
136             best_entry = ent
137     return search_rtree_extent(best_entry.child, e)
```

На рис. 7.7 показано десять MBR, которые мы проиндексируем с помощью R-дерева. Интересно отметить, что при использовании MBR не требуется, чтобы многоугольники были топологически правильными. Иными словами, два многоугольника могут пересекаться, не имея точек пересечения. Так, R15 и R17 на этом рисунке пересекаются, что означает, что соответствующие им многоугольники пересекаются. Возможно также, что некоторым MBR соответствуют многоугольники, состоящие из нескольких несвязных частей. Такие ситуации встречаются нечасто, но в некоторых пространственных наборах данных имеют место, и R-дерево с ними справляется.

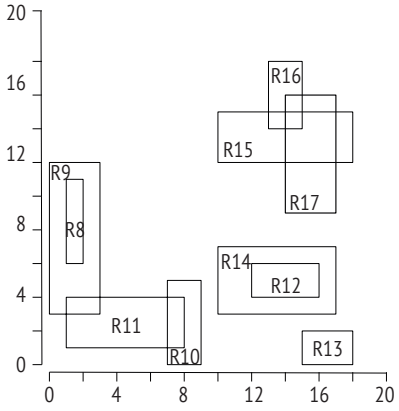

Рис. 7.7 ❖ Минимальные ограничивающие прямоугольники, использованные для создания R-дерева

В листинге 7.9 показано, как создать R-дерево по данным на рис. 7.7. MBR хранятся в списке MBRs, причем порядок прямоугольников случаен. Мы преобразуем MBRs в список объектов Extent (строка 18). Корневой узел создается в строке 16. Экстенты вставляются в дерево один за другим (строка 20), причем всякий раз вставка производится в узел с наибольшим коэффициентом перекрытия. Чтобы вернуть все листовые узлы вместе с их глубиной, мы пользуемся простой функцией get_all_leaves.

Листинг 7.9 ❖ Использование R-дерева (use_rtree.py)

```
 1 from rtree1 import *
 2 from rtree2 import *
 3
 4 MBRs = [ [10,17,3,7],           # R14
 5          [12,16,4,6],           # R12
 6          [7,9,0,5],             # R10
 7          [1,8,1,4],             # R11
 8          [1,2,6,11],            # R8
 9          [15,18,0,2],           # R13
10          [0,3,3,12],            # R9
11          [13,15,14,18],         # R16
12          [10,18,12,15],         # R15
13          [14,17,9,16] ]         # R17
14
15 M = 3
16 root = RTreeNode(M, None)
17
18 extents = [Extent(mbr[0], mbr[1], mbr[2], mbr[3])
19            for mbr in MBRs]
20 for e in extents:
21     n = search_rtree_extent(root, e)
22     insert(n, e)
23
24 root.get_all_leaves()
```

При выполнении тестового кода в листинге 7.9 получается такой результат:

```
2 - [1, 9, 0, 5] [[1, 8, 1, 4], [7, 9, 0, 5]]
2 - [0, 3, 3, 12] [[0, 3, 3, 12], [1, 2, 6, 11]]
2 - [13, 17, 9, 18] [[13, 15, 14, 18], [14, 17, 9, 16]]
2 - [12, 18, 0, 6] [[15, 18, 0, 2], [12, 16, 4, 6]]
2 - [10, 18, 3, 15] [[10, 17, 3, 7], [10, 18, 12, 15]]
```

Видно, что все листовые узлы имеют одинаковую глубину, как и должно быть в R-дереве. На рис. 7.8 показано, как сгруппированы MBR и как выглядит соответствующее R-дерево.

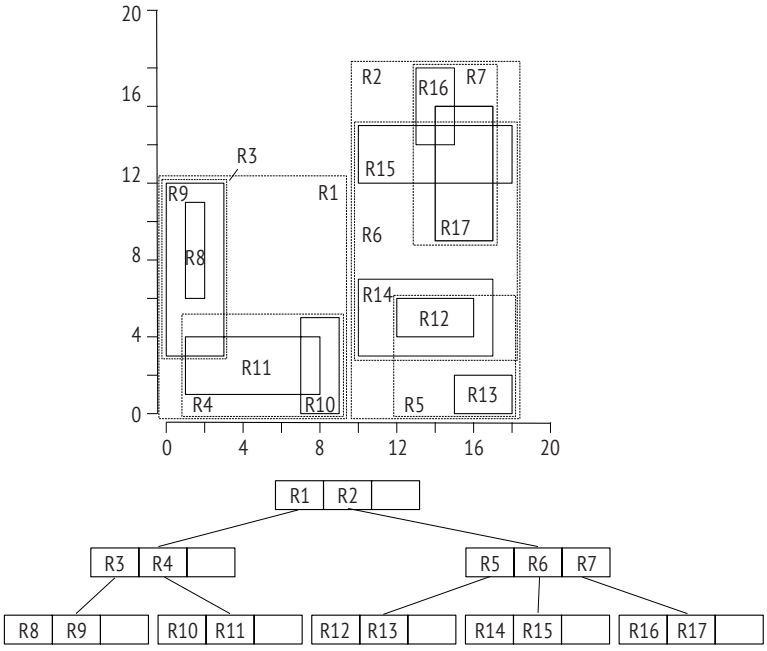

Рис. 7.8 ❖ Группировка минимальных ограничивающих прямоугольников

7.3. Примечания

Идея PM-квадродеревьев впервые была выдвинута в работе Samet and Webber (1985). Из трех вариантов PM-квадродеревьев PM1 требует максимального объема памяти из-за применяемого представления ребер. Но и в PM3 может храниться слишком много точек и отрезков в каждом узле, что увеличивает время вычислений, потому что в процессе поиска нужно проверять все ребра, ассоциированные с каждым квадрантом. PM2 представляется сбалансированным компромиссом, полезным для полигональных карт.

Идею квадродеревьев можно обобщить на большее число измерений. Например, для хранения точек в трехмерном пространстве применяются октодеревья (Meagher, 1980, 1982); пространство разбивается на восемь частей (октантов), и разбиение продолжаеттся, пока выполняется некоторое условие (например, существуют октанты, содержащие более одной точки). Дополнительные сведения об октодеревьях можно найти в работе (2005) и в книгах Samet (1990a,b, 2006).

R-дерево впервые предложено в работе Guttman (1984) как способ хранения и индексирования информации о многоугольниках. В нашем обсуждении R-деревьев речь шла в основном о геопространственных данных, но эту идею можно использовать и в более широком контексте, когда нужно

быстро искать объекты. Например, основной областью применения этого метода индексирования является компьютерная графика. Ключевые особенности R-деревьев, вставка и расщепление узла, похожи на встречающиеся в B-деревьях (Bayer and McCreight, 1972) – основном методе индексирования упорядоченных объектов (например, целых чисел или вообще любых объектов, которые можно расположить по порядку). Обсуждение B-деревьев выходит за рамки этой книги.

Как и в случае *k*D-деревьев, построение R-дерева может обходиться дорого. В общем случае временная сложность вставки n MBR в дерево составляет $O(n)$. Но поиск в R-дереве очень эффективен, время работы составляет $O(M\log_M n)$, где M – максимальное число записей в узле. В этой главе мы не обсуждали другие операции. Например, иногда нужно удалять MBR из дерева, в результате чего структура дерева перестраивается. Детали этой и других операций можно найти в книгах (Manolopoulos et al., 2006; Samet, 2006).

Обсуждение в этой главе вскрыло некоторые недостатки R-деревьев. Их преодоление – предмет активных исследований. Например, очевидно, что выбор двух «родоначальников» при расщеплении узла может оказать существенное влияние на общую структуру дерева, а значит, и на эффективность его использования. Были предложены другие алгоритмы и варианты R-дерева. Например, в гильбертовом R-дереве применяются кривые Гильберта, заполняющие пространство, для разбиения пространства и группировки MBR (Kamel and Faloutsos, 1994). Предложены также алгоритмы для борьбы с основным недостатком R-деревьев: перекрытием MBR. Например, R^+-дерево (Sellis et al., 1987) предназначено для избегания перекрытий, а R^*-дерево (Beckmann et al., 1990) – для минимизации перекрытия[1].

7.4. Упражнения

1. Проанализируйте код точечно-регионных *k*D-деревьев и замените `xyrange` классом `Extent`. Соответственно модифицируйте код операций в дереве.
2. Нарисуйте PM2- и PM3-квадродеревья для разбиения пространства, показанного на рис. 7.3 и 7.4.
3. Спроектируйте и реализуйте алгоритм поиска в R-дереве и найдите MBR, содержащий заданную точку.

[1] Отметим, что «R-дерево» читается не «R минус дерево», а просто «R дерево».

Часть III

ПРОСТРАНСТВЕННЫЙ АНАЛИЗ И МОДЕЛИРОВАНИЕ

Глава 8

Интерполяция

Повсеместное проникновение геопространственных данных в век «больших данных» создает ложное ощущение полного покрытия мира наблюдениями. Иными словами, мы ошибочно думаем, будто знаем всё о каждой точке заменой поверхности. Доказательство: зайдите на сайт maps.google.ru и выберите вид «Спутник». Создается впечатление, что мы можем посмотреть на любую точку с высоты птичьего полета (рис. 8.1). Впрочем, у этого всезнания есть и обратная сторона: все мы слышали, что совершенная ракета может нанести удар чуть ли не в любое место, поскольку имеются точные модели рельефа, которыми ракета руководствует на пути к цели.

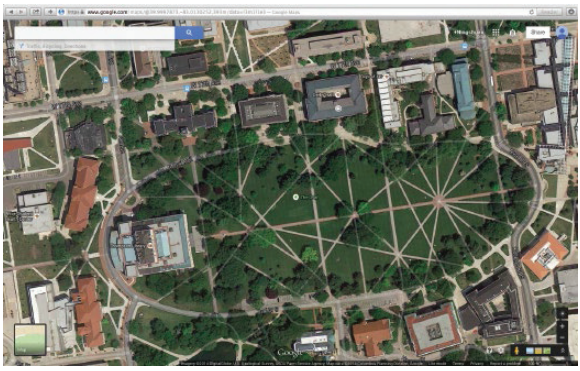

Рис. 8.1 ❖ Часть кампуса Университета штата Огайо на спутниковом представлении карт Google

Чтобы развеять тайны, которые окутывают поражающее воображение изобилие данных, следует понимать одно фундаментальное понятие, встречающееся в географии и многих других физических науках: масштаб. В географической литературе у термина «масштаб» есть несколько значений. Применительно к картам под масштабом понимается отношение между расстоянием на карте и на местности. Иногда можно услышать выражение «крупномасштабный» проект, это означает, что работа широка по охвату или велика по протяженности. Помимо этих двух значений, у концепции масштаба есть и другие смыслы при употреблении в контексте измерений или операционной деятельности. Например, на первом же занятии по системам для атмосферных наблюдений вы узнаете о различии между погодой и кли-

матом. Мы можем следить за погодным каналом, но не за климатическим каналом (если бы такая штука вообще существовала), потому что первый относится к атмосферным условиям за сравнительно короткий период времени (несколько суток) и небольшому участку пространства (регион), тогда как под климатом понимается динамика в более длительной перспективе (сезон, год и дольше) и на большей территории (например, глобально). Поэтому ясно, что движение воздуха в атмосфере в целом, с одной стороны, и ураганы и землетрясения, с другой стороны, – явления разного масштаба, для описания которых нужна разная терминология.

Четвертый аспект масштаба интересует нас больше всего: масштаб измерений, т. е. наименьшая различимая часть наблюдаемого объекта. Например, фотограф располагает лишь конечным числом пикселей, каждый из которых должен представлять какую-то часть реальности, запечатленной на фотографии. Две вещи ограничивают детальность фотографии: количество физических пикселей и размер наименьшего объекта, распознаваемого устройством, с помощью которого сделана фотография. Хотя то и другое можно улучшить благодаря технологическим достижениям, никуда не деться от того факта, что реальность описывается бесконечным объемом информации, которую мы хотим уместить в конечное число пикселей. Что, разумеется, невозможно. А раз так, приходится идти на компромисс – попытаться произвести выборку, которая могла бы служить заменой реальности. Тогда образуется два типа реальности: известная, для которой у нас есть наблюдения, и неизвестная, для которой наблюдений нет. А что делать, если мы хотим получить результаты измерений в точках, где наблюдений нет? Высказать гипотезу. Иначе говоря, довольствоваться оценкой.

В разных прикладных областях оценка производится по-разному, и само пространство трактуется по-разному. В случае обсуждавшегося ранее дистанционного зондирования мы говорим, что в области, покрытой одним пикселем, все результаты измерений одинаковы. Для других измерений, например рельефа, мощности почвы и температуры, пространство описывается расстояниями от известных точек до неизвестных, как показано на рис. 8.2. И нам нужны результаты измерений в точках, для которых есть наблюдения (черные кружки), чтобы оценить результаты в точках, где наблюдения отсутствуют (белый кружок).

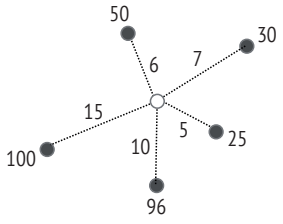

Рис. 8.2 ❖ Интерполяция. Черными кружками представлены точки, в которых результаты измерений известны и показаны числами. Белый кружок представляет точку, для которой результата измерения нет, поэтому значение необходимо оценивать. Расстояния между неизвестной точкой и известными показаны числами рядом с пунктирными линиями

Техника оценивания значений в точках, для которых нет наблюдений, называется интерполяцией. Картографам и землемерам эта техника давно знакома. В истории картографии и землеустроения целью всегда было точное измерение координат и высот точек рельефа. Традиционно эта задача решалась путем обустройства на местности геодезических опорных сетей, каждый узел которых представляет точку наблюдения. На основе этих сетей можно оценить высоту рельефа в других точках. Можно поставить и обратную задачу: имея наблюдения в некоторых точках, оценить, где будет наблюдаться заданное значение. Эта техника очень полезна для создания топографических карт, на которых присутствует сетка изолиний, соединяющих точки с предположительно одинаковыми значениями некоторой величины (рис. 8.3). В этой главе мы познакомимся с двумя хорошо себя зарекомендовавшими методами интерполяции: методом обратных взвешенных расстояний и кригингом. Мы также обсудим метод имитационного моделирования, позволяющий генерировать поверхность, которая выглядит «как настоящая».

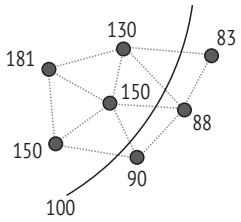

Рис. 8.3 ❖ Интерполяции изолинии, соединяющей точки со значением 100

8.1. Метод обратных взвешенных расстояний

Пусть имеется набор наблюдений некоторой величины, скажем высоты над уровнем моря, и мы хотим воспользоваться им для оценки этой величины в некоторой точке. Как поступить? В общем случае мы можем назначить каждому наблюдаемому значению некоторый вес, а затем использовать веса для вычисления оценки. Формально, если дан набор n точек, в которых известны значения, то обозначим $z(x_i)$ результат измерения в i-й точке x_i, а λ_i – вес этого измерения. Тогда неизвестное значение в точке x можно записать в виде

$$z(x) = \frac{\sum_i \lambda_i z(x_i)}{\sum_i \lambda_i}.$$

Отметим, что знаменатель можно опускать, если есть гарантия, что сумма всех весов равна 1 ($\sum_i \lambda_i = 1$).

Интересен вопрос о том, как назначать веса разным точкам. Хотя это можно делать и произвольным образом, разумнее, когда близким точкам назначаются более высокие веса. В конце концов, если действительно существует

что-то, влияющее на значение в данной точке, то мы скорее будем доверять точкам поблизости от нее, чем отстоящим далеко (так, высота или температура в Калифорнии вряд ли будут столь же эффективно предсказывать значения этих величин в Колумбусе, штат Огайо, как точки поблизости от центра Огайо). Эта вера легла в основу первого закона географии, сформулированного Тоблером. Правда, многие исследователи сомневаются в том, что утверждение Тоблера можно назвать законом, но мы будем считать его полезной рекомендацией. Оно гласит, что «всё влияет на всё, но то, что ближе, влияет сильнее».

Метод обратных взвешенных расстояний (ОВР, англ. IDW) применяется именно так, как рекомендует первый закон географии, поскольку близким точкам назначаются большие веса, чем отдаленным. Точнее, вес, назначаемый точке с известным результатом наблюдения, уменьшается с ростом расстояния до точки, в которой производится оценка:

$$\lambda_i = \frac{1}{d_i^b},$$

где d_i – расстояние между i-м наблюдением и интересующей нас точкой, а b – константа.

Код в листинге 8.1 реализует метод ОВР. Функция `IDW` принимает два аргумента: известные данные `Z` и степень интерполяции `b`. Здесь `Z` – список, каждый элемент которого является списком из четырех элементов: координаты X и Y, значение в точке с такими координатами и расстояние от этой точки до интерполируемой (см. ниже тест, в котором эти данные задаются).

Листинг 8.1 ❖ Интерполяция методом обратных взвешенных расстояний (idw.py)

```
def IDW(Z, b):
    """
    Интерполяция методом обратных взвешенных расстояний.
    Вход
      Z: список списков, каждый элемент которого содержит четыре
         значения: X, Y, значение и расстояние до интерполируемой
         точки. Z может быть также двумерным массивом NumPy.
      b: степень расстояния
    Выход
      Оценка значения в интерполируемой точке.
    """
    zw = 0.0                    # взвешенная сумма z
    sw = 0.0                    # сумма весов
    N = len(Z)                  # число точек данных
    for i in range(N):
        d = Z[i][3]
        if d == 0:
            return Z[i][2]
        w = 1.0/d**b
        sw += w
        zw += w*Z[i][2]
    return zw/sw
```

Для демонстрации мы используем 100 значений высоты в северо-западной части Колумбуса, штат Огайо (рис. 8.4). Этот же набор данных будет использован в следующем разделе при обсуждении кригинга. Для тестирования метода OBP мы сначала загружаем данные из текстового файла. Поскольку в этой главе нам предстоит читать данные несколько раз, мы написали специальную функцию для этой цели `read_data` (см. листинг 8.2). Эта функция просто считывает каждую строку в список (строка 12), а затем выделяет из нее элементы с помощью библиотечной функции Python для обработки строк `split`. Это делается с помощью механизма спискового включения, и результатом является список списков (строка 13). Однако элементами внутренних списков являются строки, которые нужно преобразовать в числа с плавающей точкой (строка 14).

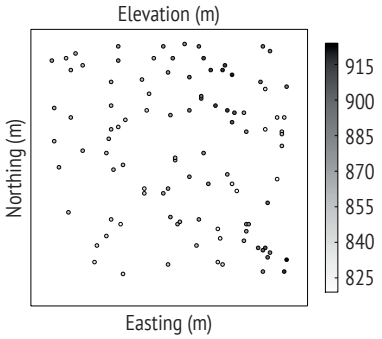

Рис. 8.4 ❖ Тестовый набор данных с результатам измерения высоты. Длины измерены в метрах

Листинг 8.2 ❖ Чтение данных из текстового файла (read_data.py)

```
1 def read_data(fname):
2     """
3     Читает данные из файла. В каждой строке должно быть три столбца:
4     X, Y и Значение.
5     Вход
6         fname: путь к файлу
7     Выход
8         x3: список списков размерности 3×n
9         Каждый внутренний список содержит 3 элемента: X, Y, Значение
10    """
11    f = open(fname, 'r')
12    x1 = f.readlines()
13    x2 = [x.strip().split() for x in x1]
14    x3 = [[float(x[0]),float(x[1]),float(x[2])] for x in x2]
15    return x3
```

Следующая важная задача – подготовка данных для интерполяции. Мы используем функцию в листинге 8.3 для создания списка данных, передаваемого функции `IDW`. Вычисляется расстояние между каждой известной точкой

и интерполируемой точкой, для этого достаточно одной строки (строка 5), где применяется механизм спискового включения Python. В строке 7 готовится единый список данных, каждый элемент которого соответствует одной точке и содержит ее координаты (*X* и *Y*), наблюдаемое значение и расстояние до интерполируемой точки. Затем список сортируется по расстоянию (строка 8) и выбираются десять точек, ближайших к интерполируемой (строка 9). Помимо `N` точек, функция еще возвращает среднее значение данных, которое понадобится нам ниже в этой главе.

Листинг 8.3 ❖ Подготовка данных для интерполяции (prepare_interpolation_data.py)

```
 1 from math import sqrt
 2 def prepare_interpolation_data(x, Z, N=10):
 3     vals = [z[2] for z in Z]
 4     mu = sum(vals)/len(vals)
 5     dist = [sqrt((z[0]-x.x)**2 + (z[1]-x.y)**2) for z in Z]
 6     Z1 = [(Z[i][0], Z[i][1], Z[i][2], dist[i])
 7           for i in range(len(dist))]
 8     Z1.sort(key=lambda Z1: Z1[3])
 9     Z1 = Z1[:N]
10     return Z1, mu
```

Мы поместили этот код в отдельный файл, чтобы можно было воспользоваться им при тестировании функции `IDW` в листинге 8.4. Для тестирования интерполяции взята точка (337000, 4440911) (строка 11).

Листинг 8.4 ❖ Тестирование интерполяции методом обратных взвешенных расстояний (test_idw.py)

```
 1 import sys
 2 sys.path.append('../geom')
 3 from point import *
 4 from idw import *
 5 from read_data import *
 6 from math import sqrt
 7 from prepare_interpolation_data import *
 8
 9 Z = read_data('../data/necoldem.dat')
10
11 x = Point(337000, 4440911)
12
13 N = 10
14
15 Z1 = prepare_interpolation_data(x, Z, N)[0]
16
17 print 'power=0.0:', IDW(Z1, 0)
18 print 'power=0.5:', IDW(Z1, 0.5)
19 print 'power=1.0:', IDW(Z1, 1.0)
20 print 'power=1.5:', IDW(Z1, 1.5)
21 print 'power=2.0:', IDW(Z1, 2.0)
```

Взяв четыре разных значения степени, получаем следующие оценки:

```
power=0.0: 858.8
power=0.5: 859.268444166
power=1.0: 859.737719755
power=1.5: 859.951544335
power=2.0: 859.998590795
```

Видно, что результат зависит от значения b (степень расстояния). Так какое же значение использовать? Формула для вычисления весов простая, но неопределенная, потому что мы не знаем, какое b выбрать. И как же его определить? Без дополнительного исследования – никак. Вообще говоря, чем больше степень расстояния, тем больший вес назначается близким точкам. Если b равно нулю, то вес любой точки равен 1, поэтому мы просто берем в качестве оценки неизвестного значения среднее арифметическое значений в окружающих точках. При ненулевом значении b расстояние влияет на результат интерполяции. На рис. 8.5 показано, как именно степень расстояния влияет на результирующие веса. Для нахождения оптимального значения b нужно попробовать отыскать такое, при котором качество интерполяции максимально. Обычно качество измеряется среднеквадратической ошибкой (СКО, англ. RMSE):

$$\mathrm{RMSE} = \left(\frac{1}{n} \sum_j (p_j - o_j)^2 \right)^{1/2},$$

где p_j – оценка в точке j ($1 \leqslant j \leqslant n$), а o_j – наблюдаемое значение в точке j. Традиционный способ контроля качества заключается в том, чтобы зарезервировать часть данных, не участвующую в интерполяции. На этой части затем будет оцениваться качество результатов интерполяции.

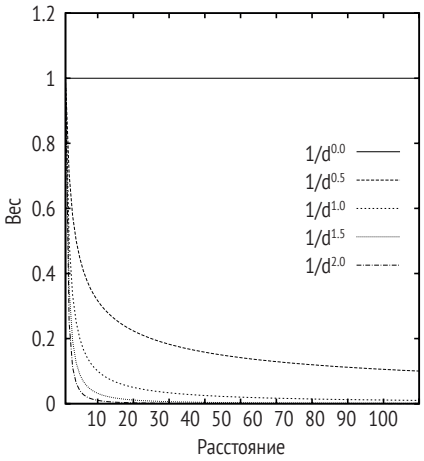

Рис. 8.5 ❖ Влияние степени расстояния на интерполяцию методом ОВР

Алгоритм ОВР позволяет делать еще две вещи: протестировать влияние степени расстояния и использовать наилучшее значение для построения поверхности. Первую задачу мы решим сейчас, а вторую отложим до следующего раздела.

Для тестирования влияния степени расстояния возьмем одну из известных точек и оценим значение в ней с помощью функции IDW. Проделаем это для всех известных точек с данным значением степени и вычислим для этого значения СКО. Затем простое сравнение даст нам оптимальное значение b.

Листинг 8.5 ❖ Перекрестная проверка для метода ОВР (idw_cross_validation.py)

```
 1 import sys
 2 sys.path.append('../geom')
 3 from point import *
 4 from idw import *
 5 from read_data import *
 6 from math import sqrt
 7 from prepare_interpolation_data import *
 8
 9 Z = read_data('../data/necoldem.dat')
10 N = len(Z)
11 numNeighbors = 10
12 mask = [True for i in range(N)]
13 powers = [0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5]
14 test_results = []
15 for i in range(N):
16     mask[i] = False
17     x = Point(Z[i][0], Z[i][1])
18     P = [ Z[j] for j in range(N) if mask[j] == True]
19     P1 = prepare_interpolation_data(x, P, numNeighbors)[0]
20     diff = []
21     for n in powers:
22         zz = IDW(P1, n)
23         diff.append(zz-Z[i][2])
24     test_results.append(diff)
25     mask[i] = True
26
27 for i in range(len(powers)):
28     rmse = sqrt(sum([r[i]**2 for r in test_results])/
29                 len(test_results))
30     print rmse, '[', powers[i], ']'
```

Код в листинге 8.5 можно использовать для перекрестной проверки степени расстояния b. Массив масок (строка 16) используется для того, чтобы на каждой итерации выбирать одну наблюдаемую точку. Хорошо видно, что СКО уменьшается с ростом степени расстояния, пока не достигает минимального значения при $b = 1.5$.

```
14.8683220304 [ 0 ]
14.1634839958 [ 0.5 ]
13.5663662616 [ 1 ]
```

```
13.2774718585 [ 1.5 ]
13.3066095177 [ 2 ]
13.5193354447 [ 2.5 ]
13.7913037092 [ 3 ]
14.060307198 [ 3.5 ]
14.3081392974 [ 4 ]
14.5338050919 [ 4.5 ]
14.7398249928 [ 5 ]
```

8.2. Кригинг

Метод обратных взвешенных расстояний, описанный в предыдущем разделе, может дать оценку значения интересующей нас величины в данной точке. Однако, как мы видели, существует произвол в выборе значения степени расстояния b. Пусть даже метод перекрестной проверки позволяет найти оптимальное значение, при котором ошибка минимальна, хотелось бы иметь способ, который дает какое-то теоретическое обоснование качества интерполяции. Эту проблему решает метод кригинга (kriging). В литературе его даже называют методом оптимальной интерполяции, в основном потому, что в нем используется теоретическая модель ошибки.

8.2.1. Полудисперсия

Основная идея кригинга – связать дисперсию наблюдаемых значений с расстоянием до соответствующих точек. Определим так называемую полудисперсию:

$$\gamma(h) = \frac{1}{2}E([z(x+h) - z(x)]^2)$$
$$= \frac{1}{2N}\sum_{i=1}^{N}[z(x_i + h) - z(x_i)]^2,$$

где x – точка, $x + h$ – другая точка на расстоянии h от x, $z(x)$ и $z(x + h)$ – значения в этих точках, E обозначает среднее, или математическое ожидание (в данном случае – квадратов разностей), N – количество пар наблюдений, где каждая пара состоит из двух точек на расстоянии h, и (x_i) – i-я позиция среди N пар. В литературе по кригингу h часто называют лагом. Величина $\gamma(h)$ называется *полу*дисперсией, потому что равна половине математического ожидания квадратов разностей. На рис. 8.6 показана полудисперсия набора данных, который будет использован ниже в этой главе. Обратите внимание на общий тренд $\gamma(h)$ при возрастании лага (h).

Для вычисления $\gamma(h)$ необходимо найти пары наблюдений, находящиеся на расстоянии h друг от друга. Но на практике, если только наблюдения не производятся систематически, так что расстояния между ними контролируются, найти пары наблюдений, отстоящие друг от друга в точности на h, трудно. Чтобы решить эту проблему, мы можем воспользоваться интервалами рас-

стояний, так что пары, попадающие в один интервал, будут учитываться при вычислении полудисперсии среднего расстояния для этого интервала. Проиллюстрируем вычисление полудисперсии на нашем тестовом наборе данных.

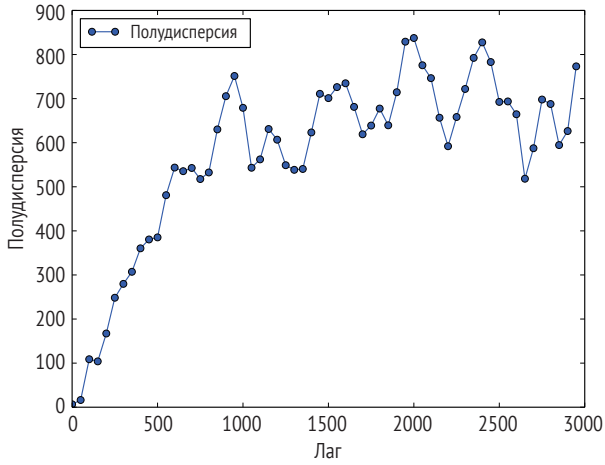

Рис. 8.6 ❖ Эмпирическая полудисперсия, вычисленная по тестовому набору данных

Программа в листинге 8.6 возвращает двумерный массив, содержащий полудисперсию для каждого значения лага[1]. Функция `semivar` принимает три параметра: двумерный массив данных, в котором каждый элемент является массивом, содержащим координаты X и Y и результат измерения, массив лагов и полудлину интервала расстояний. Здесь мы используем несколько очень полезных средств из популярного Python-модуля NumPy для численного анализа (его основы приведены в приложении А). Модуль NumPy позволяет очень просто выполнять операции над массивами и матрицами – в смысле выразительности кода, а не собственно вычислений. Мы еще не раз воспользуемся этими средствами, но убедительный пример имеется уже в строке 30, где вычисляется матрица расстояний для всех пар наблюдений во входных данных. Результатом является матрица $n \times n$, где n – число точек данных. Для каждого лага мы берем все пары точек, расстояния между которыми попадают в каждый интервал (строка 35), и помещаем квадраты разности между наблюдениями в этих точках в пустой список. Затем вычисляем среднее по этим парам (строка 40), чтобы получить полудисперсию для заданного значения лага. Наконец, лаг и соответствующая ему полудисперсия объединяются в пару (строка 44), и возвращается список этих пар.

[1] Python-код для кригинга, в частности вычисление полудисперсии, ковариации и простого кригинга, основан на коде из проекта https://github.com/cjohnson318/geostatsmodels и подвергнут модификации.

Листинг 8.6 ❖ Вычисление полудисперсии (semivariance.py)

```
 1 import numpy as np
 2 from math import sqrt
 3
 4 def distance(a, b):
 5     """
 6     Вычисляет расстояние между точками a и b
 7     Вход
 8       a: список [X, Y]
 9       b: список [X, Y]
10     Выход
11       Расстояние между a и b
12     """
13     d = (a[0]-b[0])**2 + (a[1]-b[1])**2
14     return sqrt(d)
15
16 def semivar(z, lags, hh):
17     """
18     Вычисляет эмпирическую полудисперсию по данным
19     Вход:
20       z - список или двумерный массив NumPy,
21           каждый элемент которого является списком X, Y, Значение
22       lags - одномерный массив значений лагов
23       hh - половина длины интервала расстояний
24     Выход:
25       Двумерный массив [ [h, gamma(h)], ...]
26     """
27     semivariance = []
28     N = len(z)
29     D = [[distance(z[i][0:2], z[j][0:2])
30           for i in range(N)] for j in range(N)]
31     for h in lags:
32         gammas = []
33         for i in range(N):
34             for j in range(N):
35                 if D[i][j] >= h-hh and D[i][j]<=h+hh:
36                     gammas.append((z[i][2]-z[j][2])**2)
37         if len(gammas)==0:
38             gamma = 0
39         else:
40             gamma = np.sum(gammas) / (len(gammas)*2.0)
41         semivariance.append(gamma)
42     semivariance = [ [lags[i], semivariance[i]]
43                      for i in range(len(lags))
44                      if semivariance[i]>0]
45     return np.array(semivariance).T
```

В следующем листинге приведен код, который прогоняет программу вычисления полудисперсии на наших тестовых данных. Длина выборочной области по каждой стороне равна 2970 м. Мы задаем диапазон лагов от 0 до 3000 с шагом 50. В строке 10 функция возвращает массив пар, в котором

каждому значению лага соответствует полудисперсия. На рис. 8.7 показан график зависимости полудисперсии от лага в виде кривой с нанесенными точками. Это эмпирическая полудисперсия, вычисленная по данным. Очевидно, что если изменить длину интервала, то получатся другие значения полудисперсии. Однако закономерность должна сохраниться. Отметим, что вместо термина «полудисперсия» часто употребляется термин «полувариограмма», относящийся к графику на рис. 8.7. Конечно, числа полезны, но полувариограмма часто гораздо более наглядно отражает связь между лагами и полудисперсией.

Листинг 8.7 ❖ Выполнение программы semivariance.py на тестовых данных

```
 1 import sys
 2 sys.path.append('../geom')
 3 from point import *
 4 import numpy as np
 5 from semivariance import *
 6
 7 Z = read_data('../data/necoldem.dat')
 8 hh = 50
 9 lags = np.arange(0, 3000, hh)
10 gamma = semivar(Z, lags, hh)
```

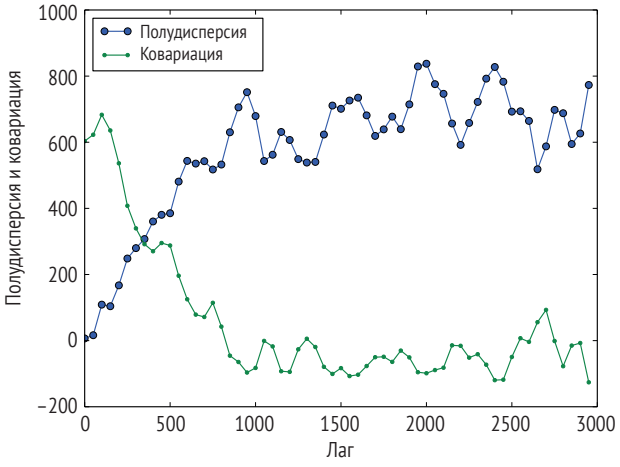

Рис. 8.7 ❖ Эмпирическая полудисперсия и ковариация для тестовых данных

8.2.2. Моделирование полудисперсии

Мы будем использовать полудисперсию для предсказания неизвестных значений в точках, для которых нет наблюдений. Полученная выше эмпирическая полудисперсия не идеальна для этой цели. Дело в том, что эмпирическая полудисперсия изменяется, когда мы изменяем участвующие в ее вычисле-

нии лаги, и, что еще более важно, эмпирическая кривая «нерегулярна», а это затрудняет получение значения $\gamma(h)$ для отсутствующих данных. Нам нужна теоретическая кривая, т. е. математическая функция, которая отражает наши представления о реальности и может дать значения полудисперсии при любом лаге.

Эмпирическая кривая все-таки позволяет составить некоторое представление о том, как должна выглядеть теоретическая. Прежде всего кривая должна начинаться в точке, соответствующей нулевому значению h. Это значение мы называем *наггетом* (самородок). В теоретической геостатистике, дисциплине, построенной на основе концепции кригинга, наггет – это дисперсия наблюдений, имеющая место, даже когда лаг равен нулю, – если два результата измерений, сделанных в одной точке на местности, различаются, то ошибка, скорее всего, связана с измерительным прибором. Далее видно, что полудисперсия достигает некоторого значения, после чего выходит на плато. Это значение называется *порогом* (sill), а расстояние, при котором достигается порог, называется *размахом* (range). Таким образом, в общем случае нужно определить три значения: наггет, порог и размах. Тот факт, что порог и размах присутствуют во многих эмпирических случаях, показывает, как расстояние влияет на дисперсию наблюдаемых значений: чем больше расстояние, тем выше дисперсия, а значит, тем сильнее несходство величин (в смысле измеренных значений).

Прежде чем переходить собственно к теоретической модели для аппроксимации данных, поговорим еще немного об идее сходства и несходства. Как уже отмечалось, полудисперсия дает способ измерить сходство наблюдений на основе их удаленности друг от друга, а сейчас мы введем еще одну количественную меру такого сходства. Это ковариация между парами:

$$C(h) = E[(z(x) - \mu)(z(x+h) - \mu)]$$
$$= \frac{1}{N}\sum_{i=1}^{N} z(x_i)z(x_i + h) - \mu^2,$$

где μ – среднее наблюдаемое значение при лаге h. По аналогии с программой `semivariance.py` мы можем написать на Python программу для вычисления ковариации, зная данные, интервалы лагов и длину интервала (листинг 8.8). Кривая эмпирической ковариации для наших тестовых данных с нанесенными на нее точками имеет тенденцию к убыванию с ростом лага (рис. 8.7).

Листинг 8.8 ❖ Вычисление ковариации (covariance.py)

```
import numpy as np
from semivariance import distance

def covar(z, lags, hh):
    """
    Вычисляет эмпирическую ковариацию по данным
    Вход:
      z - список,  каждый элемент которого является списком [x, y, data]
      lags - одномерный массив значений лагов
      hh - половина длины интервала расстояний
```

```
    Выход:
      Двумерный массив [ [h, C(h)], ...]
    """
    covariance = []
    N = len(z)
    D = [ [distance(z[i][0:2],z[j][0:2])
           for i in range(N)] for j in range(N)]
    for h in lags:
        C = []
        mu = 0
        for i in range(N):
            for j in range(N):
                if D[i][j] >= h-hh and D[i][j]<=h+hh:
                    C.append(z[i][2]*z[j][2])
                    mu += z[i][2] + z[j][ 2]
        if len(C)==0:
            Ch = 0
        else:
            mu = mu/(2*len(C))
            Ch = np.sum(C) / len(C) - mu*mu
        covariance.append(Ch)
    covariance = [ [lags[i], covariance[i]]
                   for i in range(len(lags))]
    return np.array(covariance).T
```

На рис. 8.7 хорошо видно, что тренды полудисперсии и ковариации противоположны. На самом деле если предположить, что среднее и дисперсия наблюдаемой величины постоянны в пространстве, то между этими величинами существуют такие связи:

$$C(0) = E[(z(x) - \mu)^2] = \text{var}[z(x)]$$

и

$$\gamma(h) = C(0) - C(h),$$

где var обозначает дисперсию (в данном случае – данных $z(x)$).

Как видно на рис. 8.7, при достаточно больших h (например, больших размаха) $\gamma(h)$ стремится к порогу, тогда как $C(h)$ равно нулю или очень близко к нему. Поэтому мы можем положить порог равным $C(0)$, т. е. дисперсии данных, как показывает первая формула. Теперь возникает вопрос: как найти значения наггета и размаха? С наггетом все просто – это значение полудисперсии при нулевом лаге. Итак, как оценить порог и наггет, мы знаем, единственный неизвестный параметр – размах. К этому вопросу мы вернемся позже. А пока предположим, что наггет, порог и размах заданы, и обозначим их соответственно c_0, c и a.

Существует много разных моделей, позволяющих аппроксимировать данные на основе этих значений. Простейшая среди них – линейная модель, которая выходит на пороговый уровень по прямой с постоянным угловым коэффициентом:

$$\gamma(h) = \begin{cases} c_0 + c\left(\frac{h}{a}\right) & 0 \le h \le a \\ c_0 + c & h > a \\ 0 & h = 0 \end{cases}.$$

Пожалуй, чаще других используется сферическая модель:

$$\gamma(h) = \begin{cases} c_0 + c\left[\frac{3h}{2a} - \frac{1}{2}\left(\frac{h}{a}\right)^3\right] & 0 \le h \le a \\ c_0 + c & h > a \\ 0 & h = 0 \end{cases}.$$

В сферической модели порог достигается на расстоянии размаха. Но, в отличие от линейной модели, при подходе к порогу угловой коэффициент уменьшается. Экспоненциальная модель описывается формулой

$$\gamma(h) = \begin{cases} c_0 + c\left[1 - e^{\left(\frac{h}{a}\right)}\right] & h > 0 \\ 0 & h = 0 \end{cases}.$$

Она похожа на сферическую модель, но на расстоянии размаха достигает только 95 % от порога. Кривая гауссовой модели имеет сигмоидную форму и не достигает порога на расстоянии размаха:

$$\gamma(h) = \begin{cases} c_0 + c\left[1 - e^{\left(\frac{h^2}{a^2}\right)}\right] & h > 0 \\ 0 & h = 0 \end{cases}.$$

Наконец, для моделирования полувариограммы можно использовать степенную функцию:

$$\gamma(h) = \begin{cases} c_0 + c|h|^{\lambda} & h > 0 \\ 0 & h = 0 \end{cases},$$

где λ – параметр от 0 до 2, который можно подобрать для конкретных данных.

Python-код в листинге 8.9 завершает аппроксимацию полувариограммы в функции `fitsemivariogram`, которая принимает три параметра: данные (`z`), эмпирическую полудисперсию (`s`) и одну из четырех определенных выше моделей. Формат определения этих моделей легко обобщается на любую другую модель, зависящую от лага (`h`), наггета (`c0`), порога (`c`) и размаха (`a`). Здесь мы используем имеющееся в NumPy средство – векторизацию функций, которое объявлено с помощью команды `@np.vectorize`. Векторизация в Python позволяет функциям принимать и возвращать массивы. Мы подробнее рассмотрим ее ниже.

В строке 56 вычисляется дисперсия данных, которая, как было сказано выше, равна значению порога. В строках 57–60 мы получаем значение наггета. Затем, в строке 61, программа использует минимальное и максимальное значения размаха в предположении, что наибольший размах не превос-

ходит верхней границы лага в эмпирической полудисперсии. На практике фактический размах может превышать наибольший лаг в данных, но тогда данные нельзя будет аппроксимировать моделью. Поэтому, когда мы пытаемся построить автоматический процесс вычисления параметров кригинга, нужно тщательно продумывать подготовку данных. В частности, как мы уже видели, необходимо, чтобы нулевой лаг присутствовал в данных, иначе мы не сможем вычислить значение наггета.

Для оценки значения размаха (A) для каждой модели мы используем полный перебор – проверяем много значений размаха и оставляем то, при котором ошибка минимальна. После вычисления эмпирической полудисперсии мы знаем предполагаемые значения полудисперсии при заданных значениях лага и можем использовать их для оценки качества модели. В строке 62 программа создает ряд размахов для проверки. По умолчанию создается 200 размахов. Для каждого размаха вычисляется среднеквадратическая ошибка между оценками при всех значениях лагов и известными наблюдениями (строка 64). Обратите внимание, как работает код в строке 64: все объявленные модели векторизованы, поэтому когда массив передается функции модели (здесь массив – это `s[0]`, и он содержит все лаги в наших данных), она обрабатывает каждый элемент массива и возвращает массив того же размера, который затем используется для вычисления квадрата разности при каждом значении лага. Лаги образуют массив, и в конце мы получаем среднеквадратическую ошибку. Функция NumPy `np.mean` вычисляет среднюю ошибку для каждого проверяемого размаха. Когда все потенциальные значения размаха будут проверены, то, при котором ошибка минимальна, выбирается в качестве размаха (a) модели (строка 65). Функция `fitsemivariogram` возвращает лямбда-функцию (см. приложение А.3), которая принимает всего один параметр (поскольку все остальные значения, c_0, c и a, откалиброваны).

Листинг 8.9 ❖ Аппроксимация полувариограммы (fitsemivariance.py)

```
 1 import numpy as np
 2
 3 @np.vectorize
 4 def spherical(h, c0, c, a):
 5     """
 6     Вход
 7       h: расстояние
 8       c0: порог
 9       c: наггет
10       a: размах
11     Выход
12       Теоретическая вариограмма с расстоянием h
13     """
14     if h<=a:
15         return c0 + c*(3.0*h/(2.0*a) - ((h/a)**3.0)/2.0)
16     else:
17         return c0 + c
18
19 @np.vectorize
```

```
20 def gaussian(h, c0, c, a):
21     """
22     Вход и выход, как для сферической
23     """
24     return c0 + c*(1-np.exp(-h*h/((a)**2)))
25
26 @np.vectorize
27 def exponential(h, c0, c, a):
28     """
29     Вход и выход, как для сферической
30     """
31     return c0 + c*(1-np.exp(-h/a))
32
33 @np.vectorize
34 def linear(h, c0, c, a):
35     """
36     Вход и выход, как для сферической
37     """
38     if h<=a:
39         return c0 + c*(h/a)
40     else:
41         return c0 + c
42
43 def fitsemivariogram(z, s, model, numranges=200):
44     """
45     Аппроксимирует теоретическую модель полудисперсии.
46     Вход
47       z: данные, 2-мерный массив NumPy, строки которого имеют вид X, Y, Значение
48       s: эмпирические полудисперсии
49       model: одна из моделей полудисперсии: сферическая,
50              гауссова, экспоненциальная или линейная
51     Выход
52       Лямбда-функция, играющая роль аппроксимированной модели
53       полувариограммы. Эта функция принимает один параметр
54       (расстояние).
55     """
56     c = np.var(z[:,2])                  # c, sill
57     if s[0][0] is not 0.0:              # c0, наггет
58         c0 = 0.0
59     else:
60         c0 = s[0][1]
61     minrange, maxrange = s[0][1], s[0][-1]
62     ranges = np.linspace(minrange, maxrange, numranges)
63     errs = [np.mean((s[1] - model(s[0], c0, c, r))**2)
64               for r in ranges]
65     a = ranges[errs.index(min(errs))]   # оптимальный размах
66     return lambda h: model(h, c0, c, a)
```

Листинг 8.10 является продолжением листинга 8.7, в нем продемонстрирована аппроксимация полувариограммы на примере наших тестовых данных. В строке 18 задается аппроксимация сферической моделью с данными `Z` и эмпирической полудисперсией `gamma`. Последний параметр функции

`fitsemivariogram` – имя функции, реализующей модель. В Python имя любой функции также является именем переменной, которая может быть передана другой функции. Мы также производим аппроксимацию другими теоретическими моделями: линейной (строка 19) и гауссовой (строка 20). Далее в программе используется еще один мощный Python-модуль pylab, который позволяет создавать эффектные визуализации. Сам код прост как для понимания, так и для обобщения. Отметим, что здесь снова используется векторизация функций. Например, в строке 24 на график наносится список данных – первый массив содержит значения лагов, откладываемых по оси *x*, а второй вычисляется с помощью аппроксимированной полувариограммы в этих точках.

Листинг 8.10 ❖ Аппроксимация полудисперсии с помощью тестовых данных (test_fitsemivariance.py)

```
 1 from pylab import *
 2 import numpy as np
 3 import sys
 4 sys.path.append('../geom')
 5 from point import *
 6 from semivariance import *
 7 from covariance import *
 8 from read_data import *
 9 from fitsemivariance import *
10
11 Z = read_data('../data/necoldem.dat')
12
13 hh = 50
14 lags = range(0, 3000, hh)
15 gamma = semivar(Z, lags, hh)
16 covariance = covar(Z, lags, hh)
17 Z1 = np.array(Z)
18 svs = fitsemivariogram(Z1, gamma, spherical)
19 svl = fitsemivariogram(Z1, gamma, linear)
20 svg = fitsemivariogram(Z1, gamma, gaussian)
21 sve = fitsemivariogram(Z1, gamma, exponential)
22
23 p1, = plot(gamma[0], gamma[1], 'o')
24 p2, = plot(gamma[0], svs(gamma[0]), color='grey', lw=2)
25 p3, = plot(gamma[0], svl(gamma[0]), color='grey', lw=2,
26            linestyle="--")
27 p4, = plot(gamma[0], svg(gamma[0]), color='grey', lw=2,
28            linestyle="-.")
29 p5, = plot(gamma[0], sve(gamma[0]), color='grey', lw=2,
30            linestyle=":")
31 models = ["Эмпирическая", "Сферическая", "Линейная",
32            "Гауссова", "Экспоненциальная"]
33 l1 = legend([p1,p2,p3,p4, p5], models, loc='lower right')
34 ylabel('Полудисперсия')
35 xlabel('Лаг (м)')
36 savefig('semivariogram_data_model.eps',fmt='eps')
37 show()
```

На рис. 8.8 показано различие между четырьмя аппроксимированными моделями (кривая с данными из EPS-файла на рисунке опущена). Видно, что сферическая модель дает вполне разумные оценки, близкие к эмпирическим данным.

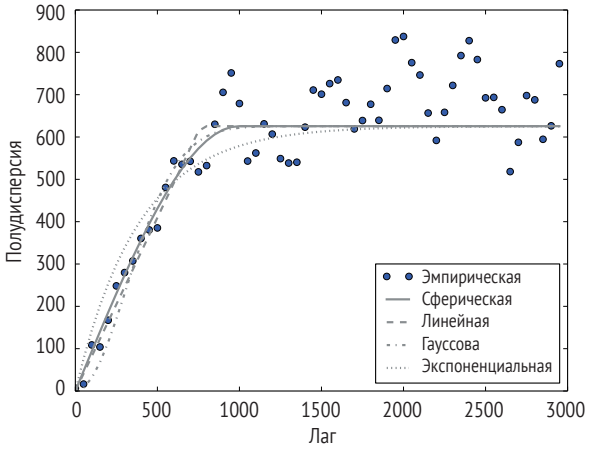

Рис. 8.8 ❖ Теоретические полувариограммы, построенные с помощью четырех разных моделей

8.2.3. Обыкновенный кригинг

Теперь обратимся к использованию полувариограммы в кригинге для интерполяции. Как и в случае метода обратных взвешенных расстояний, мы вычисляем неизвестное значение в точке, используя набор весов для некоторых близких точек:

$$\hat{z}(x_0) = \sum_i \lambda_i z(x_i),$$

где предполагается, что $\sum_i \lambda_i = 1$. Для кригинга очень важна гипотеза стационарности, предполагающая, что среднее и дисперсия не зависят от положения точек, т. е. формально $E[\hat{z}(x_0)] = E[z(x_0)] = E[z(x_i)]$ и $\mathrm{var}[\hat{z}(x_0)] = \mathrm{var}[z(x_0)] = \mathrm{var}[z(x_i)]$, где x_i – любая точка, в которой известно значение.

При обсуждении кригинга мы будем использовать матрицы. Когда не возникает неоднозначности, мы опускаем x_0 в z и $\hat{z}$. Оценка значения в точке x_0 записывается в виде

$$\hat{z} = \boldsymbol{\lambda}^{\mathrm{T}} \boldsymbol{z},$$

где $\boldsymbol{\lambda} = [\lambda_1, \lambda_2, \ldots, \lambda_n]^{\mathrm{T}}$ – матрица весов размера $n \times 1$, а $\boldsymbol{z} = [z(x_1), z(x_2), \ldots, z(x_n)]^{\mathrm{T}}$ – матрица n наблюдаемых значений, по которым производится интерполяция в точке x_0. Предполагается, что сумма весов равна 1:

$$\boldsymbol{\lambda}^{\mathrm{T}} \boldsymbol{J} = 1,$$

где $\boldsymbol{J}$ – матрица $n \times 1$, состоящая из единиц.

Начнем с ошибки интерполяции, которая равна разности между оценкой $\hat{z}$ и истинным, но неизвестным значением z в точке x_0:

$$e = \hat{z} - z.$$

Разумеется, истинного значения мы не знаем. Но хитрость в том, чтобы с помощью ряда преобразований представить ошибку в другой форме, которую уже можно будет вычислить. Рассмотрим дисперсию ошибки в точке x_0:

$$\text{var}(e) = E[(e - E[e])^2] = E[e^2] - (E[e])^2.$$

Для минимизации дисперсии ошибки нужно, чтобы ее математическое ожидание было равно нулю:

$$E[e] = 0,$$

откуда

$$\text{var}(e) = E[e^2] = E[(\hat{z} - z)^2].$$

Записывая взвешенную сумму в матричном виде, имеем

$$\text{var}(e) = E[\boldsymbol{\lambda}^{\mathrm{T}}\boldsymbol{z} - z)^2]$$

и после раскрытия скобок получаем

$$\text{var}(e) = E[z^2] + E[(\boldsymbol{\lambda}^{\mathrm{T}}\boldsymbol{z})^2] - 2E[z\boldsymbol{\lambda}^{\mathrm{T}}\boldsymbol{z}].$$

Можно показать, что каждый из трех членов в правой части можно раскрыть далее и записать в терминах ковариаций, пользуясь гипотезой стационарности. Обозначим $\boldsymbol{C}$ ковариационную матрицу размера $n \times n$ с элементами $c_{ij} = \text{cov}(x_i, x_j)$, равными ковариации между наблюдениями в точках x_i и x_j, а $\boldsymbol{C}_0$ – ковариационную матрицу размера $n \times 1$ с элементами $c_i = \text{cov}(x_0, x_i)$, равными ковариации между x_0 и каждым из наблюдений. Тогда дисперсию ошибки можно переписать в виде[1]

$$\text{var}(e) = \text{var}(z) + \boldsymbol{\lambda}^{\mathrm{T}}\boldsymbol{C}\lambda - 2\boldsymbol{\lambda}^{\mathrm{T}}\boldsymbol{C}_0,$$

где var(z) – дисперсия случайной величины в точке x_0. Думать о том, как вычислить var(z), не нужно, потому что в ходе последующих выкладок она пропадет.

В этом выводе нужно обратить внимание еще на одну вещь: требуется, чтобы сумма весов была равна 1, поскольку это предположение используется несколько раз. Сформулируем это условие как ограничение в задаче математической оптимизации:

---

[1] Для доказательства нужно воспользоваться свойством ковариации между двумя величинами X и Y: $\text{cov}(X, Y) = E[XY] - E[X]E[Y]$ и $\text{var}(X) = \text{cov}(X, X)$. Мы опускаем детали, которые можно найти в учебниках, упомянутых в конце главы.

минимизировать var(e)
при условии $2\mu(\lambda^T \boldsymbol{J} - 1) = 0$.

В этой формулировке единственное ограничение – сумма весов должна быть равна 1. Для включения сюда μ, среднего значения в точке x_0, есть веская причина: таким образом мы сумеем оценить среднее одновременно с оценкой весов. Эта задача называется *обыкновенным кригингом*.

Для решения данной задачи оптимизации мы рассматриваем ограничение как лагранжиан и переносим его в целевую функцию, так что целью становится минимизация следующей новой целевой функции:

$$q = \text{var}(z) + \boldsymbol{\lambda}^T \boldsymbol{C} \boldsymbol{\lambda} - 2\boldsymbol{\lambda}^T \boldsymbol{C}_0 - 2\mu(\boldsymbol{\lambda}^T \boldsymbol{J} - 1).$$

Для минимизации дисперсии необходимо, чтобы частные производные новой целевой функции обращались в нуль:

$$\frac{\partial q}{\partial \lambda} = 2\boldsymbol{C}\boldsymbol{\lambda} - 2\boldsymbol{C}_0 - 2\boldsymbol{\mu} = 0;$$
$$\frac{\partial q}{\partial \mu} = -2\boldsymbol{\lambda}^T \boldsymbol{J} + 2 = 0,$$

где $\boldsymbol{\mu}$ – матрица $n \times 1$, в которой все элементы равны μ. Вместе эти два условия дают следующую систему уравнений:

$$\begin{cases} \boldsymbol{C}\boldsymbol{\lambda} - \boldsymbol{\mu} = \boldsymbol{C}_0 \\ \boldsymbol{\lambda}^T \boldsymbol{J} = 1 \end{cases}.$$

Это система с $n + 1$ переменной, первый набор уравнений можно переписать в виде

$$\begin{pmatrix} c_{11} & c_{1,2} & \dots & c_{1n} & 1 \\ c_{21} & c_{2,2} & \dots & c_{2n} & 1 \\ \vdots & & & \vdots & \vdots \\ c_{n1} & c_{n2} & \dots & c_{nn} & 1 \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \\ -\mu \end{pmatrix} = \begin{pmatrix} c_{01} \\ c_{02} \\ \vdots \\ c_{0n} \end{pmatrix}.$$

Поскольку мы умеем моделировать ковариации, эту систему уравнений можно использовать для вычисления весов и оценки среднего. Но можно также преобразовать ее в более распространенную форму с помощью полувариограмм. Вспоминая связь между полудисперсией и ковариацией, $\gamma(h) = C(0) - C(h)$, получаем

$$c_{ij} = c(0) - \gamma_{ij}$$

и

$$c_{0i} = c(0) - \gamma_{0i}.$$

Таким образом, уравнения можно переписать в виде

$$\begin{pmatrix} \gamma_{11} & \gamma_{1,2} & \cdots & \gamma_{1n} & 1 \\ \gamma_{21} & \gamma_{2,2} & \cdots & \gamma_{2n} & 1 \\ \vdots & & & \vdots & \vdots \\ \gamma_{n1} & \gamma_{n2} & \cdots & \gamma_{nn} & 1 \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \\ \mu \end{pmatrix} = \begin{pmatrix} \gamma_{01} \\ \gamma_{02} \\ \vdots \\ \gamma_{0n} \end{pmatrix},$$

или, эквивалентно, в матричной форме

$$\boldsymbol{\gamma\lambda} + \boldsymbol{\mu} = \boldsymbol{\gamma}_0,$$

где $\boldsymbol{\gamma}$ – матрица $n \times n$, в которой элемент γ_{ij} равен полудисперсии между точками x_i и x_j, а $\boldsymbol{\gamma}_0$ – матрица $n \times 1$ полудисперсий между точкой x_0 и n точками, для которых имеются наблюдения.

Эту форму можно немного изменить, включив второе уравнение ($\boldsymbol{\lambda}^T\boldsymbol{J} = 1$):

$$\begin{pmatrix} \gamma_{11} & \gamma_{1,2} & \cdots & \gamma_{1n} & 1 \\ \gamma_{21} & \gamma_{2,2} & \cdots & \gamma_{2n} & 1 \\ \vdots & & & \vdots & \vdots \\ \gamma_{n1} & \gamma_{n2} & \cdots & \gamma_{nn} & 1 \\ 1 & 1 & \ldots & 1 & 0 \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \\ \mu \end{pmatrix} = \begin{pmatrix} \gamma_{01} \\ \gamma_{02} \\ \vdots \\ \gamma_{0n} \\ 1 \end{pmatrix}.$$

Обозначим первую матрицу размера $(n + 1) \times (n + 1)$ в левой части $\boldsymbol{K}$, вторую $[\lambda, \mu]$, а матрицу размера $(n + 1) \times 1$ в правой части $\boldsymbol{k}$. Тогда можно записать

$$\boldsymbol{K}[\lambda, \mu] = \boldsymbol{k}.$$

Для решения этой системы уравнений можно воспользоваться методом исключения Гаусса–Жордана. Но в компьютерных программах часто удобнее непосредственно умножить $\boldsymbol{k}$ на матрицу, обратную к $\boldsymbol{K}$, чтобы получить элементы $\boldsymbol{\lambda}$:

$$[\lambda, \mu] = \boldsymbol{K}^{-1}\boldsymbol{k}.$$

Вычислив веса, мы можем записать оценку дисперсии ошибки в виде

$$\hat{\sigma}^2 = [\lambda + \mu]^T\boldsymbol{k}.$$

Программа в листинге 8.11 реализует обыкновенный кригинг. Функция принимает два параметра: данные для близких точек и полувариограмму. Мы снова пользуемся полезными возможностями модуля NumPy. В строке 17 с помощью векторизации функции `semivariogram` (переданной в качестве параметра `model`) возвращаются полудисперсии для каждого расстояния от известной точки до интерполируемой. Затем мы присваиваем переменной `k` матрицу `1×N`, последний элемент которой равен 1 (строки 18 и 20). Для вычисления матрицы `K` мы сначала вычисляем матрицу расстояний `N×N` (строка 22). Эта матрица сериализуется в массив, содержащий `N×N` элементов, который используется для вычисления полудисперсии для каждого расстояния (строка 23). Результат преобразуется обратно в матрицу `N×N` (строки 24 и 25).

Затем мы дополняем ее до матрицы размера (N+1)×(N+1), добавив строку и столбец, содержащие единицы, за исключением правого нижнего угла, в котором находится 0 (строки 26–30). В строке 31 решается система уравнений, из которой находятся веса, используемые в строке 32 для нахождения оценки. В строках 33–37 вычисляется стандартное отклонение ошибки и гарантируется, что оно положительно. Наконец, программа возвращает оценку значения, оценку стандартного отклонения ошибки, оценку среднего и веса.

Листинг 8.11 ❖ Обыкновенный кригинг (okriging.py)

```
 1 import numpy as np
 2 from semivariance import distance
 3
 4 def okriging(Z, model):
 5     """
 6     Обыкновенный кригинг.
 7     Вход
 8       Z: массив элементов вида [X, Y, Val, расстояние до x0]
 9       model: имя модели аппроксимации полудисперсии
10     Выход
11       zhat: оценка значения в точке x0
12       sigma: стандартная ошибка
13       mu: оценка среднего
14       w: веса
15     """
16     N = len(Z)                              # число точек
17     k = model(Z[:,3])                       # получить [gamma(xi, x0)]
18     k = np.matrix(k).T                      # k - матрица 1×N
19     k1 = np.matrix(1)
20     k = np.concatenate((k, k1), axis=0)     # добавить строку единиц
21     K = [ [distance(Z[i][0:2],Z[j][0:2])
22           for i in range(N)] for j in range(N)]
23     K = np.array(K)                         # список -> массив NumPy
24     K = model(K.ravel())                    # [gamma(xi, xj)]
25     K = np.matrix(K.reshape(N, N))          # массив -> матрица N×N
26     ones = np.matrix(np.ones(N))            # матрица N×1, содержащая единицы
27     K = np.concatenate((K, ones.T), axis=1) # добавить столбец единиц
28     ones = np.matrix(np.ones(N+1))          # матрица (N+1)×1, содержащая единицы
29     ones[0, N] = 0.0                        # последний элемент равен 0
30     K = np.concatenate((K, ones), axis=0)   # добавить строку
31     w = np.linalg.solve(K, k)               # решить систему: K w = k
32     zhat = (np.matrix(Z[:,2])*w[:-1])[0, 0] # оценка значения
33     sigmasq = (w.T * k)[0, 0]               # оценка дисперсии ошибки
34     if sigmasq < 0:
35         sigma = 0
36     else:
37         sigma = np.sqrt(sigmasq)            # ошибка
38     return zhat, sigma, w[-1][0], w         # оценка, ошибка, mu, w
```

Затем мы протестировали интерполяцию для двух точек (листинг 8.12). Сначала мы взяли произвольную точку (337000, 4440911); получилась оценка 860.314 со стандартной ошибкой 6.757. Потом мы взяли первую точку дан-

ных (отметим, что эта точка не исключалась из данных на этапе вычисления весов) и получили оценку 869.0 со стандартной ошибкой 0. Как видим, метод обыкновенного кригинга дает точное значение для точек с известными результатами измерения. Это свойство, называемое точной интерполяцией, можно доказать формально, опираясь на приведенные выше формулы.

Листинг 8.12 ❖ Тестирование обыкновенного кригинга (ordinary_kriging_test.py)

```
import numpy as np
import sys
sys.path.append('../geom')
from point import *
from fitsemivariance import *
from semivariance import *
from covariance import *
from read_data import *
from prepare_interpolation_data import *
from okriging import *

Z = read_data('../data/necoldem.dat')

hh = 50
lags = range(0, 3000, hh)
gamma = semivar(Z, lags, hh)
covariance = covar(Z, lags, hh)

Z1 = np.array(Z)
semivariogram = fitsemivariogram(Z1, gamma, spherical)
#semivariograml = fitsemivariogram(Z1, gamma, linear)
#semivariogramg = fitsemivariogram(Z1, gamma, gaussian)

if __name__ == "__main__":
    x = Point(337000, 4440911)
    P1 = prepare_interpolation_data(x, Z1)[0]
    print okriging(np.array(P1), semivariogram)[0:2]

    x = Point(Z1[0,0], Z1[0,1])
    P1 = prepare_interpolation_data(x, Z1)[0]
    print okriging(np.array(P1), semivariogram)[0:2]
```

8.2.4. Простой кригинг

В случае простого кригинга мы считаем, что среднее постоянно во всей области. Этим он отличается от обыкновенного кригинга, когда среднее изменяется от точки к точке. В таком случае нас больше интересует то, что останется после вычитания среднего из значения: поскольку среднее всегда одинаково, включать его в данные не имеет смысла. Поэтому определим функцию невязки в каждой точке x:

$$R(x) = z(x) - \mu,$$

где μ – постоянное среднее. Теперь можно определить матрицу $\boldsymbol{R}$ размера $n \times 1$, где $R_i = R(x_i)$ – невязка в точке x_i. Положим также $R = R(x_0)$ и $\hat{R} = \hat{R}(x_0)$, чтобы упростить обозначения.

Как и в случае обыкновенного кригинга, рассмотрим ошибку оценивания:

$$\begin{aligned} e &= \hat{z} - z \\ &= (\hat{z} - \mu) - (z - \mu) \\ &= \hat{R} - R \\ &= \boldsymbol{\lambda}^{\mathrm{T}}\boldsymbol{R} - R. \end{aligned}$$

Дисперсию ошибки можно записать в виде

$$\begin{aligned} \operatorname{var}(e) &= E[(\boldsymbol{\lambda}^{\mathrm{T}}\boldsymbol{R} - R)^2] \\ &= E[R^2] + E[(\boldsymbol{\lambda}^{\mathrm{T}}\boldsymbol{R})^2] - 2E[R\boldsymbol{\lambda}^{\mathrm{T}}\boldsymbol{R}] \\ &= \operatorname{var}(R) + \boldsymbol{\lambda}^{\mathrm{T}}\boldsymbol{C}\boldsymbol{\lambda} - 2\boldsymbol{\lambda}^{\mathrm{T}}\boldsymbol{C}_0. \end{aligned}$$

Поскольку при выводе этой формулы мы уже воспользовались тем, что среднее постоянно, сразу возьмем частные производные дисперсии ошибки по каждому весу; дисперсия достигает минимума, когда все частные производные равны нулю. Это называется *простым кригингом*. Имеем

$$\frac{\partial \operatorname{var}(e)}{\partial \boldsymbol{\lambda}} = 2\boldsymbol{C}\boldsymbol{\lambda} - 2\boldsymbol{C}_0 = 0.$$

Воспользовавшись связью между полудисперсией и ковариацией, мы можем переписать эту систему уравнений в виде

$$\boldsymbol{\gamma}\boldsymbol{\lambda} = \boldsymbol{\gamma}_0,$$

или в матричной форме

$$\begin{pmatrix} \gamma_{11} & \gamma_{1,2} & \cdots & \gamma_{1n} \\ \gamma_{21} & \gamma_{2,2} & \cdots & \gamma_{2n} \\ \vdots & & & \vdots \\ \gamma_{n1} & \gamma_{n2} & \cdots & \gamma_{nn} \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \end{pmatrix} = \begin{pmatrix} \gamma_{01} \\ \gamma_{02} \\ \vdots \\ \gamma_{0n} \end{pmatrix}.$$

Для единообразия с формулой обыкновенного кригинга запишем это в виде

$$\boldsymbol{K}\boldsymbol{\lambda} = \boldsymbol{k},$$

где $\boldsymbol{k}$ и $\boldsymbol{K}$ – матрицы размера $n \times 1$ и $n \times n$ соответственно. Для решения этого уравнения умножим обе части на матрицу, обратную $\boldsymbol{K}$:

$$\boldsymbol{\lambda} = \boldsymbol{K}^{-1}\boldsymbol{k}.$$

После вычисления весов получаем оценку дисперсии ошибки

$$\hat{\sigma}^2 = \boldsymbol{\lambda}^{\mathrm{T}}\boldsymbol{k}.$$

Программа в листинге 8.13 реализует метод простого кригинга. Важно отметить, что для интерполяции неизвестного значения мы должны задать

глобальное среднее. Кроме того, для вычисления оценки мы используем не сами исходные данные, а невязки (строка 25), и получаем оценку невязки, а не искомого значения. Чтобы получить оценку значения, нужно сложить оценку невязки со средним (строка 27).

Листинг 8.13 ❖ Простой кригинг (skriging.py)

```
 1 import numpy as np
 2 from semivariance import distance
 3
 4 def skriging(Z, mu, model):
 5     """
 6     Простой кригинг.
 7     Вход
 8       Z: массив элементов вида [X, Y, Val, расстояние до x0]
 9       mu: среднее
10       model: имя модели аппроксимации полудисперсии
11     Выход
12       zhat: оценка значения в интерполируемой точке
13       sigma: стандартная ошибка
14       w: веса
15     """
16     N = len(Z)                      # число точек
17     k = model(Z[:,3])               # получить [gamma(xi, x0)]
18     k = np.matrix(k).T              # матрица 1×N
19     K = [ [distance(Z[i][0:2],Z[j][0:2])
20             for i in range(N)] for j in range(N)]
21     K = np.array(K)                 # список -> массив NumPy
22     K = model(K.ravel())            # [gamma(xi, xj)]
23     K = np.matrix(K.reshape(N, N))  # массив -> матрица N×N
24     w = np.linalg.solve(K, k)       # решить систему K w = k
25     R = Z[:,2] - mu                 # получить невязки
26     zhat = (np.matrix(R)*w)[0, 0]   # оценка невязки
27     zhat = zhat + mu                # оценка значения
28     sigmasq = (w.T*k)[0, 0]         # оценка дисперсии ошибки
29     if sigmasq<0:
30         sigma = 0
31     else:
32         sigma = np.sqrt(sigmasq)    # ошибка
33     return zhat, sigma, w           # оценка, ошибка, веса
```

Мы можем написать еще одну Python-программу для тестирования функции `skriging` на данных (листинг 8.14), при этом мы повторно использовали код подготовки данных из программы обыкновенного кригинга.

Листинг 8.14 ❖ Тестирование простого кригинга (simple_kriging_test.py)

```
1 from ordinary_kriging_test import *
2 from skriging import *
3
4 if __name__ == "__main__":
5     x = Point(337000, 4440911)
```

```
 6     P1, mu = prepare_interpolation_data(x, Z1)
 7     print skriging(np.array(P1), mu, semivariogram)[0:2]
 8
 9     x = Point(Z1[0,0], Z1[0,1])
10     P1, mu = prepare_interpolation_data(x, Z1)
11     print skriging(np.array(P1), mu, semivariogram)[0:2]
```

Результат тестирования простого кригинга показан ниже. Оценка в первой точке равна 860.344 с ошибкой 6.758, что немного больше предыдущей оценки. В случае оценки для первой точки данных код дает тот же результат, что и раньше, – точное значение.

```
(860.34415501300725, 6.7576953251206504)
(869.0, 0)
```

8.3. Применение методов интерполяции

Теперь мы воспользуемся методами обратных взвешенных расстояний и кригинга для интерполяции поверхности по выборочным данным. В листинге 8.15 использован написанный выше код (строка 1–19) и показано, как интерполировать поверхность с помощью изученных нами методов. Мы хотим создать поверхности с размером ячейки, равным 1/100 от длины наименьшей стороны области (строка 24). В нашем случае разрешение будет равно 30 м, а поверхность представлена сеткой 100×100. Результаты интерполяции будем хранить в двумерных массивах (строки 27–31), где для обоих методов кригинга имеется поверхность значений и поверхность ошибок. В процессе генерации поверхностей мы используем координаты каждой точки (строка 34) для подготовки данных (строка 35), необходимых методам интерполяции (строки 37–43). Мы сохраняем данные в файлах, а для рисования поверхностей применяем программу gnuplot с открытым исходным кодом. Мы создали также дополнительный файл для хранения поверхности, представляющей разность двух методов кригинга.

Листинг 8.15 ❖ Интерполяция поверхности (interpolate_surface.py)

```
 1 import sys
 2 sys.path.append('../geom')
 3 from point import *
 4 import numpy as np
 5 from semivariance import *
 6 from fitsemivariance import *
 7 from prepare_interpolation_data import *
 8 from read_data import *
 9 from skriging import *
10 from okriging import *
11 from idw import *
12
13 Z = read_data('../data/necoldem.dat')
14 Z = np.array(Z)
```

```
15
16 hh = 50
17 lags = range(0, 3000, hh)
18 gamma = semivar(Z, lags, hh)
19 semivariogram = fitsemivariogram(Z, gamma, spherical)
20
21 x0, x1 = Z[:,0].min(), Z[:,0].max()
22 y0, y1 = Z[:,1].min(), Z[:,1].max()
23 dx, dy = x1-x0, y1-y0
24 dsize = min(dx/100.0, dy/100.0)
25 nx = int(np.ceil(dx/dsize))
26 ny = int(np.ceil(dy/dsize))
27 surfaceOK = np.zeros((nx, ny))
28 errorOK = np.zeros((nx, ny))
29 surfaceSK = np.zeros((nx, ny))
30 errorSK = np.zeros((nx, ny))
31 surfaceIDW = np.zeros((nx, ny))
32 for i in range(nx):
33     for j in range(ny):
34         x = Point(x0+i*dsize, y0+j*dsize)
35         Z1, mu = prepare_interpolation_data(x, Z)
36         Z1 = np.array(Z1)
37         kresult = okriging(Z1, semivariogram)
38         surfaceOK[i,j] = kresult[0]
39         errorOK[i,j] = kresult[1]
40         kresult = skriging(Z1, mu, semivariogram)
41         surfaceSK[i,j] = kresult[0]
42         errorSK[i,j] = kresult[1]
43         surfaceIDW[i,j] = IDW(Z1, 1.5)
44
45 f1 = open('results/surfaceOK', 'w')
46 f2 = open('results/errorOK', 'w')
47 f3 = open('results/surfaceSK', 'w')
48 f4 = open('results/errorSK', 'w')
49 f5 = open('results/surfaceIDW', 'w')
50 f6 = open('results/surfaceKDiff', 'w')
51 for j in range(ny):
52     for i in range(nx):
53         f1.write(str(surfaceOK[i, j])+' ')
54         f2.write(str(errorOK[i, j])+' ')
55         f3.write(str(surfaceSK[i, j])+' ')
56         f4.write(str(errorSK[i, j])+' ')
57         f5.write(str(surfaceIDW[i, j])+' ')
58         f6.write(str(surfaceOK[i, j]-surfaceSK[i, j])+' ')
59     f1.write('\n')
60     f2.write('\n')
61     f3.write('\n')
62     f4.write('\n')
63     f5.write('\n')
64     f6.write('\n')
65
66 f1.close()
67 f2.close()
```

```
68 f3.close()
69 f4.close()
70 f5.close()
71 f6.close()
```

Поверхности значений, сгенерированные методами обыкновенного и простого кригинга, показаны на рис. 8.9. Первое, что бросается в глаза, – близкое сходство между двумя результатами. Различие показано на рис. 8.10. В наших данных значения изменяются в диапазоне от 12.1491 до 16.9583, а по рисунку видно, что разность между двумя методами кригинга составляет от –0.1 до 0.25, т. е. примерно 2 %. На рис. 8.10 заметны также кластеры на карте разностей.

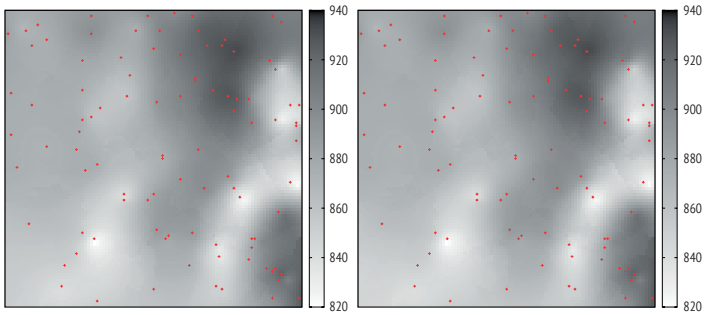

Рис. 8.9 ❖ Поверхности, сгенерированные методом обыкновенного (слева) и простого (справа) кригинга

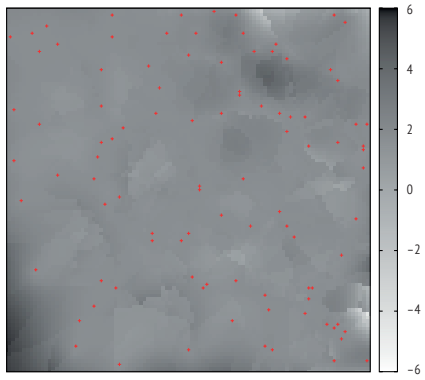

Рис. 8.10 ❖ Разность между результатами обыкновенного и простого кригинга

На рис. 8.11 показаны поверхности ошибок для обоих методов кригинга. И снова карты очень похожи. Очевидно совпадение областей низких ошибок и областей наблюдаемых данных. Этот рисунок наглядно показывает, почему кригинг называют методом точной интерполяции: он возвращает наблюдаемое значение в точках, где это значение известно.

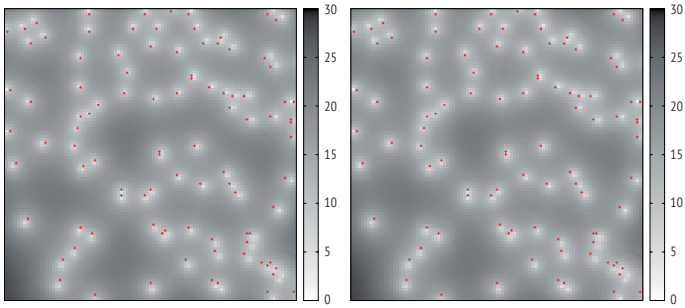

Рис. 8.11 ❖ Ошибка оценки

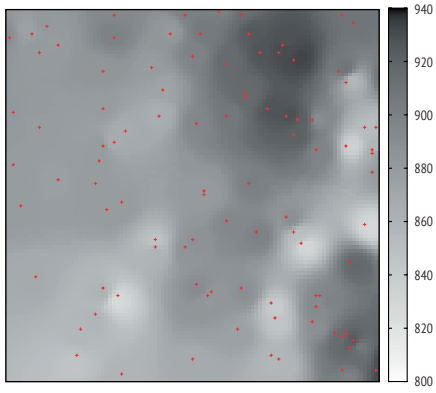

Рис. 8.12 ❖ Поверхность,
сгенерированная методом обратных взвешенных расстояний

Наконец, на рис. 8.12 показана интерполированная поверхность, сгенерированная методом ОВР. Она заметно отличается от тех, что сгенерированы методами кригинга. Внимательное изучение этой карты также показывает влияние данных наблюдения на результаты: имеется сильная корреляция между каждым наблюдением и результатом интерполяции рядом с ним. Выглядит это как «бычьи глаза» (светлые или темные круговые области) вокруг таких точек. Хотя тот же эффект можно обнаружить на поверхностях, созданных с помощью кригинга, в случае ОВР он выражен ярче. Иными словами, поверхности, полученные кригингом, оказываются более гладкими, чем полученные с помощью ОВР.

Так какой же метод, кригинг или ОВР, дает лучший результат? Поверхность теоретической ошибки, возвращенная кригингом, на самом деле ничего не говорит о качестве этого метода, особенно в сравнении с другими методами, в частности ОВР. Поэтому воспользуемся методом перекрестной проверки, как делали для ОВР, чтобы понять, какова ошибка кригинга в точках, где нам известен результат (но при этом не будем использовать точку, в которой проверяем качество, при вычислении весов). Эта идея реализована в листинге 8.16, вычисленная СКО равна 13.2574012443. Что, если бы мы использовали полувариограмму, вычисленную по всему набору данных, но

исключали по одной точке для проверки качества интерполяции? Мы можем внести небольшие изменения в код, чтобы не вычислять гамму и полувариограмму каждый раз. В таком случае получается СКО 13.4853542512. Следует также отметить, что для вычисления полудисперсии мы используем только ближайших соседей (т. е. передаем `P1`, а не `P` в строке 27). Конечно, это повлияет (в сторону увеличения) на величину СКО, зато мы выигрываем за счет уменьшения времени работы. В любом случае полученная СКО очень близка к наилучшей для метода OBP (13.2774718585).

Листинг 8.16 ❖ Перекрестная проверка для обыкновенного кригинга (kriging_cross_validation.py)

```
 1 import numpy as np
 2 import sys
 3 sys.path.append('../geom')
 4 from point import *
 5 from fitsemivariance import *
 6 from semivariance import *
 7 from covariance import *
 8 from read_data import *
 9 from prepare_interpolation_data import *
10 from okriging import *
11
12 Z = read_data('../data/necoldem250.dat')
13
14 hh = 50
15 lags = np.arange(0, 3000, hh)
16 test_results = []
17 N = len(Z)
18 mask = [True for i in range(N)]
19 numNeighbors = 10
20
21 for i in range(N):
22     mask[i] = False
23     x = Point(Z[i][0], Z[i][1])
24     P = [ Z[j] for j in range(N) if mask[j] == True]
25     P1 = prepare_interpolation_data(x, P, numNeighbors)[0]
26     P1 = np.array(P1)
27     gamma = semivar(P1, lags, hh)
28     if len(gamma) == 0:
29         continue
30     semivariogram = fitsemivariogram(P1, gamma, spherical)
31     kresult = okriging(P1, semivariogram)
32     test_results.append(kresult[0]-Z[i][2])
33     mask[i] = True
34
35 print np.sqrt(sum([r**2 for r in test_results])/
36               len(test_results))
```

Приведенные выше результаты оставляют нас в недоумении: действительно ли кригинг дает более точные оценки? Конечно, невозможно прийти

к определенному выводу на основании лишь одного простого примера. Но что этот простой пример все-таки может нам сказать? Посмотрим, откуда взялись данные, показанные на рис. 8.13. Наши 100 точек были случайно выбраны из данных цифровой модели рельефа (ЦМР) Геологической службы США в квадрате со стороной 7,5 угловой минуты с разрешением 30 м. Сравнение оригинальных данных ЦМР с интерполированными с очевидностью показывает, что они различны, хотя интерполированные поверхности похожи на оригинальные. А что, если увеличить размер выборки?

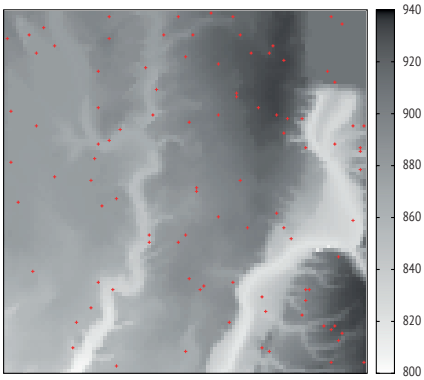

Рис. 8.13 ❖ Оригинальная ЦМР, из которой выбрано 100 точек

Увеличив число точек в выборке до 250 и выполнив ОВР с перекрестной проверкой, мы получим следующий результат:

```
12.2694123739 [ 0 ]
11.2761773623 [ 0.5 ]
10.4449626353 [ 1 ]
9.88226564121 [ 1.5 ]
9.55568320108 [ 2 ]
9.3882134655 [ 2.5 ]
9.32030718212 [ 3 ]
9.31308533549 [ 3.5 ]
9.34163738372 [ 4 ]
9.38982234683 [ 4.5 ]
9.44722571849 [ 5 ]
```

Увеличение выборки определенно повышает точность оценки. Метод обыкновенного кригинга с перекрестной проверкой дает СКО 8.66773948442. На этот раз кригинг, очевидно, лучше. Но нужно поставить больше экспериментов, чтобы увереннее судить о плюсах и минусах обоих методов. Еще одна проблема состоит в том, что оптимальное значение степени в ОВР изменилось и стало равно 3.5, а не 1.5, как при выборке 100 точек; кригинг лишен такой субъективности, потому что все параметры оцениваются на основе данных. На рис. 8.14 показан новый результат, улавливающий детали, которые ускользнули, когда точек было всего 100.

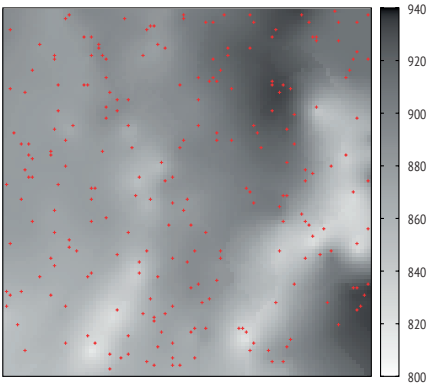

Рис. 8.14 ❖ Модель ЦМР, созданная с помощью обыкновенного кригинга по выборке из 250 точек

8.4. Смещение средней точки

В этом разделе мы распространим обсуждение интерполяции на один интересный вопрос: как сгенерировать случайную искусственную поверхность, повторяющую реальный земной ландшафт? Здесь мы рассматриваем разность высот в двух точках как функцию от расстояния между точками. Обозначим d расстояние между точками x и $x + d$ и предположим, что дисперсия значений в этих двух точках описывается соотношением

$$E[z(x) - z(x + d)]^2 \mu |d|^{2H},$$

где $0 \leqslant H \leqslant 1$ – постоянная, определяющая изменение значений в целом.

В статье Fournier et al. (1982) разработан алгоритм смещения средней точки, который рекурсивно добавляет среднюю точку в заданный набор четырех точек; значения в этих четырех точках выбраны из нормального распределения с дисперсией σ^2, и предполагается, что четыре точки расположены в вершинах квадрата со стороной s. Средняя точка находится на расстоянии $s/\sqrt{2}$ от каждой вершины. Значение в средней точке вычисляется путем выборки из нормального распределения с дисперсией $(1/2)^H\sigma^2$ и средним, равным среднему арифметическому значений в четырех точках. Иными словами, мы прибавляем небольшое смещение, выбранное из нормального распределения $N(0, (1/2)^H\sigma^2)$, к среднему значению в четырех точках, отсюда и название алгоритма. При увеличении H от 0 до 1 дисперсия смещения уменьшается от σ^2 до $1/2\sigma^2$.

Существуют две пространственные конфигурации четырех точек: квадратная и ромбовидная (рис. 8.15). На первой итерации алгоритм рассматривает квадратную конфигурацию и чередует одну с другой для интерполяции дополнительных точек, постепенно заполняющих пространство. На каждой итерации n сначала выполняется интерполяция с помощью квадратной

конфигурации, а затем с помощью ромбовидной. Количество точек на каждой стороне на n-й итерации равно $2^n + 1$, а общее количество точек равно $(2^n + 1)^2$. Дисперсия на n-й итерации равна $(1/2)^{nH}\sigma^2$. На рис. 8.16 показаны три итерации, в результате которых число точек увеличилось до 81. Алгоритм также прибавляет к сгенерированным значениям небольшую величину, выбираемую из того же нормального распределения, чтобы повысить степень случайности результата

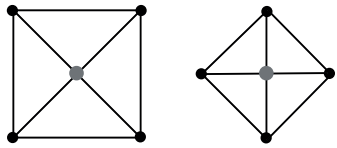

Рис. 8.15 ❖ Смещение средней точки в квадрате (слева) и ромбе (справа). Черными кружками обозначены текущие точки, а серым – сгенерированная точка

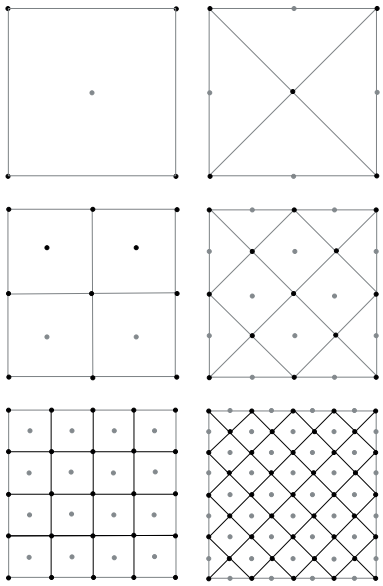

Рис. 8.16 ❖ После трех итераций смещения средней точки создается сетка, содержащая 81 точку

Python-код в листинге 8.17 по существу является адаптированным псевдокодом, опубликованным в книге Peitgen and Saupe (1988). Функция `f` (строка 7) используется для вычисления смещения среднего значения списка точек при заданном стандартном отклонении `delta`. Технически количество известных точек может быть любым, но мы будем использовать либо 4, как в типичном случае, когда имеется четыре вершины квадрата, либо 3 для интерполяции на краю сетки (как, например, для левой средней точки на правом верхнем рис. 8.16).

Функция `midpoint2d` реализует основную часть алгоритма. Входной параметр `maxlevel` задает число итераций алгоритма, а значит, и размер финальной сетки (строка 11). Параметр `sigma` – это начальное стандартное отклонение нормального распределения значений. В строках 14–17 инициализируются четыре угла сетки, значения в каждом выбираются из нормального распределения со средним 0 и стандартным отклонением, заданным параметром `midpoint2d`. На каждой итерации переменные `D` и `d` используются для решения о том, какие точки интерполировать, а начальные значения этих переменных устанавливаются в строках 18 и 19. На каждой итерации стандартное отклонение пересчитывается для квадратной (строка 21) и ромбовидной (строка 30) конфигураций. В случае квадратной конфигурации мы всегда имеем четыре точки для интерполяции значения во внутренней точке, что делается в цикле `for`, начинающемся в строке 23. Дополнительная случайная поправка прибавляется (если необходимо) во фрагменте, начинающемся в строке 26. Но для ромбовидной конфигурации нужно вычислять точки на границе, когда имеется всего три входные точки. Это делается в цикле `for`, начинающемся в строке 31. Внутренние точки в ромбовидной конфигурации для случаев, когда доступно четыре точки, обсчитываются в блоках, начинающихся в строках 36 и 40. При необходимости прибавляется случайная поправка (строка 44).

Листинг 8.17 ❖ Фрактальная интерполяция (midpoint2d.py)

```
 1 import random
 2 import math
 3 import numpy as np
 4 import sys
 5 import string
 6
 7 def f(delta, x):
 8     return sum(x)/len(x)+delta*random.gauss(0,1)
 9
10 def midpoint2d(maxlevel, sigma, H, addition=False):
11     N = int(math.pow(2, maxlevel))
12     X = [ [0]*(N+1) for x in range(N+1)]
13     delta = sigma
14     X[0][0] = delta*random.gauss(0,1)
15     X[0][N] = delta*random.gauss(0,1)
16     X[N][0] = delta*random.gauss(0,1)
17     X[N][N] = delta*random.gauss(0,1)
18     D = N
19     d = N/2
20     for stage in range(1, maxlevel+1):
21         delta = delta*math.pow(0.5, 0.5*H)
22         for x in range(d, N-d+1, D):
23             for y in range(d, N-d+1, D):
24                 X[x][y] = f(delta, [X[x+d][y+d], X[x+d][y-d],
25                                     X[x-d][y+d], X[x-d][y-d]])
```

```
26          if addition is True:
27              for x in range(0, N+1, D):
28                  for y in range(0, N+1, D):
29                      X[x][y] += delta*random.gauss(0, 1)
30          delta = delta*math.pow(0.5, 0.5*H)
31          for x in range(d, N-d+1, D):
32              X[x][0] = f(delta,[X[x+d][0],X[x-d][0],X[x][d]])
33              X[x][N] = f(delta,[X[x+d][N],X[x-d][N],X[x][N-d]])
34              X[0][x] = f(delta,[X[0][x+d],X[0][x-d],X[d][x]])
35              X[N][x] = f(delta,[X[N][x+d],X[N][x-d],X[N-d][x]])
36          for x in range(d, N-d+1, D):
37              for y in range(D, N-d+1, D):
38                  X[x][y] = f(delta, [X[x][y+d], X[x][y-d],
39                                      X[x+d][y], X[x+d][y]])
40          for x in range(D, N-d+1, D):
41              for y in range(d, N-d+1, D):
42                  X[x][y] = f(delta, [X[x][y+d], X[x+d][y-d],
43                                      X[x+d][y], X[x-d][y]])
44          if addition is True:
45              for x in range(0, N+1, D):
46                  for y in range(0, N+1, D):
47                      X[x][y] += delta*randomgauss(0, 1)
48              for x in range(d, N-d+1, D):
49                  for y in range(d, N-d+1, D):
50                      X[x][y] += delta*random.gauss(0, 1)
51          D=D/2
52          d=d/2
53      return X
54
55  if __name__ == '__main__':
56      """
57      Python midpoint2d.py
58      Python midpoint2d.py maxlevel sigma H
59      """
60      maxlevel = 6
61      sigma = 0.5
62      H = 0.1
63      if len(sys.argv) == 4:
64          maxlevel = string.atoi(sys.argv[1])
65          sigma = string.atof(sys.argv[2])
66          H = string.atof(sys.argv[3])
67      X = midpoint2d(maxlevel, sigma, H)
68      for i in X:
69          for j in i:
70              print j,
71          print
```

На рис. 8.17 показаны три поверхности, сгенерированные этим алгоритмом. Во всех трех случаях начальное стандартное отклонение равно 0.5. Ясно, что при увеличении *H* поверхность получается более гладкой (слева), а при уменьшении – более неровной.

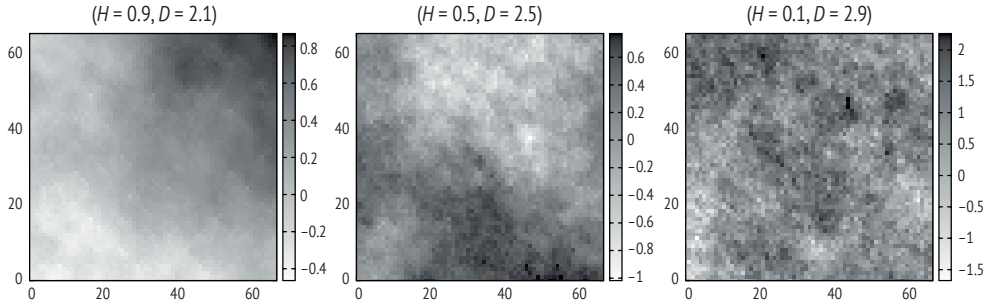

Рис. 8.17 ❖ Поверхности, созданные алгоритмом смещения средней точки при различных значениях *H*

8.5. Примечания

Что такое кригинг на самом деле? Метод кригинга называют также оптимальным предсказанием или оптимальной интерполяцией, поскольку он минимизирует дисперсию ошибки. Мы показали, как использовать кригинг для оценивания неизвестных значений на основе известных наблюдений. Дисциплина, которая теперь называется геостатистикой, зародилась в 1950-х годах из эмпирической работы Дани Г. Криге, южноафриканского горного инженера, который разработал вероятностный подход к оцениванию плотности минерально-сырьевых ресурсов (Krige, 1951). Однако работа Криге оставалась малоизвестной, до тех пор пока французский инженер Матерон (1963) не назвал «кригингом» этот метод пространственной интерполяции в честь его первооткрывателя. В обзоре Cressie (1990) показано, что различные ученые по-разному определяли кригинг, но в большинстве определений признается весовая природа метода. Начиная с 1980-х годов геостатистика стала признанной частью наук о Земле. Этой теме посвящено много книг по различным предметным областям, в т. ч. по геологии и природным ресурсам (Webster and Oliver, 1990; Cressie, 1991; Carr, 1995). Читателям, интересующимся деталями, рекомендуем обратиться к этим книгам.

Помимо простого и обыкновенного кригинга, на которых мы сосредоточились в этой книге, существуют и другие методы кригинга, предназначенные для применения в различных ситуациях. Например, универсальный кригинг способен улавливать общий тренд в данных. В приведенном выше описании полувариограмма не изменяется при смене направления на известную точку. Конечно, можно учесть пространственную анизотропию (т. е. ситуацию, когда свойство зависит от направления), допустив такие изменения. Можно также включить в рассмотрение пространственную связь между величинами, представляющими интерес, и другими пространственными переменными, что может дать больше информации для составления лучших прогнозов – это епархия кокригинга.

Кроме кригинга и метода обратных взвешенных расстояний, есть и другие методы интерполяции. Теоретической подоплекой всех этих методов

всегда был первый закон географии (Tobler, 1970). В работе Burrough and McDonnell (1998) приведен список методов интерполяции и сравнение их сильных и слабых сторон. По мнению авторов, метод OBP хорош для «быстрой интерполяции разреженных данных на регулярной сетке или выборки с нерегулярными расстояниями» и характеризуется «небольшой» вычислительной нагрузкой. Они утверждают, что, создавая «умеренную» вычислительную нагрузку, «когда данных достаточно для вычисления вариограмм, кригинг является хорошим интерполятором разреженных данных. Двоичные и дискретные данные можно интерполировать методом индикаторного кригинга. Можно также включать качественную информацию, например тренды или стратификацию. Для интерполяции многомерных данных применяется кокригинг». Далее в этой книге мы рассмотрим другие методы, применяемые для интерполяции. Один из них – диаграммы Вороного, когда предполагается, что во всех точках, расположенных ближе к данной известной точке, чем к остальным известным точкам, значение такое же, как в данной точке.

Еще один вопрос, заслуживающий обсуждения, – тип интерполируемых данных. Мы рассматривали ситуацию, когда известны результаты наблюдений в некоторых точках, а требуется вычислить значения в других точках. А что, если речь идет о линиях? Например, методы интерполяции уже давно применяются в картографии для построения изолиний (Xiao and Murray, 2015) как способа представления ландшафта.

Следует также подумать о временном аспекте: много ли машинного времени потребляют методы интерполяции? Во внешнем цикле алгоритмов каждая точка обрабатывается только один раз, поэтому временная сложность линейная. Однако когда речь заходит о подготовке данных, то нагрузка оказывается заметно больше: нам нужно найти N ближайших точек к интерполируемой. В этой главе мы использовали полный перебор. Но впоследствии мы увидим, что методы индексирования пространственных данных могут значительно улучшить производительность.

Наконец, мы заметили, что алгоритм смещения средней точки порождает поверхность, для которой изменение в близких точках схоже и определяется стандартным отклонением используемого нормального распределения. Поверхность, созданная таким способом, обладает свойством самоподобия – если увеличить небольшую ее часть, то будут видны те же черты, что присущи всей области. Существует огромный пласт литературы, посвященной этому явлению – фракталам (Mandelbrot, 1967, 1977a,b; Peitgen et al., 1992). Показатель, применяемый для измерения самоподобия, называется фрактальной размерностью; в нашем методе он просто равен $D = 3 - H$. Эти числа приведены для каждой из поверхностей на рис. 8.17.

8.6. Упражнения

1. В код кригинга в этой главе не включена степенная модель. Напишите на Python код для этой модели и сравните результаты с другими моделями, используя те же тестовые данные.

2. Мы можем попробовать применить обратное конструирование для изучения искусственной поверхности, сгенерированной методом смещения средней точки. Идея в том, чтобы сначала создать поверхности этим методом. Затем отложить в сторону несколько точек, принадлежащих поверхности, и применить метод кригинга или ОВР для интерполяции значений в этих точках. Насколько близки интерполированные значения к истинным? В случае кригинга какая модель лучше всего аппроксимирует данные?
3. Оба рассмотренных в этой главе метода кригинга (простой и обыкновенный) не годятся, когда в данных присутствует некий тренд. Тренд можно аппроксимировать полиномиальной функцией, например кубической или даже линейной. Вычитание значения тренда из исходного значения возвращает ошибку, которую затем можно смоделировать с помощью метода кригинга. Это называется универсальным кригингом. Почитайте про универсальный кригинг (Cressie, 1991) и напишите на Python реализующую его программу.

Глава 9

Пространственные паттерны и их анализ

> Взрослые очень любят цифры. Когда рассказываешь им, что у тебя появился новый друг, они никогда не спросят о самом главном. Никогда они не скажут: «А какой у него голос? В какие игры он любит играть? Ловит ли он бабочек?» Они спрашивают: «Сколько ему лет? Сколько у него братьев? Сколько он весит? Сколько зарабатывает его отец?» И после этого воображают, что узнали человека.
>
> Когда говоришь взрослым: «Я видел красивый дом из розового кирпича, в окнах у него герань, а на крыше голуби», – они никак не могут представить себе этот дом. Им надо сказать: «Я видел дом за сто тысяч франков», – и тогда они восклицают: «Какая красота!»
>
> *Антуан де Сент-Экзюпери «Маленький принц»*

Кое-что мы можем сказать о распределении пространственных данных. Так уж получилось, что в точном соответствии с прелестной книжкой «Маленький принц» мы вынуждены вести себя как взрослые, которым больше интересны цифры, чем иные описания. Как правило, нам необходимо число или набор чисел, чтобы описать наблюдаемые характеристики пространственных данных, это и называется пространственным анализом. Круг интересов пространственного анализа широк, но мы ограничимся очень специальной областью: пространственными паттернами. На рис. 9.1 выделена группа деревьев в Овале университетского кампуса в штате Огайо. Случайно ли распределены эти деревья? Но прежде чем отвечать на этот вопрос, давайте поймем, почему так важно знать, распределены ли деревья случайно. Быть может, в данном случае кому-то просто любопытны принципы ландшафтного дизайна. Однако у подобных вопросов есть и более важные причины. Допустим, что у нас имеется информация о местах преступлений, тогда очень важно, случайны ли эти места или в их расположении есть какая-то закономерность. Если точки распределены не случайно, то, возможно, они группируются в кластеры, поэтому в одних местах больше шансов встретить дополнительные точки, чем в других. Отсюда вытекает необходимость дальнейшего исследования с целью понять причины и должным образом обработать события.

Существует много методов и алгоритмов, предназначенных для анализа паттернов в данных разных типов. Сначала мы познакомимся с двумя ме-

тодами анализа точечных паттернов. А затем перейдем к паттернам в явлениях, обладающих пространственной протяженностью, иначе говоря, площадных. Важное применение анализ паттернов находит в ландшафтной экологии, и мы обсудим одну из метрик в этой области. Затем расширим обсуждение паттернов и познакомимся с методом кластеризации, который поможет выявлять кластеры в пространственных данных.

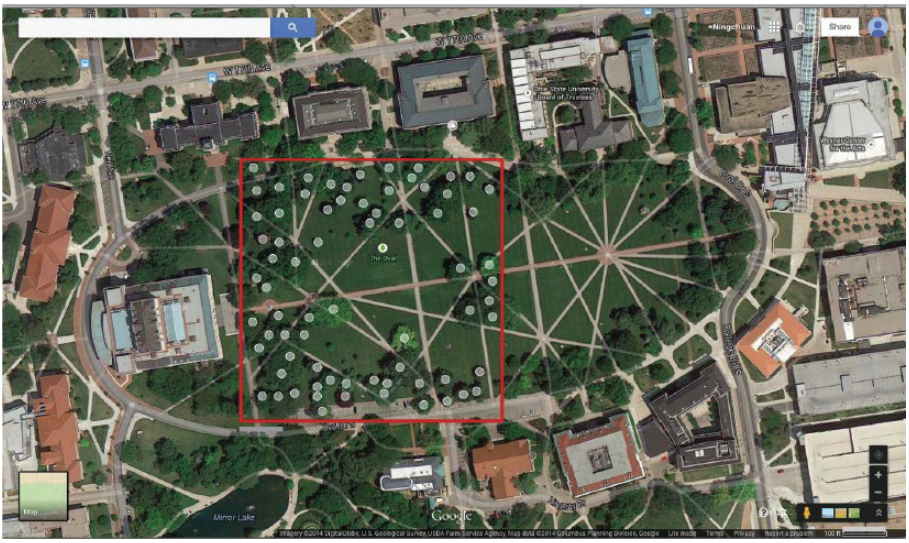

Рис. 9.1 ❖ Деревья в Овале университетского кампуса в штате Огайо. Деревья идентифицированы путем визуального обследования растительных насаждений на изображении

9.1. Анализ точечных паттернов

Иногда нас интересует только сам факт возникновения событий, а не ассоциированные с ними значения. Например, если речь идет о преступлении, то важно его место, а не тип преступления и не причиненный им ущерб. Или же нас могут интересовать места произрастания деревьев, а не их высота. В таких случаях говорят об анализе точечных паттернов. Для этой цели разработаны различные методы. В этом разделе мы расскажем о двух из них.

9.1.1. Анализ ближайшего соседа

Пусть $P = \{p_1, p_2, \ldots, p_n\}$ – множество точек, а d_i – расстояние между i-й точкой и ее ближайшим соседом. Вычислим среднее расстояние до ближайшего соседа для всех точек: $R_0 = (1/n)\sum_{i=1}^{n} d_i$. Каково ожидаемое значение R_0? Рассмотрим ситуацию, когда все точки случайным образом распределены в области. Каким будет среднее расстояние до ближайшего соседа в этом случае?

Конечно, это зависит от количества точек или от плотности точек в области. Предположим, что место расположения точки в области полностью случайно, т. е. не зависит от других точек и подчиняется распределению Пуассона. Для любой точки в пространстве вероятность, что в радиусе x от нее нет ни одной точки, равна $e^{-\lambda\pi x^2}$, где λ – плотность точек в области. Поэтому вероятность, что расстояние до ближайшего соседа X меньше радиуса x, равна $\Pr(X \leqslant x) = 1 - e^{-\lambda\pi x^2}$. Таким образом, πX^2 имеет экспоненциальное распределение, и можно показать, что математическое ожидание расстояния до X равно $1/(2\sqrt{\lambda})$, а его дисперсия равна $(4 - \pi)/4\lambda\pi$.

Предположим, что в области площадью A имеется n точек. Плотность точек равна $\lambda = n/A$, поэтому ожидаемое среднее расстояние до ближайшего соседа равно $R_e = 1/2\sqrt{A/n}$, и мы можем вычислить статистику расстояния до ближайшего соседа

$$R = \frac{R_0}{R_e}.$$

Если R много больше 1, то точки рассеяны, а если много меньше 1, то образуют кластер. Если же значение R близко к 1, то мы ожидаем встретить случайный паттерн. Но что такое «много»? Поскольку мы знаем среднее и дисперсию, то можем стандартизовать R следующим образом:

$$z = \frac{R - 1}{\sqrt{(4 - \pi)/4\lambda\pi}}.$$

Чтобы оценить паттерн, можно воспользоваться t-критерием. При уровне доверительной вероятности 0.95 значение z, меньшее –1.96, означает, что имеется кластер, значение, большее 1.96, – что точки рассеяны, а промежуточные значения – что паттерн случайный.

Программа в листинге 9.1 вычисляет статистику расстояния до ближайшего соседа. В функции `nnd` (строка 12) используется разработанное ранее kD-дерево для индексирования точек (строка 21) и алгоритм поиска ближайшего соседа для каждой точки (строка 25). Поскольку алгоритм индексирования и поиска уже были реализованы в разделе 5.1, код вычисления статистики получился простым и коротким. В результате возвращаются среднее и ожидаемое расстояние до ближайшего соседа (соответственно R_0 и R_e), а также их отношение (R), дисперсия и стандартизованная оценка z.

Листинг 9.1 ❖ Расстояние до ближайшего соседа (nnd.py)

```
 1 from math import sqrt, pi
 2 import numpy as np
 3 import random
 4 import sys
 5 sys.path.append('../geom')
 6 sys.path.append('../indexing')
 7 from point import *
 8 from extent import *
 9 from kdtree1 import *
10 from kdtree3 import *
```

```
11
12 def nnd(points, density):
13     """
14     Расстояние до ближайшего соседа.
15     Вход
16       points: список объектов типа Point
17       density: плотность точек
18     Выход
19         R0, Re, R, Var, z
20     """
21     tree = kdtree2(points)
22     R0 = 0.0
23     for p in points:
24         neighbor = kdtree_nearest_neighbor_query(tree,
25                                                  p, 2)[1][0]
26         R0 += p.distance(neighbor)
27     R0 /= len(points)
28     Re = 1.0/(2.0*sqrt(density))
29     R = R0/Re
30     Var = (4-pi)/(4*density*pi)
31     z = (R-1)/sqrt(Var)
32     return R0, Re, R, Var, z
33
34 def test():
35     n = 100
36     xmin = 10
37     ymin = 10
38     xmax = 40
39     ymax = 40
40     area = Extent(xmin=xmin, xmax=xmax, ymin=ymin, ymax=ymax)
41     density = float(n)/area.area()
42
43     # регулярное
44     nrow = 10
45     ncol = 10
46     dhorizontal = float((ymax-ymin))/ncol
47     dvertical = float((xmax-xmin))/nrow
48     xs = [ ymin + dvertical/2.0 + dvertical*i
49            for i in range(ncol) ]
50     ys = [ xmin + dvertical/2.0 + dvertical*i
51            for i in range(nrow) ]
52     points = [ Point(x, y) for x in xs for y in ys]
53     R0, Re, R, Var, z = nnd(points, density)
54     print "Регулярное:", R0, Re, R, z
55
56     # случайное
57     points = [ Point(random.uniform(xmin, xmax),
58                      random.uniform(ymin, ymax))
59                for i in range(n) ]
60     R0, Re, R, Var, z = nnd(points, density)
61     print "Случайное:", R0, Re, R, z
62
```

```
63     # кластерное
64     points = [ Point(random.gauss(25, 2),
65                      random.gauss(25, 2))
66                for i in range(n) ]
67     R0, Re, R, Var, z = nnd(points, density)
68     print "Гауссово:", R0, Re, R, z
69
70 if __name__ == "__main__":
71     test()
```

В функции `test` мы применяем функцию `nnd` к трем наборам данных: регулярному, когда точки расположены в узлах регулярной сетки (строка 52), случайному, когда координаты X и Y каждой точки выбираются из равномерного распределения (строка 59), и кластерному, когда точки сконцентрированы вокруг центра круговой области, а их координаты выбираются из гауссова распределения (строка 66). Первый случай можно рассматривать как паттерн рассеяния – точки, говоря метафорически, стараются держаться друг от друга подальше. Программа дает следующие результаты, где напечатанные значения – это соответственно R_0, R_e, R и z.

```
$ python nnd.py
Регулярное: 3.0869848481 1.5 2.05798989873 1.34933096134
Случайное: 1.81649019531 1.5 1.21099346354 0.269095209073
Гауссово: 0.438146681302 1.5 0.292097787535 -0.90283883998
```

Результаты тестирования выглядят разумно, поскольку в регулярном случае (рассеяние) расстояние до ближайшего соседа велико, а в кластерном очень мало по сравнению со случайным. Но оценки z сравниваются с теоретически случайным распределением, и в какой мере мы можем доверять теоретическому выводу среднего расстояния? Сначала рассмотрим ситуацию неформально, чтобы понять смысл ожидаемого расстояния до ближайшего соседа, когда точки случайно распределены. Точнее, мы хотим знать, какова средняя площадь области, в которой мы ожидаем найти точку. Если в области площадью A находится n точек, то мы ожидаем найти точку в области, площадь которой A/n обратно пропорциональна плотности, $1/\lambda = A/n$. Если это квадрат, то длина его стороны равна $1/\sqrt{\lambda}$. Предположим, что точка находится в центре квадрата; где тогда находится ближайшее место, в котором мы ожидаем найти другую точку? В середине любой стороны, на расстоянии $1/\sqrt{\lambda}$ от центра квадрата.

Для случайно выбранной точки это неформальное рассуждение показывает, что точка расположена в центре квадрата со стороной $1/\sqrt{\lambda}$. Но то же самое верно для остальных точек: все они расположены в центрах своих собственных квадратов. Но ведь тогда расстояние между точками равно $1/\sqrt{\lambda}$, а не $1/(2\sqrt{\lambda})$, разве не так? Звучит убедительно, но цифры не совсем правильные. Мы не станем здесь развивать тему теоретического вывода, отчасти чтобы избежать бесконечных споров о смысле полной пространственной случайности и вытекающих из нее следствиях. Вместо этого мы можем вычислить, как должно выглядеть среднее расстояние. Воспользуемся методом Монте-Карло и выполним много случайных испытаний; в каждом испытании точки

размещаются в области случайным образом[1], и значение R_0 вычисляется эмпирически. Эта идея реализована в функции `nnd_monte_carlo` в листинге 9.2. По умолчанию производится 10 000 испытаний, но при запуске программы можно задать другое число. Функция возвращает 2.5-й, 50-й и 97.5-й процентили, что дает хорошую оценку медианы (50-й процентиль) и доверительного интервала.

Листинг 9.2 ❖ Вычисление расстояния до ближайшего соседа методом Монте-Карло (nnd_monte_carlo.py)

```
from nnd import *

def nnd_monte_carlo(n, area, verbal=False, N=10000):
    Rm = []
    density = float(n)/area.area()
    for i in range(N):
        points = [ Point(random.uniform(area.xmin, area.xmax),
                         random.uniform(area.ymin, area.ymax))
                   for i in range(n) ]
        Rm.append(nnd(points, density)[0])
    Rm = np.array(Rm)
    if verbal:
        for r in Rm:
            print r
    return np.percentile(Rm, 2.5),\
        np.percentile(Rm, 50),\
        np.percentile(Rm, 97.5)

if __name__ == "__main__":
    n = 100
    xmin = 10
    ymin = 10
    xmax = 40
    ymax = 40
    area = Extent(xmin=xmin, xmax=xmax, ymin=ymin, ymax=ymax)
    Rm = nnd_monte_carlo(n, area)
    print Rm
```

Эта программа печатает следующий результат:

```
(1.5492612728187518, 1.7516694945432549, 1.9620854828335306)
```

Видно, что эмпирическое среднее расстояние до ближайшего соседа для 100 точек в области равно не 1.5, а приблизительно 1.75, что согласуется с рассуждением выше. Видно также, что для случайно выбранных точек среднее расстояние до ближайшего соседа хорошо укладывается в 95%-ный доверительный интервал, тогда как при кластерном и рассеянном распределениях выходит за границы этого интервала. Задав параметр `verbal` равным `True`,

[1] На самом деле псевдослучайным, поскольку в вычислениях используется генератор псевдослучайных чисел, встроенный в Python.

мы сможем распечатать все расстояния для каждой из 10 000 выборок. Мы построили гистограмму этих расстояний (рис. 9.2), которая дает наглядное представление о связях.

Теперь протестируем статистику расстояний до ближайшего соседа на примере деревьев в Овале (рис. 9.1). Сначала оцифруем все помеченные деревья, используя их экранные координаты. Полученные 72 пары координат сохранены в списке `rawpoints` в листинге 9.3. Следует отметить, что деревья идентифицированы приближенно, поскольку некоторые купы расположены слишком близко друг к другу и разделить их на отдельные деревья трудно; с другой стороны, некоторые маленькие деревья вообще не включены. Далее в программе производится преобразование экранных координат (причем *Y* увеличивается сверху вниз) в обычную евклидову систему координат, начало которой расположено в левом нижнем углу, а область, занятая деревьями, растянута в квадрат 6×6, включая поля шириной 0.1 вдоль каждой стороны. В последней строке эти координаты преобразуются в список объектов `Point`, используемых в дальнейших вычислениях.

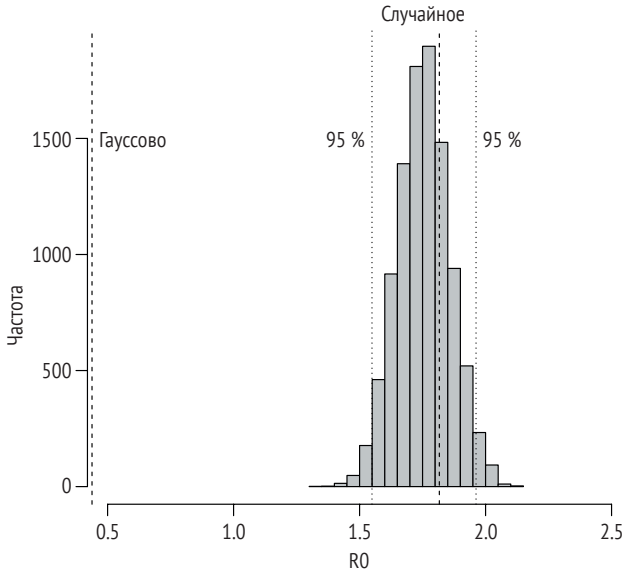

Рис. 9.2 ❖ Гистограмма средних расстояний до ближайшего соседа, построенная по результатам 10 000 обсчетов модели со 100 случайными точками, координаты *X* и *Y* которых изменяются в диапазоне от 10 до 40. Две жирные пунктирные прямые представляют два набора данных, содержащих по 100 точек вокруг точки (25, 25), выбранных из гауссова распределения (с дисперсией 2 для обеих координат) и из равномерного распределения во всей области

Листинг 9.3 ❖ Точки расположения деревьев в Овале (oval_trees.py)

```
import sys
sys.path.append('../geom')
from point import *

```

```
# экранные координаты точек
rawpoints = [
    [3.99, 4.15], [3.81, 3.33], [5.26, 2.98], [3.64, 4.46],
    [4.86, 4.50], [3.33, 4.22], [3.01, 4.30], [3.21, 4.46],
    [3.09, 4.62], [3.40, 4.46], [3.36, 4.93], [3.33, 5.21],
    [3.52, 5.17], [3.13, 5.21], [5.11, 2.66], [3.80, 5.21],
    [3.80, 5.09], [3.96, 5.01], [3.80, 4.85], [4.15, 5.05],
    [4.14, 5.21], [4.61, 5.17], [4.65, 5.05], [4.49, 5.01],
    [4.81, 4.85], [5.04, 5.05], [5.36, 5.00], [5.38, 5.26],
    [5.58, 5.18], [5.74, 5.14], [5.78, 4.91], [5.66, 4.16],
    [5.54, 3.65], [5.38, 2.74], [3.65, 2.74], [4.79, 3.10],
    [4.44, 3.10], [4.52, 2.98], [4.67, 2.70], [4.36, 2.86],
    [3.93, 2.94], [4.08, 2.82], [4.16, 2.66], [4.32, 2.51],
    [3.96, 2.63], [3.69, 2.55], [3.33, 2.66], [3.33, 2.98],
    [3.06, 3.02], [3.06, 3.77], [3.18, 3.89], [3.26, 3.53],
    [3.53, 3.57], [4.67, 2.43], [4.95, 2.55], [5.03, 2.43],
    [5.89, 3.61], [5.93, 3.81], [5.89, 4.04], [5.93, 4.24],
    [5.66, 2.53], [5.42, 2.53], [5.89, 2.69], [5.70, 2.96],
    [5.09, 5.31], [3.86, 5.39], [3.46, 4.72], [3.70, 4.25],
    [3.14, 3.29], [3.02, 2.43], [3.33, 3.33], [3.06, 2.70] ]

ymax = max([p[1] for p in rawpoints])
ymin = min([p[1] for p in rawpoints])
xmin = min([p[0] for p in rawpoints])
xmax = max([p[0] for p in rawpoints])

# Прибавить 0.1 с каждой стороны, чтобы получился квадрат 6×6
xratio = 5.8/(xmax-xmin)
yratio = 5.8/(ymax-ymin)

# Преобразование экранных координат в декартовы
points = [ Point(xratio*(p[0]-xmin)+0.1,
                 yratio*(ymax-p[1])+0.1)
           for p in rawpoints]
```

Построив карты деревьев, мы можем воспользоваться кодом в листинге 9.4, чтобы вычислить статистику расстояния до ближайшего соседа. Кроме того, мы хотим проверить результат эмпирически с помощью метода Монте-Карло.

Листинг 9.4 ❖ Тестирование расстояния до ближайшего соседа (test_oval_trees_nnd.py)

```
from oval_trees import *
from nnd import *
from nnd_monte_carlo import *

area = Extent(xmin=0.0, xmax=6.0, ymin=0.0, ymax=6.0)
density = len(points)/area.area()
R0, Re, R, Var, z = nnd(points, density)
print R0, Re, R, Var, z
Rm = nnd_monte_carlo(len(points), area)
print Rm
```

Для деревьев программа печатает следующие результаты:

```
0.458288702278 0.353553390593 1.29623619649 0.0341549430919
  1.60291812584
(0.35449964236594012, 0.41160208470080117, 0.4703053746438054)
```

Мы также нанесли результаты на гистограмму (рис. 9.3). Среднее расстояние составляет приблизительно 0.4583, т. е. немного меньше 95-го процентиля из моделирования методом Монте-Карло. Таким образом, распределение деревьев, скорее всего, случайно. Хотя абсолютной уверенности в случайности у нас нет, мы можем точно сказать, что это не кластерный паттерн, потому что в противном случае среднее расстояние должно было бы находиться в другой (левой) части гистограммы. Если мы готовы смириться с уровнем доверия меньше 95 %, то можем даже считать паттерн рассеянным. Этот результат имеет смысл с точки зрения ландшафтного дизайна: хорошо, когда деревья растут на значительной части общественного пространства, каковым является Овал.

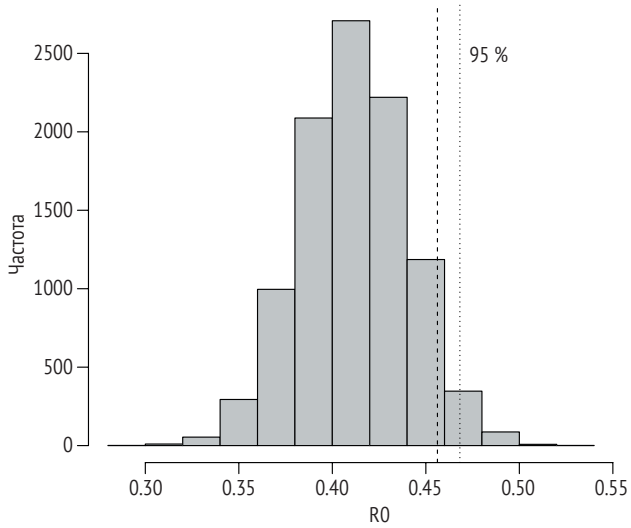

Рис. 9.3 ❖ Гистограмма средних расстояний до ближайшего соседа для 10 000 обсчетов модели на 72 деревьях в Овале

9.1.2. *K*-функция Рипли

Прежде чем переходить к другой теме, поразмышляем о принципе, заложенном в статистику расстояний до ближайшего соседа. Почему мы смогли сделать выводы о паттерне расположения множества точек: случайном, рассеянном или кластерном? Потому что можем сказать, насколько близко расположены точки и насколько близко они должны быть расположены, чтобы считаться отвечающими какому-то паттерну. Но предположим, что вместо фиксированного способа измерения близости точек (на основе расстояния

до ближайшего соседа) мы нарисовали окружность с центром в точке и смотрим, сколько точек реально попало в эту окружность и сколько должно было бы попасть, будь распределение случайным. Можно также изучить паттерн при разных пространственных масштабах, изменяя радиус окружности. При таком сравнении можно прийти к определенным выводам. Эта идея легла в основу K-функции Рипли.

Обозначим λ плотность точек в интересующей нас области. Для каждой точки проводим окружность радиуса d с центром в этой точке и подсчитываем число n точек внутри окружности (включая саму данную точку). Определим метрику $K(d) = n/\lambda$. Каково ее ожидаемое значение, если точки распределены в области случайно? Площадь круга равна πd^2. Если площадь всей области равна A и в ней расположено N точек, то круг занимает долю $\pi d^2/A$ этой области. Если точки распределены случайно, то следует ожидать, что внутри окружности окажется $n = N\pi d^2/A$ точек, поэтому $K(d) = (N\pi d^2/A)/\lambda$. Поскольку $\lambda = N/A$, получаем, что для случайного паттерна $K(d) = \pi d^2$. Формально можно определить K-функцию следующим образом:

$$L(d) = \sqrt{\frac{K(d)}{\pi}}.$$

Затем мы можем увеличить радиус окружности и повторить вычисления, чтобы установить связь между $L(d)$ и радиусом. Для случайного паттерна следует ожидать, что $L(d) = d$, т. е. график зависимости $L(d)$ от d будет прямой с углом наклона 45°. Если точек много, то мы можем вычислить $L(d)$ для каждой точки и усреднить по всем точкам для каждого значения d. Возможность исследовать зависимость K-функции от радиуса позволяет получить представление о распределении точек при разных масштабах. Если значение $L(d)$ меньше d, значит, мы имеем кластерный паттерн, когда точек больше, чем ожидалось. Если $L(d)$ больше d, то фактическое число точек меньше ожидаемого, так что налицо рассеянный паттерн. Если же $L(d)$ равно d, то мы имеем случайный паттерн.

Чтобы эмпирически проверить, соответствует ли значение $L(d)$ случайному паттерну, мы можем воспользоваться моделированием методом Монте-Карло. Идея в том, чтобы сгенерировать большое число случайных паттернов, для каждого из них вычислить значения $L(d)$ и для каждого d получить 2.5-й и 97.5-й процентили, дающие 95%-ный доверительный интервал.

Вычислить $L(d)$ и выполнить испытания Монте-Карло нетрудно, особенно если воспользоваться kD-деревом и поиском в круговом диапазоне. В листинге 9.5 приведены функции для обеих задач (строки 12 и 31 соответственно). Заметим, что функция `kfunc` принимает на входе индексное дерево. В отличие от программы вычисления расстояния до ближайшего соседа, в которой дерево создавалось внутри функции `nnd`, здесь мы требуем, чтобы дерево было создано вне функции `kfunc`. Связано это с тем, что вычисление $L(d)$ повторяется многократно для разных значений d, а создать дерево для заданного множества точек нужно только один раз. Точнее, параметрами функции `kfunc` являются kD-дерево, исследуемая точка, радиус окружности и плотность точек в области (строка 12). Найдя точки, оказавшиеся внутри

окружности (строка 25), уже нетрудно вычислить значение $L(d)$. Функция моделирования методом Монте-Карло `kfunc_monte_carlo` принимает на входе список значений радиуса.

Листинг 9.5 ❖ *K*-функция (kfunction.py)

```
 1 import sys
 2 sys.path.append('../geom')
 3 from point import *
 4 sys.path.append('../indexing')
 5 from extent import *
 6 from kdtree1 import *
 7 from kdtree2b import *
 8 import random
 9 from math import sqrt, pi
10 import numpy as np
11
12 def kfunc(tree, p, d, density):
13     """
14     Вход
15       tree: kD-дерево
16       p: точка, в которой вычисляется K-функция
17       d: радиус окружности с центром в p
18       density: плотность точек в этой области
19
20     Выход
21       n: количество точек внутри окружности
22       ld: значение L(d)
23     """
24     neighbors = []
25     range_query_circular(tree, p, d, neighbors)
26     n = len(neighbors)
27     kd = n/density
28     ld = sqrt(kd/pi)
29     return n, ld
30
31 def kfunc_monte_carlo(n, area, radii, density, rounds=100):
32     """
33     Вход
34       n: число точек
35       area: объект Extent, определяющий область
36       radii: список радиусов окружностей
37       rounds: число испытаний
38     Выход
39       percentiles: список 2.5-х и 97.5-х процентилей
40                    для каждого d в списке radii
41     """
42     alllds = []
43     for test in range(rounds):
44         points = [ Point(random.uniform(area.xmin, area.xmax),
45                          random.uniform(area.ymin, area.ymax))
46                    for i in range(n) ]
```

```
    t = kdtree2(points)
    lds = [0 for d in radii]
    for i, d in enumerate(radii):
        for p in points:
            ld = kfunc(t, p, d, density)[1]
            lds[i] += ld
        lds = [ld/n for ld in lds]
        alllds.append(lds)
    alllds = np.array(alllds)
    percentiles = []
    for i in range(len(radii)):
        percentiles.append([np.percentile(alllds[:,i], 2.5),
                            np.percentile(alllds[:,i], 97.5)])
    return percentiles

def test(points, area):
    """
    Вход
      points: список объектов Point
      area: объект Extent

    Выход
      ds: список радиусов
      percentiles: результат моделирования Монте-Карло для радиусов из ds
      lds: значения L(d) для каждого радиуса из ds
    """
    n = len(points)
    density = n/area.area()
    t = kdtree2(points)
    d = min([area.xmax-area.xmin,area.ymax-area.ymin])*2/3/10
    ds = [ d*(i+1) for i in range(10)]
    lds = [0 for d in ds]
    for i, d in enumerate(ds):
        for p in points:
            ld = kfunc(t, p, d, density)[1]
            lds[i] += ld
    lds = [ld/n for ld in lds]
    percentiles = kfunc_monte_carlo(n, area, ds, density)
    return ds, percentiles, lds

    if __name__ == "__main__":
        n = 100
        # случайный паттерн
        points = [ Point(random.uniform(20, 30),
                         random.uniform(30, 40))
                   for i in range(n) ]
        xmin = min([p.x for p in points])
        ymin = min([p.y for p in points])
        xmax = max([p.x for p in points])
        ymax = max([p.y for p in points])
        area = Extent(xmin=xmin, xmax=xmax,
                      ymin=ymin, ymax=ymax)
```

```
 99          ds, percentiles, lds = test(points, area)
100        print "Случайный"
101        for i, v in enumerate(percentiles):
102            print ds[i], v[0], lds[i], v[1]
103
104
105        # три блока точек
106        points1 = [ Point(random.uniform(10, 20),
107                          random.uniform(10, 20))
108                    for i in range(n/3) ]
109        points2 = [ Point(random.uniform(30, 40),
110                          random.uniform(10, 20))
111                    for i in range(n/3) ]
112        points3 = [ Point(random.uniform(20, 30),
113                          random.uniform(30, 40))
114                    for i in range(n/3) ]
115        points = points1 + points2 + points3
116        xmin = min([p.x for p in points])
117        ymin = min([p.y for p in points])
118        xmax = max([p.x for p in points])
119        ymax = max([p.y for p in points])
120        area = Extent(xmin=xmin,xmax=xmax,ymin=ymin,ymax=ymax)
121        ds, percentiles, lds = test(points, area)
122        print "Три блока"
123        for i, v in enumerate(percentiles):
124            print ds[i], v[0], lds[i], v[1]
```

Функция `test` в листинге 9.5 вычисляет значения $L(d)$ для 10 радиусов, при этом максимальный радиус равен двум третям от длины меньшей стороны области (строки 76 и 77). Для каждого d используется среднее значение $L(d)$ (строка 83). Для прогона теста мы взяли два набора данных (рис. 9.4): в первом наборе точки распределены случайно, а во втором имеется три кластера (но в каждом кластере распределение точек случайное).

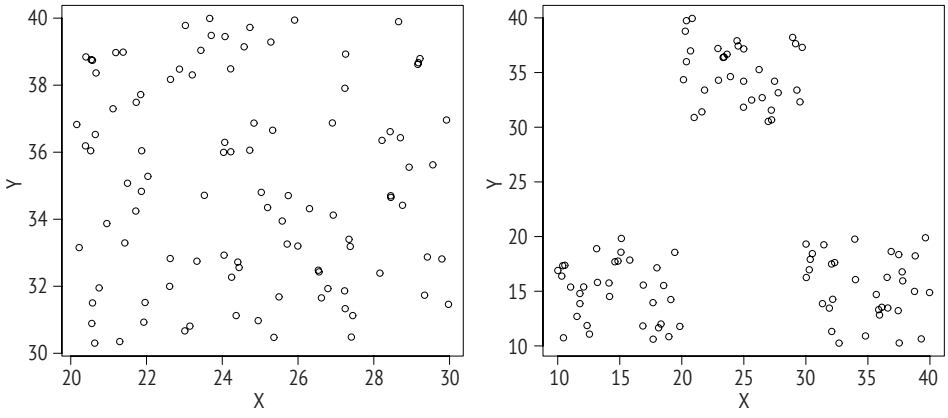

Рис. 9.4 ❖ Случайные (слева) и объединенные в кластеры (справа) точки для тестирования *K*-функции

Результат показан ниже. На рис. 9.5 мы нанесли на график диагональ, доверительный интервал и значения *K*-функции. Для случайного паттерна видно, что значения *K*-функции точно укладываются между двумя границами доверительного интервала, т. е. при всех масштабах паттерн признан случайным. Но набор данных с тремя блоками демонстрирует интересную динамику пространственного паттерна. При небольшом (локальном) масштабе, т. е. когда исследуется близкая окрестность каждой точки, мы наблюдаем кластерный паттерн, потому что точек оказывается больше, чем ожидалось (с учетом размера области). Но когда радиус окружности превышает некоторый порог (приблизительно 10), паттерн все больше напоминает рассеянный. Заметим, что размер каждого блока точек в этом случае составляет 10 единиц длины, что и объясняет, почему при увеличении масштаба наблюдается рассеянный паттерн. Этот пример помогает понять преимущества *K*-функции: она показывает, как пространственный паттерн зависит от масштаба. Статистика расстояний до ближайшего соседа этим свойством не обладает.

```
Случайный
0.658746284775 0.782447251926 0.823192147599 0.878923490623
1.31749256955 1.25294384468 1.29708270824 1.41427475067
1.97623885433 1.76164558885 1.80567490505 1.93508213498
2.6349851391 2.25189975889 2.30911526409 2.45740489315
3.29373142388 2.68641955296 2.81525363907 2.93901601286
3.95247770865 3.07522982843 3.24727346436 3.38607862897
4.61122399343 3.44377836531 3.64967726331 3.82597112471
5.2699702782 3.7986238341 4.00592538201 4.20939096454
5.92871656298 4.1270080257 4.33537220243 4.54396121198
6.58746284775 4.39711802739 4.64973313045 4.82682955853
Три блока
1.97215401624 2.30732019217 3.35751451614 2.61282902369
3.94430803248 3.80730296142 5.50894499646 4.198217178
5.91646204871 5.32284684915 7.31791426457 5.84865572826
7.88861606495 6.6964392654 8.68481246242 7.37419127158
9.86077008119 8.0705796119 9.56165773421 8.85356120542
11.8329240974 9.31989267977 9.71540908647 10.2497400438
13.8050781137 10.477205022 10.0085157011 11.4893468682
15.7772321299 11.5205533322 10.5218604296 12.6373133211
17.7493861461 12.5309706034 11.2422919216 13.6603894493
19.7215401624 13.4235638982 12.319968568 14.5096243377
```

Наконец, в листинге 9.6 мы вычисляем значения *K*-функции для деревьев в Овале. Результаты показаны на рис. 9.6, и из них следует, что при очень мелком масштабе кривая *K*-функции находится внутри доверительного интервала. Можно сравнить результаты, полученные с помощью *K*-функции, и расстояния до ближайшего соседа, которые указывают на случайный паттерн. Напомним, что во втором случае статистика вычислялась на основе расстояния до ближайшего соседа, который, по идее, должен попадать в окружность малого радиуса. Ясно, что *K*-функция говорит больше о том, как точки организованы в пространстве. Следует также отметить, что у радиуса окружности есть предел. Мы не хотим увеличивать окружности до бесконечности. В приведенной выше программе мы ограничили радиус двумя третями меньшей стороны области.

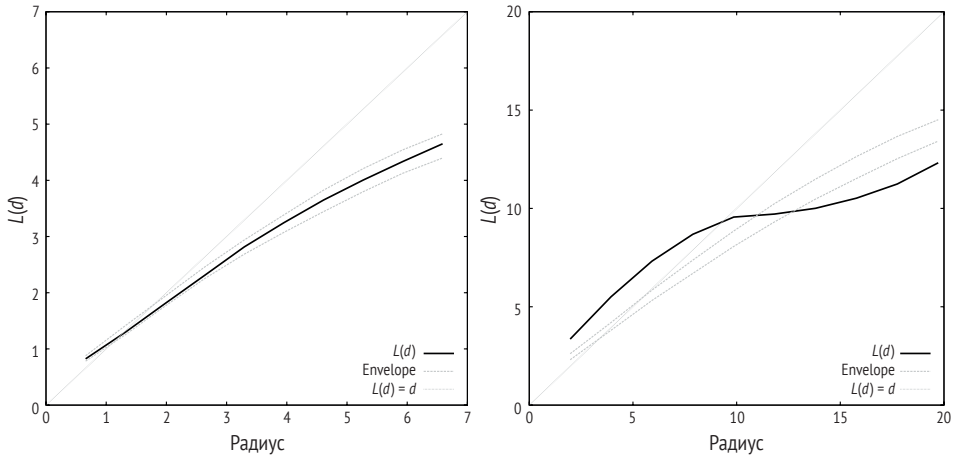

Рис. 9.5 ❖ Значения *K*-функции для наборов данных со случайными точками (слева) и с тремя кластерами (справа)

Листинг 9.6 ❖ *K*-функция для деревьев в Овале (test_oval_trees.py)

```
 1 import sys
 2 sys.path.append('../indexing')
 3 from extent import *
 4 from kfunction import test
 5 from oval_trees import *
 6
 7 area = Extent(xmin=0.0, xmax=6.0, ymin=0.0, ymax=6.0)
 8 ds, percentiles, lds = test(points, area)
 9 for i, v in enumerate(percentiles):
10     print ds[i], v[0], lds[i], v[1]
```

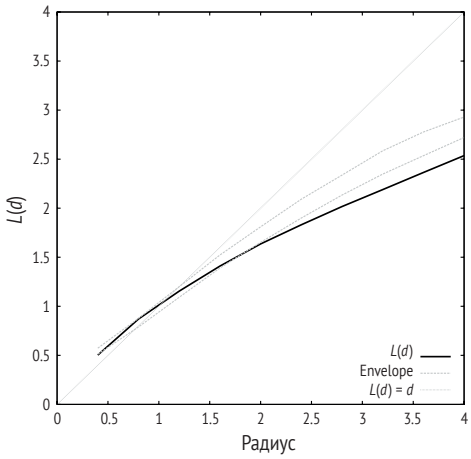

Рис. 9.6 ❖ Значения *K*-функции для деревьев в Овале

9.2. Пространственная автокорреляция

В предыдущем разделе мы говорили о том, как анализировать паттерны точек в ситуации, когда нас в основном интересует местоположение каждой точки, а не значение в ней. Но бывают случаи, когда нужно знать, как значения связаны друг с другом в зависимости от положения в пространстве. Например, следует ли ожидать, что область с высоким медианным доходом семьи будет расположена рядом с областями со схожими доходами семьи? Или же наоборот – области с высоким доходом будут соседствовать с областями с низким доходом? Или вообще не будет наблюдаться никакой закономерности – рядом с областями с высоким доходом могут находиться области как с высоким, так и с низким доходом? Когда возникает необходимость рассматривать непрерывные значения, следует использовать другой набор статистик, и одной из самых распространенных, пожалуй, является индекс *I* Морана. Рассмотрим перекрестное произведение отклонений от среднего значений случайной величины в двух точках *i* и *j*, взвешенное с учетом близости этих точек. Среднее по всем парам *i* и *j* затем нормируется на дисперсию случайной величины для всех точек. Формально индекс *I* Морана определяется следующим образом:

$$I = \frac{N\sum_i\sum_j w_{ij}(z_i - \bar{z})(z_j - \bar{z})}{\sum_i\sum_j w_{ij}\sum_i (z_i - \bar{z})^2},$$

где z_i – значение случайной величины в точке *i*, $\bar{z}$ – среднее по всем точкам, N – число точек, а w_{ij} – элемент матрицы весов, назначенный паре точек *i* и *j*.

Программа в листинге 9.7 вычисляет индекс Морана, зная матрицу весов `w` и значения в каждой точке `z`. Мы перебираем все пары точек и, прежде чем вычислять произведение разностей, проверяем, не равен ли вес пары нулю. Это прямолинейный способ, но в нем, похоже, уделяется излишнее внимание парам несоседних точек. Скорее всего, такое вычисление индекса Морана будет занимать много времени.

Листинг 9.7 ❖ Вычисление индекса Морана для заданной матрицы весов (moransi.py)

```
def moransi(z, w):
    """
    Вход
      z: список значений
      w: матрица весов
    Выход
      I: индекс I Морана
    """
    n = len(z)
    mean = float(sum(z))/n
    S0 = 0
    d = 0
    var = 0
    for i in range(n):
```

```
        var += (z[i]-mean)**2
        for j in range(i):
            if w[i][j]:
                S0 += w[i][j]
                d += (z[i]-mean)*(z[j]-mean)
    I = n*d/S0/var
    return I
```

Когда величина представлена многоугольниками и каждому многоугольнику сопоставлен набор значений атрибутов, матрица весов $W = \{w_{ij} | 1 \leqslant i, j \leqslant N\}$ часто определяется тем, являются ли многоугольники смежными. Иными словами, мы полагаем $w_{ij} = 1$, если многоугольники i и j смежные, и $w_{ij} = 0$ в противном случае. В этом случае матрица весов W упрощается и содержит только соседние пары, так что формулу вычисления индекса Морана можно переписать следующим образом:

$$I = \frac{N\sum_{(i,j)\in W}(z_i - \bar{z})(z_j - \bar{z})}{|W|\sum_i (z_i - \bar{z})^2}.$$

Именно так мы и реализуем процесс вычисления I. Более подробное обсуждение того, как назначать веса, приведено в упражнениях.

В программе в листинге 9.8 для вычисления индекса Морана используется список пар соседних точек (`wlist`). Этот метод должен работать быстрее, хотя следует признать, что для построения списка таких пар тоже требуется какое-то время.

Листинг 9.8 ❖ Вычисление индекса *I* Морана для списка соседних точек (moransi2.py)

```
def moransi2(z, wlist):
    """
    Вход
      z: список значений
      w: список весов
    Выход
      I: индекс Морана
    """
    n = len(z)
    d = 0.0
    var = 0.0
    mean = float(sum(z))/n
    for i in range(n):
        var += (z[i]-mean)**2
    S0 = len(wlist)
    for e in wlist:
        d += (z[e[0]]-mean)*(z[e[1]]-mean)
    I = n*d/S0/var
    return I
```

Протестируем этот код на данных переписи 2000 года о плотности населения в 3109 граничащих округах США. В коде в листинге 9.9 используется

матрица смежности, созданная в приложении В.1.5 и сохраненная в формате Python pickle. Этот файл загружается в программу. Затем мы преобразуем матрицу смежности в список (строка 27) и используем ее для вычисления индекса Морана.

Листинг 9.9 ❖ Вычисление индекса *I* Морана с помощью матрицы весов (test_moransi.py)

```
 1 from osgeo import ogr
 2 from numpy import *
 3 import pickle
 4 import time
 5 from moransi import *
 6 from moransi2 import *
 7
 8 driver = ogr.GetDriverByName("ESRI Shapefile")
 9 fname = '../data/uscnty48area.shp'
10 vector = driver.Open(fname, 0)
11 layer = vector.GetLayer(0)
12 n = layer.GetFeatureCount()
13
14 adj = pickle.load(open('../gdal/uscnty48area.adj.pickle'))
15
16 z = []
17 for i in range(n):
18     feature1 = layer.GetFeature(i)
19     z.append(feature1.GetField("PopDensity"))
20
21 t1 = time.time()
22 I = moransi(z, adj)
23 t2 = time.time()
24 print "I =", I
25 print "Time =", t2-t1
26
27 wlist = []
28 for i in range(n):
29     for j in range(i):
30         if adj[i][j]:
31             wlist.append([i, j])
32
33 t3 = time.time()
34 I = moransi2(z, wlist)
35 t4 = time.time()
36 print "I =", I
37 print "Время =", t4-t3
```

Результат выполнения этой программы показан ниже. Бросается в глаза разница во времени работы обоих методов. Ниже мы увидим, что это играет решающую роль при вычислении пространственной автокорреляции.

```
I = 0.239939848606
Время = 2.76858615875
```

```
I = 0.239939848606
Время = 0.00218296051025
```

Если паттерн распределения в пространстве случайный, то математическое ожидание индекса Морана равно

$$E[I] = -\frac{1}{N-1},$$

а дисперсия равна

$$\begin{aligned}\mathrm{var}(I) &= \frac{N^2 S_1 - N S_2 + 3S_0^2}{(N^2-1)S_0^2} - E[I]^2 \\ &= \frac{N^2(N-1)S_1 - N(N-1)S_2 + 2(N-2)S_0^2}{(N+1)(N-1)^2 S_0^2},\end{aligned}$$

где

$$S_0 = \sum_i \sum_j w_{ij};$$

$$S_1 = \frac{1}{2}\sum_i \sum_j (w_{ij} + w_{ji})^2;$$

$$S_2 = \sum_k \left(\sum_j w_{kj} + \sum_i w_{ik}\right)^2.$$

Однако эта формула дисперсии индекса Морана выведена в предположении, что значения независимо выбираются из нормального распределения. Другой вид случайного паттерна получается, когда одни и те же значения тасуются между точками. В таком случае формула дисперсии выглядит иначе:

$$\mathrm{var}(I) = \frac{N S_3 - S_4 S_5}{(N-1)(N-2)(N-3)S_0^2} - E[I]^2,$$

где

$$S_3 = (N^2 - 3N + 3)S_1 - N S_2 + 3S_0^2;$$

$$S_4 = \frac{\frac{1}{N}\sum_i (z_i - \bar{z})^4}{\left[\frac{1}{N}\sum_i (z_i - \bar{z})^2\right]^2};$$

$$S_5 = N(N-1)S_1 - 2N S_2 + 6S_0^2.$$

Дисперсии, конечно, можно вычислить, но это муторно. Мы не станем приводить код здесь, а оставим реализацию в качестве упражнения. Можно вместо этого применить тот же подход, что при вычислении *K*-функции: методом Монте-Карло. Идея в том, чтобы сгенерировать большое число паттернов и для каждого из них вычислить индекс Морана. Затем мы сравниваем индекс, посчитанный по исходным данным и в результате моделирования, и решаем, сильно ли они различаются. Весь код достаточно прост

(листинг 9.10). Мы обсчитываем 10 000 испытаний, что занимает примерно полминуты. На практике было бы нецелесообразно использовать матричный вариант вычисления индекса Морана, если только не включить каких-то приемов, которые значительно повышают эффективность. В конце программы мы с помощью имеющихся в Python средств строим график.

Листинг 9.10 ❖ Моделирование индекса Морана методом Монте-Карло (test_moransi_mc.py)

```
from osgeo import ogr
from numpy import *
import pickle
import time
from random import shuffle
from moransi2 import *
import matplotlib.pyplot as plt

driver = ogr.GetDriverByName("ESRI Shapefile")
fname = '../data/uscnty48area.shp'
vector = driver.Open(fname, 0)
layer = vector.GetLayer(0)
n = layer.GetFeatureCount()

adj = pickle.load(open('../gdal/uscnty48area.adj.pickle'))
z = []
for i in range(n):
    feature1 = layer.GetFeature(i)
    z.append(feature1.GetField("PopDensity"))

I = moransi2(z, wlist)
print I
wlist = []
for i in range(n):
    for j in range(i):
        if adj[i][j]:
            wlist.append([i, j])

ms = []
for i in range(10000):
    shuffle(z)
    ms.append(moransi2(z, wlist))

print percentile(ms, 97.5)
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.hist(ms, 20, color="grey", alpha=0.8)
plt.show()
fig.savefig('moransi-mc.eps')
```

Гистограмма результатов, полученных методом Монте-Карло, показана на рис. 9.7. 97.5-й процентиль равен 0.0223. Видно, что значение индекса Морана для наших данных (0.2399) далеко от этой величины, поэтому мож-

но заключить, что имеется сильная положительная автокорреляция данных о плотности населения на нашей карте.

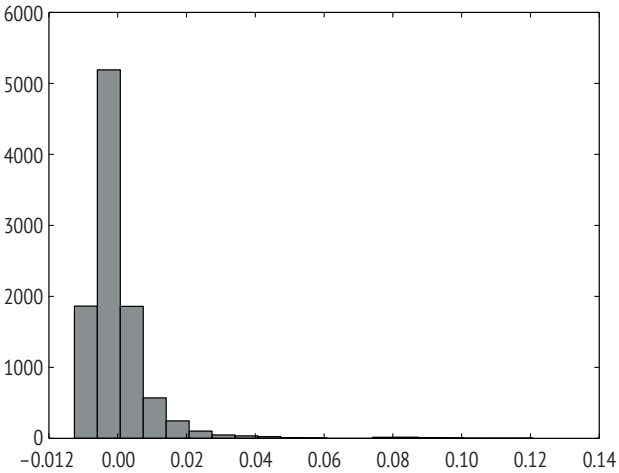

Рис. 9.7 ❖ Гистограмма значений индекса Морана, полученных методом Монте-Карло

Наконец, мы протестировали программу на другом пространственном наборе данных на регулярной сетке (листинг 9.11). Для вычисления индекса Морана используется список соседних пар. Но код создания этого списка в книге не приводится, а составляет содержание упражнения 4 в конце данной главы. Мы же предполагаем, что этот код написан и строит список соседних пар (строки 3 и 11). Сетка содержит 15 строк и 15 столбцов (строка 9). В строке 15 создается случайное множество независимых значений на сетке, а в следующей строке вычисляется индекс Морана. Вычисление повторяется 10 000 раз. Во втором сценарии мы берем один конкретный набор случайных данных и перетасовываем его 10 000 раз (строка 22). То есть тестируем перестановку значений. В третьем сценарии мы создаем специальную конфигурацию значений, когда наибольшее значение 1 находится в центре сетки, а в стороны от него значения убывают в соответствии с формулой $(15/16)[1 - (d/r)^2]^2$, где d – расстояние от ячейки сетки до центра, а параметр r называется шириной полосы. В данном случае ширина полосы равна размеру сетки ($r = 15$). Эта формула обычно называется ядерной функцией с затуханием при увеличении дальности, или бивесовым ядром.

Листинг 9.11 ❖ Тестирование индекса Морана с помощью данных на регулярной сетке (test_moransi_grid.py)

```
1 from numpy import percentile
2 from moransi2 import *
3 from create_wlist import create_wlist # TO DO
4 from random import shuffle
5 from random import random
```

```
from math import sqrt
import matplotlib.pyplot as plt

n1 = 15
n = n1*n1
wlist = create_wlist(n1, n1) # TO DO

ms1 = []
for i in range(10000):
    data = [random() for i in range(n)]
    I = moransi2(data, wlist)
    ms1.append(I)

ms2 = []
data = [random() for i in range(n)]
for i in range(10000):
    shuffle(data)
    I = moransi2(data, wlist)
    ms2.append(I)

i0 = n1/2
j0 = n1/2
for i in range(n1):
    for j in range(n1):
        d = sqrt((i-i0)*(i-i0) + (j-j0)*(j-j0))
        val = 15.0/16.0 * (1 - (d/n1)**2)**2
        pos = i*n1+j
        data[pos] = val

I = moransi2(data, wlist)
print I

print percentile(ms1, 2.5), percentile(ms1, 50),\
      percentile(ms1, 97.5)
print percentile(ms2, 2.5), percentile(ms2, 50),\
      percentile(ms2, 97.5)

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.hist(ms1, 20, color="white",alpha=0.8,label="Независимый")
ax.hist(ms2, 20, color="grey",alpha=0.3,label="Перестановка")
plt.legend(loc='upper right')
plt.show()
```

Программа печатает следующий результат:

```
0.879154984566
-0.0983709874209 -0.00442892038325 0.0918736611396
-0.0998426464734 -0.00425357474198 0.0909170028323
```

Видно, что ядро с затуханием при увеличении дальности дает сильную положительную автокорреляцию и что для всех случайных паттернов ин-

декс Морана близок к –0.004 (что очень близко к теоретическому среднему $-1/(N-1) = -0.00446$). Формы двух гистограмм тоже похожи, хотя можно возразить, что в независимом случае концентрация распределения более выражена (рис. 9.8).

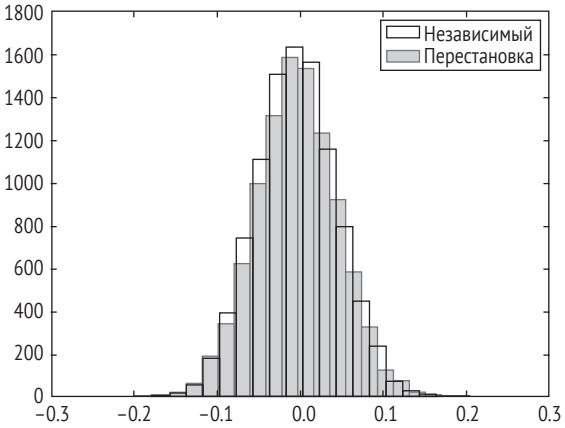

Рис. 9.8 ❖ Гистограмма значений индекса Морана на регулярной сетке, полученных методом Монте-Карло

9.3. Кластеризация

Следствием положительной пространственной автокорреляции в пространственных паттернах является появление кластеров, т. е. ситуации, когда похожие объекты располагаются близко друг к другу. Есть много способов найти, где находятся эти кластеры. Здесь мы опишем широко распространенный алгоритм *k* средних. Чтобы им воспользоваться, нужно заранее определить количество кластеров в данных – число *k*. Алгоритм сначала размещает в области *k* точек случайным образом, и эти точки становятся начальными центрами кластеров. Затем каждой точке данных сопоставляется ближайший к ней центр. По точкам, отнесенным к каждому центру, вычисляется новый центр, который может отличаться от начального. Если старый и новый центры сильно различаются, то старый центр заменяется новым, и отнесение точек к центру производится заново. В результате суммарное расстояние уменьшается. Процесс повторяется, пока изменение суммарного расстояния не станет достаточно малым. Алгоритм *k* средних на удивление простой и гибкий в том смысле, что исходные данные вообще не обязаны быть пространственными, а могут быть любыми и иметь любую размерность. Это один из самых употребительных алгоритмов машинного обучения и добычи данных.

Алгоритм *k* средних реализован в листинге 9.12. Функция `clustering_dist` относит точки к ближайшим центрам и возвращает суммарное расстояние. Функция `initk` возвращает начальное множество центров. Есть разные ме-

тоды инициализации, мы реализовали два из них. В первом – методе Форджи – из входных данных случайно выбирается *k* точек, которые становятся начальными центрами, а в рандомизированном методе *k* центров создаются с помощью генератора случайных чисел.

Листинг 9.12 ❖ Кластеризация методом *k* средних (kmeans.py)

```
import sys
sys.path.append('../geom')
from point import *
import random
from math import fabs
from math import sqrt

INF = float('inf')

def clustering_dist(points, means):
    n = len(points)
    k = len(means)
    nearests = [[] for i in range(k)]
    totaldist = 0
    for i in range(n):
        dmin = INF
        near = []
        for j in range(k):
            d = points[i].distance(means[j])
            if d < dmin:
                dmin = d
                jmin = j
        totaldist += dmin
        nearests[jmin].append(i)
    totaldist = totaldist/n
    return nearests, totaldist

def initk(points, k, init):
    n = len(points)
    xmin = INF
    ymin = INF
    xmax = -INF
    ymax = -INF
    for p in points:
        xmin = min([xmin, p.x])
        ymin = min([ymin, p.y])
        xmax = max([xmax, p.x])
        ymax = max([ymax, p.y])
    nearests = [[] for i in range(k)]
    while [] in nearests:             # пока в nearests не останется пустых множеств
        if init=="forgy":             # метод инициализации Форджи
            means = [points[i]
                     for i in random.sample(range(n), k)]
        elif init=="random":
            means = [ Point(random.uniform(xmin, xmax),
```

```
                            random.uniform(ymin, ymax))
                    for i in range(k) ]
        else:
            print "Ошибка: неизвестный метод инициализации"
            sys.exit(1)
        nearests, totaldist = clustering_dist(points, means)
    return means, nearests, totaldist

def kmeans(points, k, threshold=1e-5, init="forgy"):
    bigdiff = True
    means, nearests, totaldist = initk(points, k, init)
    while bigdiff:
        means2 = []
        for j in range(k):
            cluster = [xx for xx in nearests[j]]
            sumx = sum([points[ii].x for ii in cluster])
            sumy = sum([points[ii].y for ii in cluster])
            numpts = len(nearests[j])
            if numpts>0:
                sumx = sumx/numpts
                sumy = sumy/numpts
            means2.append(Point(sumx, sumy))
        nearests, newtotal = clustering_dist(points, means2)
        offset = totaldist - newtotal
        if offset > threshold:
            means = means2
            totaldist = newtotal
        else:
            bigdiff = False
    return totaldist, means

def test():
    n = 500
    points = [ Point(random.random(), random.random())
    for i in range(n) ]
        print kmeans(points, 10, init="forgy")[0]

    points1 = [ Point(random.uniform(10, 20),
                      random.uniform(10, 20))
              for i in range(n/2) ]
    points2 = [ Point(random.uniform(30, 40),
                      random.uniform(30, 40))
              for i in range(n/2) ]
    print kmeans(points1+points2, 2)[0]

    points1 = [ Point(random.uniform(10, 20),
                      random.uniform(10, 20))
              for i in range(n/3) ]
    points2 = [ Point(random.uniform(30, 40),
                      random.uniform(10, 20))
              for i in range(n/3) ]
    points3 = [ Point(random.uniform(20, 30),
```

```
 98                      random.uniform(30, 40))
 99              for i in range(n/3) ]
100   print kmeans(points1+points2+points3, 3)[0]
101   print kmeans(points1+points2+points3, 3,
102                init="random")[0]
103
104 if __name__ == "__main__":
105     test()
```

В коде алгоритма *k* средних тестируется три сценария на 500 точках. В первом случае точки распределены случайным образом. Во втором случае имеется соответственно два и три блока, в каждом из которых точки распределены случайно. Ниже приведены результаты выполнения программы во всех трех случаях. Мы видим, что последние две строки одинаковы, они соответствуют двум разным способам инициализации в третьем случае. На рис. 9.9 показано, как центры приближаются к окончательному положению, их последовательные позиции обозначены цифрами.

```
0.118536697855
3.77665952559
3.86220805887
3.86220805887
```

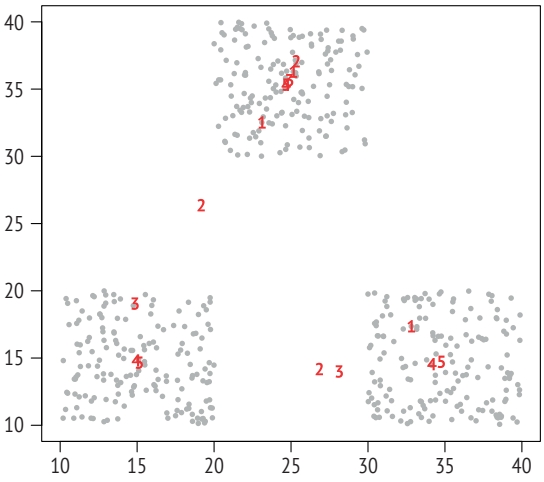

Рис. 9.9 ❖ Пример работы алгоритма *k* средних. Числа обозначают номер итерации, 1 соответствует начальному положению центров

9.4. Метрики ландшафтной экологии

Главная цель изучения любого пространственного паттерна – надежда выявить закономерность, которая поможет понять лежащий в основе явления процесс. Например, придя к выводу, что некоторые точки объединены

в кластеры, разумно задаться вопросом, почему это так. Это, в частности, один из самых важных аспектов ландшафтной экологии, которая изучает паттерны и, что более важно, способы измерения паттернов. Существует много способов оценить паттерны ландшафта, и, разумеется, мы не сможем рассмотреть их все. Мы ограничимся лишь одним, считая его хорошей отправной точкой. Но предварительно следует упомянуть, что большинство метрик ландшафтной экологии применяются к категориальным данным. В этом контексте чаще используются растровые данные, хотя существуют метрики, для вычисления которых нужны наборы векторных данных.

Индекс распространенности (contagion) – метрика ландшафтной экологии, измеряющая вероятность обнаружения разных типов в одном месте. Высокая распространенность означает групповой паттерн типов, когда один класс земель занимает значительную часть всей области, а низкая распространенность – рассеянный, фрагментированный паттерн.

Обозначим p_{ij} вероятность того, что два случайно выбранных соседних пикселя принадлежат типам i и j соответственно. Мы можем оценить p_{ij} как

$$p_{ij} = p_i N_{ij}/N_i,$$

где p_i – вероятность того, что случайно выбранный пиксель имеет тип i, N – общее число экземпляров типа i по одну сторону ребра, а N_{ij} – количество случаев соседства, когда типы i и j встречаются по обе стороны. Индекс распространенности ландшафта можно вычислить по формуле, напоминающей формулу энтропии:

$$C = 1 + \frac{\sum_i^n \sum_j^n p_{ij} \ln p_{ij}}{2 \ln n},$$

где n – количество типов в данных.

Реализация индекса распространенности на Python приведена в листинге 9.13. На вход подается двумерный массив целых чисел, потому что эта метрика применима только к категориальным данным. В начале программы мы должны выделить уникальные значения и сохранить их в списке (строка 16). Доля каждого типа (p_i) вычисляется в строке 27. Затем мы используем встроенную в Python структуру данных словаря (`types`), чтобы присвоить уникальным значениям последовательные целочисленные индексы, т. к. числовые значения данных необязательно соседние (строка 29). Далее все просто: мы проверяем каждое горизонтальное и вертикальное ребро между ячейками массива данных и подсчитываем количество ребер между парами разных типов. Мы не включаем в подсчет ребра, по обе стороны которых находятся данные одного типа, поскольку эта метрика измеряет участки, а не пиксели. Зная счетчики, мы можем вычислить значение p_{ij} (строка 61), а затем использовать его для вычисления индекса распространенности.

Листинг 9.13 ❖ Вычисление индекса распространенности (contagion.py)

```
1 import numpy as np
2 from sys import exit
3 from math import log
```

```
 4
 5 def contagion(data):
 6     """
 7     Вход
 8       data: двумерный массив NumPy, содержащий целые числа
 9
10     Выход
11       Индекс распространенности
12     """
13     nrows = len(data)
14     ncols = len(data[0])
15     # получить список уникальных значений
16     UList = []
17     Pi = []
18     for i in range(nrows):
19         for j in range(ncols):
20             if data[i, j] not in UList:
21                 UList.append(data[i, j])
22                 Pi.append(1)
23             else:
24                 itype = UList.index(data[i,j])
25                 Pi[itype] += 1
26
27     Pi = [float(i)/ncols/nrows for i in Pi]
28
29     types = dict()
30     for i in range(len(UList)):
31         types[UList[i]] = i
32
33     n = len(UList)
34     Nij=[[0]*n for x in range(n)]
35     Ni = [0 for x in range(n)]
36     for i in range(nrows):            # вертикальные ребра
37         for j in range(ncols-1):
38             i1 = types[data[i, j]]
39             j1 = types[data[i, j+1]]
40             Ni[i1] += 1               # тип i1 с одной стороны
41             Ni[j1] += 1               # тип j1 с другой стороны
42             if i1==j1:
43                 continue
44             Nij[i1][j1] += 1
45             Nij[j1][i1] += 1          # симметрично
46     for j in range(ncols):            # горизонтальные ребра
47         for i in range(nrows-1):
48             i1 = types[data[i,j]]
49             j1 = types[data[i+1, j]]
50             Ni[i1] += 1
51             Ni[j1] += 1
52             if i1==j1:
53                 continue
54             Nij[i1][j1] += 1
55             Nij[j1][i1] += 1          # симметрично
```

```
56
57      sum = 0.0
58      for i in range(n):
59          for j in range(n):
60              if Nij[i][j]:
61                  pij = float(Nij[i][j]) * Pi[i] / Ni[i]
62                  sum += pij * log(pij)
63
64      sum /= 2.0*log(n)
65      sum += 1
66      return sum
```

Сначала протестируем индекс распространенности на нескольких простых пространственных конфигурациях на сетке 4×4 (листинг 9.14). Примеры включают паттерн с четырьмя типами, в которых каждый тип соседствует только с отличными от него (`data1`), похожий паттерн всего с двумя типами, чередующимися в шахматном порядке (`data2`), паттерн, в котором имеется доминирующий тип, но при этом все же фрагментированный (`data3`), паттерн с четырьмя типами, сгруппированными в четыре равных блока (`data4`), и паттерн, в котором доминирует один большой участок (`data5`).

Листинг 9.14 ❖ Тестирование распространенности, часть 1 (test_contagion.py)

```
 1 import random
 2 from contagion import *
 3 import numpy as np
 4
 5 data1 = np.array([[ 1, 2, 3, 4],
 6                   [ 4, 3, 2, 1],
 7                   [ 1, 2, 3, 4],
 8                   [ 4, 3, 2, 1]])
 9
10 data2 = np.array([[ 1, 0, 1, 0],
11                   [ 0, 1, 0, 1],
12                   [ 1, 0, 1, 0],
13                   [ 0, 1, 0, 1]])
14
15 data3 = np.array([[ 0, 1, 1, 1],
16                   [ 1, 1, 1, 1],
17                   [ 0, 1, 1, 1],
18                   [ 1, 1, 0, 0]])
19
20 data4 = np.array([[1, 1, 2, 2],
21                   [1, 1, 2, 2],
22                   [3, 3, 4, 4],
23                   [3, 3, 4, 4]])
24
25 data5 = np.array([[ 1, 1, 1, 1],
26                   [ 1, 1, 1, 1],
27                   [ 1, 1, 1, 1],
28                   [ 1, 1, 0, 0]])
29 print "data1:"
```

```
30 print data1, contagion(data1)
31 print "data2:"
32 print data2, contagion(data2)
33 print "data3:"
34 print data3, contagion(data3)
35 print "data4:"
36 print data4, contagion(data4)
37 print "data5:"
38 print data5, contagion(data5)
```

Результаты для всех пяти случаев приведены ниже. Они согласуются с идеей индекса распространенности: наглядно виден компромисс между фрагментацией (`data1`), из-за которой распространенность уменьшается, и большим однородным участком (`data5`), в котором распространенность увеличивается.

```
data1:
[[1 2 3 4]
 [4 3 2 1]
 [1 2 3 4]
 [4 3 2 1]] 0.270741104622
data2:
[[1 0 1 0]
 [0 1 0 1]
 [1 0 1 0]
 [0 1 0 1]] 0.5
data3:
[[0 1 1 1]
 [1 1 1 1]
 [0 1 1 1]
 [1 1 0 0]] 0.557573110559
data4:
[[1 1 2 2]
 [1 1 2 2]
 [3 3 4 4]
 [3 3 4 4]] 0.617919791607
data5:
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 0 0]] 0.736734583866
```

Поставленный выше эксперимент дает поверхностное представление о факторах, влияющих на распространенность, но давайте предпримем расширенное исследование. В листинге 9.15 используется идея ядерной функции, которая создает участок (патч) во внутренней области сетки. Мы реализовали четыре ядерные функции: бивесовую, тривесовую, гауссову и однородную (строки 29, 31, 33 и 35 соответственно). Тестируется бивесовое ядро, подразумеваемое по умолчанию. Всем ячейкам со значением больше 0.3 будет назначен тип 3 (строка 37), а остальным пикселям – случайный тип от 0 до 2 (строка 39). Размер внутреннего патча определяется шириной полосы, мы тестируем значения ширины в диапазоне от 1.1 до 14.5 с шагом

0.5 (строка 45). График зависимости распространенности от ширины полосы показан на рис. 9.10, из которого видно, что индекс распространенности увеличивается с ростом ширины полосы, определяющей размер патча.

Листинг 9.15 ❖ Тестирование распространенности, часть 2 (test_contagion_kernel.py)

```
 1 import random
 2 from contagion import *
 3 from math import sqrt, pi, exp
 4 import numpy as np
 5
 6 def kernel(nrows, ncols, r, func="biweight"):
 7     """
 8     Возвращает сетку с ядром в центре.
 9     Вход
10       nrows: число строк
11       ncols: число столбцов
12       r: ширина полосы ядра
13       func: функция ядра (двувесовое, тривесовое,
14             гауссово или равномерное)
15     Выход
16       data: двумерный список (списков)
17     """
18     data = np.array([ [0]*ncols for i in range(nrows)])
19     i0 = nrows/2
20     j0 = ncols/2
21     for i in range(nrows):
22         for j in range(ncols):
23             d = sqrt((i-i0)*(i-i0) + (j-j0)*(j-j0))
24             if d>r:
25                 val = 0
26             else:
27                 u = float(d)/r
28                 if func=="biweight":
29                     val = 15.0/16.0 * (1 - u**2)**2
30                 elif func=="triweight":
31                     val = 35.0/32.0 * (1 - u**2)**3
32                 elif func=="gaussian":
33                     val = 1.0/sqrt(2*pi) * exp(-0.5*u*u)
34                 else:
35                     val = 0.5
36             if val > 0.3:
37                 val = 3
38             else:
39                 val = random.randint(0, 2)
40             data[i][j] = val
41     return data
42
43 nrows = 15
44 ncols = 15
45 rs = np.arange(1.1, ncols, 0.5)
```

```
46 for r in rs:
47     data = kernel(nrows, ncols, r, func="biweight")
48     print r, contagion(data)
```

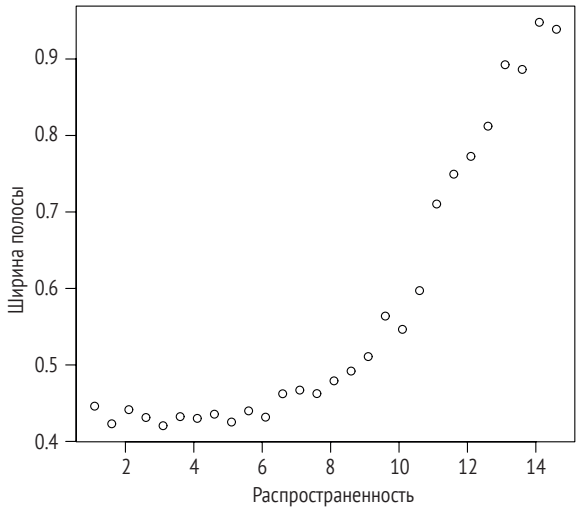

Рис. 9.10 ❖ Индекс распространенности

9.5. Примечания

Литература по количественной географии изобилует обсуждениями анализа пространственных паттернов (Cliff and Ord, 1981; Fotheringham et al., 2000). В книге Boots and Getis (1988) имеется хороший обзор многих методов анализа точечных паттернов. В книге Bailey and Gatrell (1995) можно найти углубленное изучение дополнительных методов. Статистика расстояния до ближайшего соседа была предложена в работе Clark and Evans (1954), а *K*-функция – в работе Ripley (1981).

Хотя индекс Морана (Moran, 1950) давно и широко используется, существуют статистики других типов, например индекс *C* (Geary, 1954), который может дать информацию о пространственной автокорреляции. Упомянем также локальные статистики, новое важное направление в измерении пространственной автокорреляции. Например, локальная статистика *I* Морана (Anselin, 1995) и статистики *G* и G^* (Getis and Ord, 1992) обобщают идею на каждую точку, так что становится возможно исследовать изменение пространственной автокорреляции в пространстве. Здесь мы не рассматривали эти новые работы. Но изучение таких вопросов, как пространственная смежность и индекс *I* Морана, должно вооружить читателя для познавания новых направлений. Индекс распространенности впервые был предложен в работе O'Neill et al. (1988). Однако в оригинальной формуле результат не был нормирован, исправленный вариант дает более надежные данные для интерпретации (Riitters et al., 1996).

Хотя в этой главе (и вообще в книге) мы пользуемся языком Python для демонстрации анализа пространственных паттернов, существует немало программных пакетов, предназначенных для этой цели. Пакет FRAGSTATS[1] (McGarigal and Marks, 1995) включает полный комплект инструментов для обсчета ландшафтных паттернов с применением многочисленных метрик. В библиотеке на чистом Python PySAL[2] имеются инструменты для расширенного (и базового) пространственного анализа. Есть много библиотек, написанных на языке R, из них отметим sdep[3]. В приложении В.3 приведено очень краткое руководство по применению PySAL для вычисления индекса Морана.

9.6. Упражнения

1. Преимущество статистики расстояния до ближайшего соседа заключается в том, что ее сравнительно легко объяснить, т. к. приходится иметь дело всего с одним значением. Напротив, *K*-функция сильно изменяется при вычислении ее из разных точек в исследуемой области. Зато у *K*-функции есть полезное свойство – она дает ощущение масштаба: мы можем сказать, как изменяется паттерн (кластерный или рассеянный) при рассмотрении картины в разных масштабах (при разных радиусах). Что касается расстояния до ближайшего соседа, то возникает вопрос: почему именно до *ближайшего*? Почему не до второго по близости? Наш инструмент индексирования позволяет легко изучить этот вопрос. Напишите программу, которая возвращает метрику расстояния как функцию от радиуса, и объясните, как можно ей воспользоваться для лучшего понимания паттерна пространственной величины. Например, можно использовать среднее расстояние от точки до других точек внутри окружности заданного радиуса.
2. Мы использовали *k*D-дерево, чтобы ускорить вычисление *K*-функции. Исследуйте, что будет, если не пользоваться никаким методом индексирования.
3. Сравните разные методы индексирования и посмотрите, как они проявляют себя в конкретной задаче вычисления *K*-функции для разных наборов данных.
4. Вычисление матрицы смежности для векторного набора данных часто оказывается трудоемкой задачей. Но для растровых данных, когда пространственными единицами являются квадраты, такой необходимости не возникает, потому что эти единицы организованы регулярно, и смежность двух единиц (ячеек) легко определить, зная, что это за ячейки. Напишите на Python функцию `create_wlist`, которая это делает, а затем включите ее в код вычисления индекса *I* Морана и проверьте, как она работает. Функция должна принимать два параметра: число строк и число столбцов,

[1] http://www.umass.edu/landeco/research/fragstats/fragstats.html.

[2] http://pysal.org.

[3] http://cran.r-project.org/web/packages/spdep/index.html.

именно в таком порядке. Она должна возвращать список, каждый элемент которого является списком двух целых чисел, содержащих индексы двух соседних ячеек.

5. Индекс *I* Морана – не единственная статистика для изучения пространственной автокорреляции. Еще одна статистика, индекс *C* Джири, определяется формулой

$$C = \frac{(n-1)\sum_i \sum_j w_{ij}(z_i - z_j)^2}{2\sum_i \sum_j w_{ij}(z_i - \bar{z})^2}.$$

   Напишите на Python программу, которая вычисляет индекс Джири для произвольных пространственных данных. Сравните индексы Джири и Морана для одного и того же набора данных.
6. Матрица весов, использованная нами при вычислении индекса Морана, основана на соседстве многоугольников. Это решение может показаться произвольным, потому что мы, безусловно, можем придумать и другие способы определить, являются ли две области (в нашем случае – многоугольники) достаточно близкими, чтобы учитываться при вычислении. Например, можно использовать внутреннюю точку многоугольника (скажем, центроид) и считать, что пара многоугольников имеет вес 1, если расстояние между их внутренними точками не превышает некоторый порог. Можно также заставить программу рассматривать не более *k* близких областей. В этой книге мы используем бинарный вес. Но вес может быть и непрерывным, скажем обратно пропорциональным расстоянию или квадрату расстояния. Реализуйте некоторые из этих идей и сравните новые индексы с индексом *I* Морана.
7. Напишите на Python программу, которая вычисляет обе дисперсии индекса Морана, и сравните критерии для проверки гипотез, используя эти дисперсии и результаты моделирования методом Монте-Карло.

Глава 10

Анализ сетей

В поход, беспечный пешеход,
Уйду, избыв печаль, –
Спешит дорога от ворот
В заманчивую даль,
Свивая тысячи путей
В один, бурливый, как река,
Хотя, куда мне плыть по ней,
Не знаю я пока!

Дж. Р. Р. Толкиен
«Братство Кольца»

Сети повсюду. Например, для перевозки людей и грузов необходимо наличие дорожной сети. Компьютерные сети соединяют между собой различные вычислительные устройства. Чтобы от социальных сетей была польза, необходимо сообщество – сеть – общающихся между собой людей. Реальные сети обладают различными характеристиками, но чтобы работать с ними, необходимо знать о двух фундаментальных понятиях: узлах и ребрах. Узлы представляют функциональные единицы в сети, они играют роль порождающей и принимающей стороны при распространении информации, услуг и товаров. Ребра представляют среду, обеспечивающую перемещение информации, услуг и товаров от одного узла к другому. В зависимости от приложения узлам и ребрам могут назначаться веса. Например, в транспортной сети узлу можно назначить вес, отражающий общий объем подлежащих перемещению товаров, а ребру – вес, отражающий его пропускную способность или длину. В социальных сетях узел может представлять человека (и, быть может, все, что сопровождает человека в киберпространстве), а ребро – ту или иную форму связи между людьми (например, разного вида дружеские отношения). Сети можно абстрагировать в виде графов, когда нас больше интересует не то, как взвешены узлы и ребра, а что мы можем с ними делать.

Как и другие пространственные типы данных, сети нуждаются в хорошем способе представления, позволяющем эффективно обрабатывать их с помощью компьютера. Возьмем пример простой сети на рис. 10.1. Мы видим восемь узлов, помеченных курсивными цифрами, и ребра между некоторыми узлами. Число рядом с каждым ребром показывает какую-то меру стоимости (вес); это может быть, например, расстояние или время, необходимое для прохода из одного узла в другой по этому ребру. Есть два общих подхода к кодированию такой сети. Первый – с помощью списка, каждый эле-

мент которого содержит узел и список всех узлов, соседних с данным. В эту структуру данных можно также включить дополнительную информацию, например веса ребер. Список можно сохранить в простом текстовом файле и загружать в программу для обработки. Такая структура проста, но не дает информации о связях между несмежными узлами. Вторая структура данных может решить эту проблему с помощью матрицы, каждый элемент которой содержит полный вес кратчайшего пути между узлами, соответствующими строке и столбцу этого элемента. Матричное представление по существу эквивалентно списку, но мы должны знать кратчайшие пути между всеми парами узлов, а чаще всего эта информация в данных о сети напрямую не присутствует.

Обычно мы описываем сеть проще, перечисляя все пары смежных узлов (листинг 10.1). Это представление сети можно преобразовать в список. Или подвергнуть обработке и построить матрицу сети. Основная цель данной главы в том и состоит, чтобы рассказать об алгоритмах поиска кратчайших путей между узлами сети. Анализ сети может представать в самых разных обличьях, но эти алгоритмы предназначены для решения наиболее фундаментальных задач и являются хорошей отправной точкой для дальнейших построений. Для тестирования всех алгоритмов мы будем использовать сеть на рис. 10.1. Отметим, что в этой книге обсуждаются только неориентированные сети.

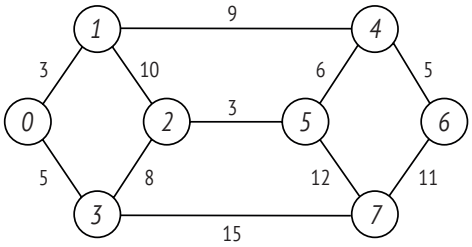

Рис. 10.1 ❖ Пример сети. Узлы помечены курсивными цифрами, а рядом с ребрами проставлены их длины (без учета масштаба)

Листинг 10.1 ❖ Список, содержащий информацию обо всех ребрах простой сети на рис. 10.1 (network-links.dat)

```
8
0 1 3
0 3 5
1 2 10
1 4 9
2 3 8
2 5 3
3 7 15
4 5 6
4 6 5
5 7 12
6 7 11
```

В листинг 10.2 включено две функции. Функция `network2list` читает данные о сети из файла (см. листинг 10.1) в список списков, чтобы получить описанное выше списковое представление сети. Функция `network2distancematrix` преобразует файл в матричную форму. В матрице мы запоминаем только расстояние, или вес для прямых связей (т. е. ребер), а для всех остальных пар узлов записываем значение `INF` (бесконечность). Далее в этой главе при обсуждении алгоритма нахождения кратчайших расстояний между всеми парами узлов мы восполним недостающее, заменив значения `INF` в матрице. Эти две функции, особенно `network2distancematrix`, будут часто использоваться в этой и двух последующих главах.

Листинг 10.2 ❖ Чтение файла сети с построением списка или матрицы (network2listmatrix.py)

```
INF = float('inf')

def network2list(fname, is_zero_based = True):
    """
    Преобразовать файл сети в списковую структуру данных.
    Вход
      fname: имя файла сети, записанного в формате:
             n
             i j расстояние
             ...
    Выход
      network: список списков, в котором i-й элемент содержит список
               всех узлов, смежных с узлом i
    """
    f = open(fname)
    l = f.readline()
    n = int(l.strip().split()[0])
    network = [[] for i in range(n)]
    for l in f:
        nodesnedge = l.strip().split()
        if len(nodesnedge)==3:
            i=int(nodesnedge[0])
            j=int(nodesnedge[1])
            if not is_zero_based:
                i = i-1
                j = j-1
            network[i].append(j)
            network[j].append(i)
    f.close()
    return network

def network2distancematrix(fname, is_zero_based = True):
    """
    Читает список из входного файла и возвращает матрицу
    смежности. Входной файл содержит n + 1 строк и должен быть
    записан в следующем формате:

```

```
    n
    i j расстояние
    ...

    где первая строка содержит число ребер, а каждая
    из последующих – индексы вершин ребра и расстояние,
    ассоциированное с этим ребром. Нумерация индексов
    начинается с 0 (по умолчанию) или с 1.
    """
    a = []
    f = open(fname)
    l = f.readline()
    n = int(l.strip().split()[0])    # число ребер
    a=[[INF]*n for x in xrange(n)]   # инициализировать 2-мерный список INF
    for l in f:
        nodesnedge = l.strip().split()
        if len(nodesnedge)==3:
            i=int(nodesnedge[0])
            j=int(nodesnedge[1])
            if not is_zero_based:
                i = i-1
                j = j-1
            d=int(nodesnedge[2])
            a[i][j] = d
            a[j][i] = d
    for i in range(n):
        a[i][i] = 0
    return a
```

10.1. Обход сети

Как наиболее эффективно обойти все узлы сети, начав с одного из них и следуя по ребрам? Эта задача обхода сети отличается от задачи перечисления узлов, не связанных ребрами. Здесь мы хотим, чтобы узлы посещались только вследствие наличия связей с другими узлами. Существует два основных метода обхода: в ширину и в глубину.

10.1.1. Обход в ширину

Находясь в начальном узле, мы можем посетить все остальные узлы сети следующим образом: сначала посетить узлы, смежные с данным, затем для каждого смежного узла посетить смежные с ним и повторять этот процесс, пока не останется непосещенных узлов. По ходу дела мы вставляем каждый посещенный или подлежащий посещению узел в некую структуру данных, устроенную таким образом, что узел, который был помещен в нее первым, первым и будет извлечен. Такая структура данных называется очередью, обслуживаемой по правилу «первым пришел – первым ушел» (first-in-first-out –

FIFO). Поскольку подобный порядок обхода гарантирует, что сначала посещаются узлы, смежные с текущим, он получил название «обход в ширину».

В листинге 10.3 показана реализация поиска в ширину. В качестве очереди мы просто используем список Python, при этом новые элементы всегда добавляем в конец списка (строка 14), а извлекаем элементы из его начала (строка 20). В этом алгоритме очередь содержит узлы, которые предстоит посетить, и мы динамически обслуживаем ее, так что на каждой итерации извлекается первый находящийся в очереди узел (строка 20). Поддерживаются также две дополнительные структуры. В списке `V` хранятся уже посещенные узлы (строка 21), а в списке `labeled` узел помечается признаком `True`, если он уже был добавлен в очередь (строка 16). Мы находим все узлы, смежные с только что извлеченным из очереди, и если смежный узел еще не рассматривался (строка 23), то помечаем его (строка 24) и помещаем в очередь (строка 25).

Листинг 10.3 ❖ Поиск в сети в ширину (bfs.py)

```
 1 from network2listmatrix import network2list
 2
 3 def bfs(network, v):
 4     """
 5     Поиск в ширину
 6     Вход
 7       network: сеть, представленная списком списков
 8       v: узел, с которого начинается поиск
 9     Выход
10       V: список посещенных узлов
11     """
12     n = len(network)
13     Q = []
14     Q.append(v)
15     V = []
16     labeled = [ False for i in range(n)]
17     labeled[v] = True
18     while len(Q) > 0:
19         # печатать list(Q)
20         t = Q.pop(0)
21         V.append(t)
22         for u in network[t]:
23             if not labeled[u]:
24                 labeled[u] = True
25                 Q.append(u)
26     return V
27
28 if __name__ == "__main__":
29     network = network2list('../data/network-links')
30     V = bfs(network, 3)
31     print "Посещенные:", V
```

Протестируем алгоритм поиска в ширину на нашей простой сети, выбрав начальным узел 3. Ниже напечатано содержимое очереди на каждой итерации и конечная последовательность посещенных узлов.

```
[3]
[0, 2, 7]
[2, 7, 1]
[7, 1, 5]
[1, 5, 6]
[5, 6, 4]
[6, 4]
[4]
Посещенные: [3, 0, 2, 7, 1, 5, 6, 4]
```

Заметим, что содержимое очереди постоянно изменяется, поскольку на каждой итерации в нее добавляются узлы, смежные с только что извлеченным. Для формирования конечной последовательности узлов на каждой итерации в список добавляется узел, оказавшийся первым в очереди.

10.1.2. Обход в глубину

При обходе в глубину смежные узлы обрабатываются иначе, чем при обходе в ширину: мы заходим в один из смежных узлов, затем в какой-то из смежных с ним и так далее, пока есть возможность продолжать. Для этого нам понадобится другая структура данных, которая называется стеком и дает доступ только к последнему помещенному в нее узлу. Такая дисциплина обслуживания называется «последним пришел – первым ушел» (last-in-first-out – LIFO). Алгоритм приведен в листинге 10.4, где мы используем список для реализации стека (строка 13). В остальном структура алгоритма такая же, как в случае поиска в ширину.

Листинг 10.4 ❖ Поиск в сети в глубину (dfs.py)

```
 1 from network2listmatrix import network2list
 2
 3 def dfs(network, v):
 4     """
 5     Поиск в глубину
 6     Вход
 7       network: сеть, представленная списком списков
 8       v: узел, с которого начинается поиск
 9     Выход
10       V: список посещенных узлов
11     """
12     n = len(network)
13     S = []          # пустой стек
14     S.append(v)
15     V = []
16     labelled = [ False for i in range(n)]
17     labelled[v] = True
18     while len(S) > 0:
19         print S
20         t = S.pop()
21         V.append(t)
```

```
22          for u in network[t]:
23              if not labelled[u]:
24                  labelled[u] = True
25                  S.append(u)
26      return V
27
28 if __name__ == "__main__":
29      network = network2list('../data/network-links')
30      V = dfs(network, 3)
31      print "Посещенные:", V
```

Для тестирования алгоритма мы снова берем нашу простую сеть и узел 3 в качестве начального:

```
[3]
[0, 2, 7]
[0, 2, 5, 6]
[0, 2, 5, 4]
[0, 2, 5, 1]
[0, 2, 5]
[0, 2]
[0]
Посещенные: [3, 7, 6, 4, 1, 5, 2, 0]
```

Результаты наглядно демонстрируют различия двух алгоритмов обхода. Здесь мы видим, что после того как на начальном шаге вставлен узел, первый элемент стека (0) не меняется до последней итерации, на которой стек опустошается. Это объясняется тем, что алгоритм стремится сначала дойти до самых дальних узлов, а затем возвращается к тем, которые были выбраны в качестве соседних на более ранних этапах. В конечной последовательности посещенных узлов оказываются узлы, которые находились в конце списка, представляющего стек, на каждой итерации.

10.2. Кратчайший путь из одного узла

Теперь обратимся к алгоритмам, которые можно отнести к числу самых полезных для анализа сетей, – алгоритмам поиска кратчайшего пути. Есть много алгоритмов, возвращающих кратчайшие пути в сети. Некоторые спроектированы для решения задачи в случае, когда задан один исходный узел и требуется найти кратчайший путь из него в конечный узел. Есть общие алгоритмы, есть алгоритмы, которые умеют работать с ребрами, имеющими отрицательные веса. Есть также алгоритмы, которые ищут кратчайшие пути сразу между всеми парами вершин. В этом разделе мы опишем алгоритм Дейкстры для поиска кратчайших путей из одной вершины, а в следующем обсудим алгоритм для всех пар вершин.

На рис. 10.2 показаны шаги выполнения алгоритма Дейкстры в нашей простой сети, когда начальным выбран узел 1. Этому алгоритму не нужно задавать конечный узел, потому что результатом его работы являются

кратчайшие пути (и их суммарные веса или длины) из начального узла во все остальные. На каждом шаге алгоритма мы должны сопоставить каждому узлу два значения: полное расстояние до начального узла по кратчайшему пути и непосредственно предшествующий узел. Мы показываем эти значения двумя числами, разделенными знаком косой черты. В начале работы мы знаем только начальный узел и помечаем его серым цветом. В этот момент интересующие нас значения известны только для начального узла; они обозначены 0/–, т. е. суммарный вес равен 0 и непосредственно предшествующего узла еще нет. Узлам, смежным с начальным, назначается вес с учетом весов ребер, соединяющих их с начальным узлом, и того факта, что предшествующим узлом является начальный. На этой стадии всем остальным узлам назначаются значения ∞/–, означающие, что полное расстояние еще не вычислено, а предыдущий узел на пути к данному еще неизвестен (рис. 10.2A).

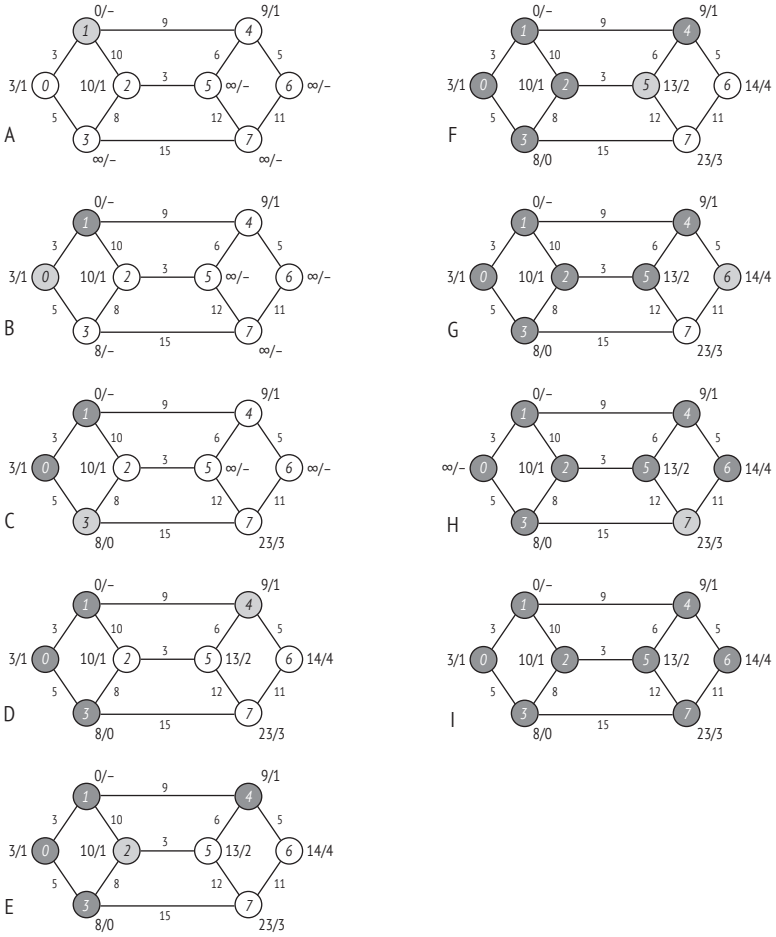

Рис. 10.2 ❖ Алгоритм Дейкстры нахождения кратчайшего пути

С узлом 1 мы на этом покончили и закрасили его черным цветом. Теперь рассмотрим смежные узлы. В нашем случае это узлы 0, 2 и 4. Следующим нас будет интересовать узел с наименьшим суммарным весом на данном шаге. При этом мы исключаем черные узлы, которые заведомо не приведут к дальнейшим изменениям, потому что мы точно знаем их полные веса и предшествующие им узлы. Таким образом, следующим будет узел 0, имеющий наименьший суммарный вес среди всех узлов (кроме узла 1) – 3. Перекрашиваем этот узел в серый цвет. Теперь обновим значения всех узлов, смежных с узлом 0. Обновление производится только в том случае, когда сумма текущего веса узла 0 и веса ребра, соединяющего 0 со смежным узлом, меньше текущего веса этого смежного узла. Если это так, то смежному узлу назначается новый суммарный вес и предыдущий узел – именно тот, с которым мы сейчас работаем (т. е. узел 0). На рис. 10.2B показан результат этого шага; как видим, суммарный вес узла 3 стал равен 8. Должно быть понятно, что узлы, уже покрашенные серым, не обновляются.

Итак, с узлом 0 покончено, мы можем закрасить его черным цветом и перейти к поиску следующего узла – того, у которого текущий суммарный вес минимален. Покрасим его в серый цвет и обновим смежные с ним узлы (рис. 10.2C). Данный процесс повторяется, пока все узлы не будут покрашены в черный цвет (рис. 10.2I). В этот момент для получения кратчайшего пути из начального узла в любой другой нужно пройти из конечного узла обратно, выбирая на каждом шаге предшествующий узел, запомненный в данном узле ранее. Кроме того, конечному узлу уже назначен суммарный вес, равный весу кратчайшего пути из начального узла в данный. Например, кратчайший путь из узла 1 в узел 6 – это 1-4-6, а его вес равен 14.

Алгоритм Дейкстры реализован в листинге 10.5. Мы поддерживаем три списка: полное расстояние до каждого узла (`dist`), предшествующий узел для каждого узла (`prev`) и узлы-кандидаты, еще не покрашенные в черный цвет (`Q`). В начале работы расстояния до всех узлов, кроме начального, инициализируются значением `INF` (строка 17). Затем начинается цикл, который повторяется, пока список узлов-кандидатов не окажется пуст (строка 20). На каждой итерации мы находим узел-кандидат с наименьшим расстоянием и удаляем его из списка `Q` (строка 21). Затем ищутся все соседи этого узла (строка 22). Для каждого смежного узла проверяется, выполнено ли условие обновления (строка 25), и если да, то обновление производится. Для нахождения узла-кандидата и удаления его из `Q` служит функция `get_remove_min`, а для получения соседей узла – функция `get_neighbor`. Наконец, функция `shortest_path` пользуется результатом работы алгоритма Дейкстры и строит кратчайший путь из начального узла в заданный.

Листинг 10.5 ❖ Алгоритм Дейкстры поиска кратчайшего пути (dijkstra.py)

```
1 from network2listmatrix import *
2
3 def dijkstra(source, distmatrix):
4     """
5     Алгоритм Дейкстры поиска кратчайших путей из одного узла.
6     Вход
```

```
      source: индекс начального узла
      distmatrix: матрица расстояний, элемент (i, j) равен INF,
                  если узлы i и j не соединены ребром, иначе
                  элемент равен весу (длине) этого ребра
    Выход
      dist: полные расстояния от начального узла до всех остальных
      prev: список, в котором для каждого узла указан предшествующий ему
            на кратчайшем пути
    """
    n = len(distmatrix)
    dist = [INF if i!=source else 0 for i in range(n)]
    prev = [None for i in range(n)]
    Q = range(n)
    while len(Q)>0:
        u = get_remove_min(Q, dist)
        U = get_neighbor(u, distmatrix, n)
        for v in U:
            newd = dist[u] + distmatrix[u][v]
            if newd < dist[v]:
                dist[v] = newd
                prev[v] = u
    return dist, prev

def get_remove_min(Q, dist):
    """
    Находит в Q узел с наименьшим расстоянием в dist и удаляет
    этот узел из Q.
    Вход
      Q: список узлов-кандидатов
      dist: список расстояний от каждого узла до начального
    Выход
      imin: индекс узла с наименьшим расстоянием
    """
    dmin = INF
    imin = -1
    for i in Q:
        if dist[i] < dmin:
            dmin = dist[i]
            imin = i
    Q.remove(imin)
    return imin

def get_neighbor(u, d, n):
    neighbors = [i for i in range(n)
                 if d[i][u]!=INF and i!=u]
    return neighbors

def shortest_path(source, destination, distmatrix):
    dist, prev = dijkstra(source, distmatrix)
    last = prev[destination]
    path = [destination]
    while last is not None:
```

```
59          path.append(last)
60          last = prev[last]
61      return path, dist[destination]
62
63 if __name__ == "__main__":
64      fname = '../data/network-links'
65      a = network2distancematrix(fname, True)
66      print shortest_path(1, 6, a)
67      print shortest_path(0, 7, a)
```

Здесь функция `network2distancematrix` загружает данные файла описания сети в матрицу, которая затем используется в функциях `shortest_path` и `dijkstra`. Выполнение тестовой части программы дает следующий результат:

```
([6, 4, 1], 14)
([7, 3, 0], 20)
```

В том, что эти пути между узлами 6 и 1 и узлами 7 и 0 действительно кратчайшие, можно убедиться, взглянув на рис. 10.2. Распечатку пути следует читать от конца к началу, чтобы получить кратчайший путь, идущий из начального узла.

Алгоритмы нахождения кратчайшего пути, похожие на алгоритм Дейкстры, без сомнения, весьма полезны и используются повсеместно. Например, службы оперативного построения маршрутов часто применяются при вождении автомобиля и решении других задач навигации. Далее мы познакомимся с одним специальным применением алгоритма кратчайшего пути в контексте задачи о калитке. Требуется найти кратчайший путь между двумя узлами с условием, что он должен (не важно, по какой причине) проходить через третий узел, называемый калиткой. Например, при планировке ландшафта мы часто хотим, чтобы маршрут проходил через живописное место. Задачу можно решить, выполнив алгоритм Дейкстры дважды, по разу для каждой концевой точки. Эта идея реализована в листинге 10.6.

Листинг 10.6 ❖ Применение алгоритма Дейкстры для решения задачи о калитке (gateway.py)

```
 1 from network2listmatrix import *
 2 from dijkstra import shortest_path
 3
 4 def gateway(s1, s2, gatewaynode, distmatrix):
 5     """
 6     Находит кратчайший путь, проходящий через калитку.
 7     Вход
 8       s1: первый конец пути
 9       s2: второй конец пути
10       gatewaynode: узел-калитка
11     Выход
12       Список узлов на пути и полный вес пути
13     """
14     path1, d1 = shortest_path(s1, gatewaynode, distmatrix)
```

```
    path2, d2 = shortest_path(s2, gatewaynode, distmatrix)
    path1.reverse()
    return path1[:-1]+path2, d1+d2

    if __name__ == "__main__":
        fname = '../data/network-links'
        a = network2distancematrix(fname, True)
        print gateway(0, 7, 2, a)
```

10.3. Кратчайшие пути между всеми парами узлов

Алгоритм Дейкстры эффективно решает задачу нахождения кратчайших путей из одного узла, но если требуется найти кратчайшие пути между всеми парами узлов сети, то его придется запускать для каждого узла в качестве начального. Это неэффективно, поэтому нам нужен другой алгоритм для этой цели. Ниже описывается алгоритм Флойда–Уоршелла.

Это итеративный процесс. Будем обозначать k номер итерации, считая, что в начальный момент $k = 0$. Мы храним матрицу расстояний D_k на k-й итерации, и элементы матрицы D_k будем обозначать d_{ij}^k. При $k = 0$ полагаем d_{ij}^k равным бесконечности, если узлы i и j не соединены ребром. Число итераций алгоритма n равно числу узлов. На каждой итерации k мы обновляем матрицу по правилу

$$d_{ij}^k = \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\},$$

где операция min возвращает меньшее из двух значений. После n итераций в каждой ячейке $d_{ij}^n \in D_n$ хранится полный вес кратчайшего пути между узлами i и j.

Реализация алгоритма Флойда–Уоршелла прямолинейна и сводится к модификации матрицы расстояний при переходе к следующей итерации. Код приведен в листинге 10.7. Хранить промежуточные матрицы нет необходимости, на протяжении всего процесса используется только одна матрица.

Листинг 10.7 ❖ Алгоритм Флойда–Уоршелла (allpairdist.py)

```
from network2listmatrix import network2distancematrix

def allpairs(a):
    """
    Возвращает матрицу весов (расстояний) для нахождения кратчайших
    путей между всеми парами узлов по алгоритму Флойда-Уоршелла.
    Вход
      a: начальная матрица расстояний, в которой расстояния между
         всеми несмежными парами узлов равны бесконечности.
    Выход
      Функция модифицирует переданную на вход матрицу.
```

```
12      """
13      n = len(a)
14      for k in range(n):
15          for i in range(n):
16              for j in range(n):
17                  if a[i][j] > a[i][k]+a[k][j]:
18                      a[i][j] = a[i][k]+a[k][j]
19
20  if __name__ == "__main__":
21      fname = '../data/network-links'
22      a = network2distancematrix(fname, True)
23      allpairs(a)
24      print a[1][6]
25      print a[0][7]
```

Программа печатает те же самые расстояния между узлами 1 и 6 и узлами 0 и 7, что и алгоритм Дейкстры ранее (т. е. 14 и 20). В табл. 10.1 показано, как изменяется матрица расстояний на каждой итерации, где $k = 0$ соответствует начальной матрице.

Очевидная проблема этого алгоритма – отсутствие информации о пути: нам известен полный вес каждого кратчайшего пути, но самого пути мы не знаем. Однако пути легко восстановить. Напомним, что правило обновления матрицы расстояний имеет вид $d_{ij}^k = d_{ik}^{k-1} + d_{ko}^{k-1}$. Это означает, что из i в j можно попасть по более короткому пути, проходящему через k. Такое возможно, только если k лежит на кратчайшем пути, и нам нужно только завести структуру данных, отражающую этот факт, и обновлять ее всякий раз, как обновляется расстояние между i и j. Добавим еще одну матрицу для хранения пути. В этой новой матрице можно хранить предшествующие узлы на пути (как в алгоритме Дейкстры) либо следующие узлы. В листинге 10.8 приведен вариант с хранением следующих узлов (метод с хранением предшествующих узлов см. в упражнениях). Точнее, матрица $\{b_{ij}\}$ устроена так, что узел, следующий за i на кратчайшем пути из i в j, хранится в элементе b_{ij}. В нашей реализации эта матрица называется `next`. Она инициализируется так, что $b_{ij} = i$, если узлы i и j соединены ребром, в противном случае $b_{ij} = -1$ (строка 20). Получив заполненную матрицу `next`, функция `recoverpath` восстанавливает путь из `s` в `t`, в цикле заменяя начальный узел `s` узлом, на который указывает `next[s][t]` (строка 30), пока не окажется, что новый `s` равен `t` (строка 29).

Листинг 10.8 ❖ Алгоритм Флойда–Уоршелла с восстановлением пути (allpairpaths.py)

```
1  from network2listmatrix import network2distancematrix, INF
2
3  def allpairswpaths(a, next):
4      """
5      Возвращает матрицу весов (расстояний) для нахождения весов
6      кратчайших путей между всеми парами узлов и самих путей
7      по алгоритму Флойда-Уоршелла.
8      Вход
9        a: начальная матрица расстояний, в которой расстояния между
```

```
        всеми несмежными парами узлов равны бесконечности.
    Выход
      Функция модифицирует переданную на вход матрицу.
    """
    n = len(a)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                if a[i][j] > a[i][k]+a[k][j]:
                    a[i][j] = a[i][k]+a[k][j]
                    next[i][j] = next[i][k]

def recoverpath(s, t, next):
    """
    Возвращает путь из s в t, применяя матрицу next.
    """
    if next[s][t] == -1:
        return None
    path = [s]
    while s != t:
        s = next[s][t]
        path.append(s)
    return path

if __name__ == "__main__":
    fname = '../data/network-links'
    a = network2distancematrix(fname, True)
    n = len(a)
    next=[[i if a[i][j] != INF else -1
          for i in range(n)]
         for j in range(n)]
    allpairswpaths(a, next)
    print recoverpath(1, 6, next), a[1][6]
    print recoverpath(0, 7, next), a[0][7]
```

При выполнении тестовой части программы печатается следующий результат:

```
[1, 4, 6] 14
[0, 3, 7] 20
```

Значения те же, какие дает алгоритм Дейкстры, примененный к двум парам узлов (обратите внимание, что порядок следования узлов здесь противоположный). Заметим, что не всегда эти два метода должны давать в точности одинаковые результаты, поскольку между двумя узлами может быть несколько кратчайших путей с одним и тем же весом. Например, из узла 6 в узел 3 есть два кратчайших пути: 6–4–5–2–3 и 6–4–1–0–3, в обоих случаях вес равен 22. В двух алгоритмах применяются разные способы разрешения неоднозначности, поэтому и кратчайшие пути получаются разными. В нашей реализации алгоритма Дейкстры неоднозначность разрешается благодаря тому, что всегда выбирается первый индекс, для которого суммарное расстояние наименьшее. Но в алгоритме Флойда–Уоршелла, после того как

кратчайший путь найден (в табл. 10.1 это происходит на итерации 5), альтернативные кратчайшие пути не рассматриваются.

Таблица 10.1 Матрицы расстояний на каждой итерации алгоритма Флойда–Уоршелла. В левом верхнем углу каждой матрицы находится значение k

0	0	1	2	3	4	5	6	7
0	0	3	–	5	–	–	–	–
1	3	0	10	–	9	–	–	–
2	–	10	0	8	–	3	–	–
3	5	–	8	0	–	–	–	15
4	–	9	–	–	0	6	5	–
5	–	–	3	–	6	0	–	12
6	–	–	–	–	5	–	0	11
7	–	–	–	15	–	12	11	0

1	0	1	2	3	4	5	6	7
0	0	3	–	5	–	–	–	–
1	3	0	10	8	9	–	–	–
2	–	10	0	8	–	3	–	–
3	5	8	8	0	–	–	–	15
4	–	9	–	–	0	6	5	–
5	–	–	3	–	6	0	–	12
6	–	–	–	–	5	–	0	11
7	–	–	–	15	–	12	11	0

2	0	1	2	3	4	5	6	7
0	0	3	13	5	12	–	–	–
1	3	0	10	8	9	–	–	–
2	13	10	0	8	19	3	–	–
3	5	8	8	0	17	–	–	15
4	12	9	19	17	0	6	5	–
5	–	–	3	–	6	0	–	12
6	–	–	–	–	5	–	0	11
7	–	–	–	15	–	12	11	0

3	0	1	2	3	4	5	6	7
0	0	3	13	5	12	16	–	–
1	3	0	10	8	9	13	–	–
2	13	10	0	8	19	3	–	–
3	5	8	8	0	17	11	–	15
4	12	9	19	17	0	6	5	–
5	16	13	3	11	6	0	–	12
6	–	–	–	–	5	–	0	11
7	–	–	–	15	–	12	11	0

4	0	1	2	3	4	5	6	7
0	0	3	13	5	12	16	–	20
1	3	0	10	8	9	13	–	23
2	13	10	0	8	19	3	–	23
3	5	8	8	0	17	11	–	15
4	12	9	19	17	0	6	5	32
5	16	13	3	11	6	0	–	12
6	–	–	–	–	5	–	0	11
7	20	23	23	15	32	12	11	0

5	0	1	2	3	4	5	6	7
0	0	3	13	5	12	16	17	20
1	3	0	10	8	9	13	14	23
2	13	10	0	8	19	3	24	23
3	5	8	8	0	17	11	22	15
4	12	9	19	17	0	6	5	32
5	16	13	3	11	6	0	11	12
6	17	14	24	22	5	11	0	11
7	20	23	23	15	32	12	11	0

7	0	1	2	3	4	5	6	7
0	0	3	13	5	12	16	17	20
1	3	0	10	8	9	13	14	23
2	13	10	0	8	9	3	14	15
3	5	8	8	0	17	11	22	15
4	12	9	9	17	0	6	5	16
5	16	13	3	11	6	0	11	12
6	17	14	14	22	5	11	0	11
7	20	23	15	15	16	12	11	0

6	0	1	2	3	4	5	6	7
0	0	3	13	5	12	16	17	20
1	3	0	10	8	9	13	14	23
2	13	10	0	8	9	3	14	15
3	5	8	8	0	17	11	22	15
4	12	9	9	17	0	6	5	16
5	16	13	3	11	6	0	11	12
6	17	14	14	22	5	11	0	11
7	20	23	15	15	18	12	11	0

10.4. Примечания

Все описанные в этой главе алгоритмы можно найти в учебниках, и прежде всего в книге Cormen et al. (2001), где имеется их подробное описание. Время работы этих алгоритмов, как правило, определяется количеством узлов (N) и ребер (E) в сети. Алгоритмы поиска в ширину и в глубину имеют сложность $O(N + E)$. Время работы алгоритма Дейкстры (Dijkstra, 1959) тоже определяется числом узлов (N) и ребер (E). Но в зависимости от способа его реализации фактическое (по-прежнему теоретическое) время работы может быть различным. Мы использовали для нахождения минимального значения в `Q` (в функции `get_remove_min`) линейный поиск, поэтому время работы составляет $O(N^2)$. Но если применить другой способ поиска минимального значения, то удастся добиться времени $O(N + E \log E)$. Из-за наличия трех вложенных циклов в алгоритме Флойда–Уоршелла (Floyd, 1962) его сложность, очевидно, равна $O(N^3)$, теоретически это эффективнее, чем выполнение алгоритма Дейкстры для каждой пары узлов. Задача о калитке (Lombard and Church, 1993) – интересное применение алгоритма Дейкстры, есть и другие применения алгоритмов кратчайшего пути, особенно в литературе по транспортным системам (Miller and Shaw, 2001).

10.5. Упражнения

1. Придумайте свою сеть, взяв за образец сеть на рис. 10.1. Напишите для нее файл описания и прогоните с ним все алгоритмы, описанные в этой главе.
2. Функции `network2list` и `network2distancematrix` принимают аргумент `is_zero_based`. Он говорит о том, как пронумерованы узлы во входном файле: с нуля или с единицы. Путем небольшого изменения программы можно разрешить нумерацию с любого числа (при условии, конечно, что номера узлов в файле согласованы). В следующей главе мы будем иметь дело с большим количеством сетей разного размера. Прогоните алгоритмы для этих файлов и протестируйте их сложность.
3. Напишите на Python программу генерации случайных сетей. Сначала выберите число узлов, а затем начните случайным образом вставлять ребра между еще не соединенными узлами. С увеличением числа ребер свойства сети будут изменяться. Например, средняя длина кратчайшего пути, которая вначале, когда сеть была несвязной, равнялась бесконечности, постепенно уменьшается. Используя только эту простую характеристику сети, можно оценить ее связность. Для каждой конфигурации сети, определяемой количеством узлов и случайных ребер, вычисление нужно проделать многократно, потому что сети всякий раз получаются разными. Постройте график зависимости между количеством случайных ребер и связностью сети.
4. Можно ли использовать алгоритм поиска в ширину или в глубину для нахождения кратчайшего пути между двумя узлами? Объясните свой ответ.

5. Мы упомянули теоретическое различие во времени работы между алгоритмом Флойда–Уоршелла и многократным применением алгоритма Дейкстры для нахождения кратчайших путей между всеми парами узлов. Можете ли вы эмпирически проверить предсказания теории?
6. В документации по Python отмечено, что использовать список для реализации очереди неэффективно. Вместо этого можно было бы воспользоваться модулем `deque`. Перепишите реализацию поиска в ширину, воспользовавшись `deque` вместо списка. Заметна ли разница?
7. Перепишите код алгоритма Дейкстры, воспользовавшись списковым представлением сети.
8. В алгоритме Флойда–Уоршелла с восстановлением путей мы хранили в матрице `next` *следующие* узлы на пути. Но точно так же можно хранить *предшествующие* узлы на пути, начиная с конечного узла. Модифицируйте код, чтобы реализовать эту идею.
9. Мы говорили, что пути, возвращаемые алгоритмами Дейкстры и Флойда–Уоршелла, одинаковы в терминах расстояния, но могут проходить через разные узлы. Напишите на Python программу, подтверждающую это утверждение. Можете ли вы объяснить, почему пути иногда оказываются различными? Можно ли заставить эти алгоритмы всегда возвращать один и тот же путь?
10. Напишите на Python программу, которая печатает количество ребер, инцидентных каждому узлу. Это один из многих способов оценки сложности сети, а сам показатель называется степенью узла и показывает важность данного узла в сети.
11. Как можно вычислить среднюю величину кратчайшего расстояния между парами узлов? Это значение показывает близость узлов в сети. В литературе по социальным сетям оно называется центральностью по близости.
12. Напишите на Python программу, которая генерирует сети с высокой центральностью по близости.
13. Узел сети может быть важен, потому что через него проходит много путей между другими узлами. Напишите на Python программу, которая находит узел с наибольшим числом проходящих через него кратчайших путей. В литературе по социальным сетям этот показатель называется центральностью по посредничеству.
14. Напишите на Python программу, которая генерирует сети с высокой центральностью по посредничеству.

Глава 11

Пространственная оптимизация

Агент Смит: Необходимо запустить поиск.
Агент Джонс: Уже сделано.

Матрица (1999)

Для многих практических задач необходимо находить решения, являющиеся в некотором смысле наилучшими, или оптимальными, и мы полагаем, что это можно сделать перебором многих потенциальных решений. В такой задаче нужен способ сравнения решений, который позволил бы сказать, что одно из них «лучше» другого. Для оценки и сравнения решений применяется целевая функция. В этой главе нас будут интересовать задачи оптимизации, содержащие пространственные компоненты, которые необходимо рассмотреть для нахождения оптимального решения. Во многих приложениях наша цель – найти множество точек, лучшее по сравнению с другими кандидатами. Рассмотрим, к примеру, задачу нахождения места для установки вышки сотовой связи. Здесь требуется выбрать место, так чтобы базовая станция обслуживала как можно больше пользователей. Мест с широкой зоной покрытия может быть много. Но предполагается, что существует по крайней мере одно, обеспечивающее покрытие большего числа пользователей, чем любое другое. Оно и будет оптимальным решением задачи. Похожий пример – пирологический мониторинг лесов, когда требуется выбирать места для наблюдательных вышек, являющихся частью системы оповещения. Если понятие покрытия имеет другой смысл, то мы получаем другую задачу. Например, при поиске места для размещения публичной библиотеки речь идет об обслуживании не какой-то части населения, а всего населения. Как тогда оценивать оптимальность места? Можно предложить много разных показателей. С точки зрения размещения в пространстве, место выбрано эффективно, если читателям удобно добираться до библиотеки. Удобство можно измерить средним временем поездки. Среди всех возможных мест существует по меньшей мере одно с наименьшим средним временем (или расстоянием, если считать его эквивалентом времени).

Существует много видов пространственной оптимизации. В этой главе мы обсудим лишь несколько, уделив особое внимание алгоритмам решения

таких задач. В общем случае имеется два подхода к решению. Первый – точные методы; как следует из названия, их цель – найти истинно оптимальные решения. Способов поиска точных решений много. Например, алгоритм кратчайшего пути, описанный в главе 10, может найти кратчайший путь в сети путем динамического улучшения пути, найденного на предыдущих итерациях. Можно также упомянуть метод полного перебора, когда рассматриваются все возможные решения и из них выбирается лучшее. Однако в задаче оптимизации множество потенциальных решений – пространство решений задачи – обычно очень велико. Его размер зависит от конкретной задачи и определяется числом переменных. Зачастую размер очень быстро растет с увеличением числа переменных. Многие задачи оптимизации трудно решить, потому что пространство решений необозримо. Например, предположим, что в примере о выборе места для библиотеки существует 100 потенциальных мест, в которых нужно разместить пять новых библиотек. Количество вариантов называется «числом сочетаний по 5 из 100» и равно $100!/(5! \times 95!) = 75\,287\,520$. Уже это число велико, но давайте рассмотрим еще один пример. Пусть имеется 20 городов, и мы хотим найти маршрут минимальной длины (или стоимости), который проходит через каждый город ровно один раз. Требуется просто предъявить последовательность посещения городов, притом что расстояние между любой парой городов известно. Это будет последовательность из 20 элементов. В i-й позиции ($0 \leqslant i \leqslant 19$) может находиться любой из $20 - i$ элементов, так что всего вариантов будет $20! = 2\,432\,902\,008\,176\,640\,000$. Огромное число, хотя городов всего 20. Эта очень трудная задача называется задачей коммивояжера.

Теперь должно быть понятно, что поиск оптимума в пространстве решений может занять *очень* много времени. Так, в задаче коммивояжера с 20 городами компьютеру, просматривающему миллиард маршрутов в секунду, на полный перебор потребуется 2 432 902 008 секунд, т. е. больше 77 лет. Но насколько реалистично предположение о просмотре миллиарда маршрутов в секунду? Проверим его с помощью следующей простой программы на Python:

```
import time

time1 = time.time()
sum = 0
while sum < 1000000000:
    sum += 1
time2 = time.time()
print sum, time2-time1
```

Результат не радует: на моем MacBook Air (1.3 GHz Intel Core i5) программа работала 122.10 секунды. Это очень долго! Но, как мы знаем, Python – интерпретируемый, а не компилируемый язык. Ладно, запустим однострочную программу[1] на компилируемом языке C. Лучше – понадобилось чуть больше 4 секунд. Однако мы всего-то выполнили миллиард итераций почти пустого цикла. Для оценки маршрута нужно было бы получить все расстояния и вы-

[1] `int main() {double sum = 0; while (sum < 1000000000) sum += 1; }`.

числить их сумму. Так что для достижения заявленной скорости понадобился бы очень мощный (быстрый) компьютер.

Очевидно, что для нахождения наилучшего решения в таком большом множестве нужно что-то более хитроумное. В противном случае поиск и оценка каждого потенциального решения быстро станут безнадежным делом при увеличении размера задачи (например, количества городов в задаче коммивояжера). Одним из примеров такого подхода является алгоритм кратчайшего пути, который не исследует все возможные пути при поиске оптимального. Для достижения очень достойного времени работы он игнорирует большую часть потенциальных решений. Именно так проектируются хорошие алгоритмы: они стремятся воспользоваться структурой задачи, чтобы уменьшить размер пространства решений.

К сожалению, не все точные методы могут эффективно уменьшить размер пространства решений. Бывает, впрочем, что нам и не нужно, чтобы решение было строго оптимальным, достаточно и близкого к оптимальному. Но для согласия на субоптимальное решение есть и более важная причина: решение оптимально только в контексте конкретной постановки задачи. На практике теоретически оптимальное решение может оказаться ужасным, потому что при постановке не были учтены какие-то факторы. Ну и конечно, мы хотим находить хорошие решения быстро. Тут-то и выходят на сцену эвристические методы: они не гарантируют нахождения оптимального решения, но позволяют быстро находить высококачественные, почти оптимальные решения.

Мы обсудим оба подхода по очереди. В этой главе рассмотрим точные методы решения задач о выборе местоположения, а в следующей перейдем к различным эвристическим методам. Точные алгоритмы можно отнести к нескольким классам. Алгоритмы кратчайшего пути – примеры динамического программирования. Невозможно в этой главе рассмотреть все виды задач оптимизации и методы их решения, да мы и не ставим такой цели. Мы сосредоточимся на том, как проектируются и реализуются подобные алгоритмы, с учетом наших знаний о географических данных. В этой главе мы изучим еще два вида точных алгоритмов поиска. Первый связан с задачей нахождения центра множества точек, в алгоритмах ее решения так или иначе используются геометрические свойства задачи. Второй представляет более общий подход: мы пытаемся свести решаемую задачу оптимизации к более общей, а затем применить методы решения этой общей задачи к нашему частному случаю. Более общей у нас будет задача линейного программирования, к которой сводится много задач оптимизации. Мы рассмотрим применение этого подхода к решению полезной задачи пространственной оптимизации – задачи о p-медиане.

11.1. Задача о 1-центре

Пусть дано множество точек, требуется найти наименьшую окружность, охватывающую все точки. Эта задача многократно изучалась под разными названиями, в т. ч. задача о 1-центре, задача о наименьшей окружности и за-

дача о наименьшей ограничивающей сфере. Если речь идет о моделировании размещения объектов, то мы предпочитаем называть ее задачей о 1-центре, но в принципе это общая задача, имеющая применения в разных областях. Итак, нам дано конечное множество точек в пространстве. Окружность можно задать положением центра и радиусом, и очевидно, что центр может находиться в любой точке непрерывного пространства. Мы рассмотрим три алгоритма точного решения данной задачи и сравним их эффективность. Во всех этих алгоритмах выбирается начальная окружность, а затем ищется способ перейти к оптимальной окружности, охватывающей все точки. Вначале берется небольшая окружность, охватывающая лишь часть точек, или, наоборот, большая окружность, охватывающая все точки. Затем применяется итеративный процесс, который либо увеличивает, либо уменьшает окружность, пока не будет найдено оптимальное решение.

Но прежде чем приступать к алгоритмам, решим несколько вопросов, касающихся структур данных. Нам нужна структура данных, пригодная для всех алгоритмов. В листинге 11.1 приведен Python-класс `disc`, предназначенный для этой цели. В строке 3 мы сообщаем интерпретатору путь к классу `Point`, который находится в файле `point.py` (листинг 2.1) в каталоге `geom` на том же уровне иерархии, что и текущий каталог[1]. В классе `disc` хранится два важных элемента информации: центр и радиус окружности. Экземпляр класса `disc` можно инициализировать двумя способами: явно задав положение центра и радиус (строка 10) или указав две либо три точки (строка 13). В последнем случае нам придется вычислить центр и радиус окружности, проходящей через две или три точки; это делают включенные в код функции. Подробности этих фундаментальных геометрических вычислений в книге не рассматриваются, а сами функции будут использоваться и в других программах в этой и следующей главах.

Листинг 11.1 ❖ Структура данных, используемая в алгоритмах решения задачи о 1-центре (disc.py)

```
 1 from math import fabs, sqrt
 2 import sys
 3 sys.path.append('../geom')
 4 from point import *
 5
 6 __all__ = ['disc']
 7
 8 class disc:
 9     def __init__(self, center=None, radius=None, points=None):
10         if points == None:
11             self.center = center
12             self.radius = radius
13         else:
14             if len(points)==2:
```

[1] Заметим, что все наши Python-программы хранятся в различных подкаталогах каталога `programs`. Так проще организовать код, относящийся к одной теме, но при этом требуется в начале программы указывать путь к другим классам.

```
15                 res = make_disc2(points[0],points[1])
16             elif len(points)==3:
17                 res = make_disc(points[0],points[1],points[2])
18             else:
19                 res = [None, None]
20             self.center = res[0]
21             self.radius = res[1]
22     def __eq__(self, other):
23         return self.center==other.center and\
24                self.radius==other.radius
25     def __repr__(self):
26         return "({0}, {1})".format(
27             self.center, self.radius)
28     def inside(self, p):
29         dx = fabs(self.center.x - p.x)
30         dy = fabs(self.center.y - p.y)
31         if dx>self.radius or dy>self.radius:
32             return False
33         if self.center.distance(p) <= self.radius:
34             return True
35         return False
36
37 def make_disc2(p1, p2):
38     dx = fabs(p1.x - p2.x)
39     dy = fabs(p1.y - p2.y)
40     radius = sqrt(dx*dx + dy*dy)/2.0
41     x = min(p1.x, p2.x) + dx/2.0
42     y = min(p1.y, p2.y) + dy/2.0
43     return Point(x, y), radius
44
45 def make_disc(p1, p2, p3):
46     x1, x2, x3 = p1.x, p2.x, p3.x
47     y1, y2, y3 = p1.y, p2.y, p3.y
48     a = fabs(x2-x1)
49     b = fabs(x3-x1)
50     c = fabs((y2-y1)/a - (y3-y1)/b)
51     xs = a
52     if b < xs:
53         xs = b
54     if c < xs:
55         xs = c
56     a = fabs(y2-y1)
57     b = fabs(y3-y1)
58     c = fabs((x2-x1)/a - (x3-x1)/b)
59     ys = a
60     if b < ys:
61         ys = b
62     if c < ys:
63         ys = c
64     if xs < ys:    # исключить x, вычислить сначала y
65         return make_disc_x(p1, p2, p3)
66     else:
```

```
67          return make_disc_y(p1, p2, p3)
68
69  def make_disc_y(p1, p2, p3):
70      x1 = p1.x
71      x2 = p2.x
72      x3 = p3.x
73      y1 = p1.y
74      y2 = p2.y
75      y3 = p3.y
76      t1 = (x1*x1-x3*x3+y1*y1-y3*y3)/(2*(x3-x1))
77      t2 = (x1*x1-x2*x2+y1*y1-y2*y2)/(2*(x2-x1))
78      t3 = (y2-y1)/(x2-x1) - (y3-y1)/(x3-x1)
79      y = (t1 - t2)/t3
80      x = -(2*(y2-y1)*y + x1*x1 - x2*x2 + y1*y1 -
81            y2*y2) / (2*(x2-x1))
82      r = sqrt((x1-x)*(x1-x) + (y1-y)*(y1-y))
83      return Point(x, y), r
84
85  def make_disc_x(p1, p2, p3):
86      x1, x2, x3 = p1.x, p2.x, p3.x
87      y1, y2, y3 = p1.y, p2.y, p3.y
88      t1 = (x1*x1-x3*x3+y1*y1-y3*y3)/(2*(y3-y1))
89      t2 = (x1*x1-x2*x2+y1*y1-y2*y2)/(2*(y2-y1))
90      t3 = (x2-x1)/(y2-y1) - (x3-x1)/(y3-y1)
91      x = (t1 - t2)/t3
92      y = -(2*(x2-x1)*x + x1*x1 - x2*x2 + y1*y1 -
93            y2*y2) / (2*(y2-y1))
94      r = sqrt((x1-x)*(x1-x) + (y1-y)*(y1-y))
95      return Point(x, y), r
```

Первый алгоритм решения задачи о 1-центре называется алгоритмом Кристала–Пирса, ему уже больше ста лет (см. примечания). Предположим, что все точки хранятся в списке P. Алгоритм начинает работу с очень большой окружности, которая проходит через две точки из P и заключает внутри себя все точки. Без ограничения общности обозначим эти точки A и B. Теперь мы ищем в P третью точку C (не совпадающую с A и B), для которой угол ACB минимален. Найденное решение оптимально, если выполнено одно из двух условий. Во-первых, если треугольник ABC не является тупоугольным, т. е. максимальный угол не больше 90°, то эти три точки определяют минимальную ограничивающую окружность. Во-вторых, если угол ACB тупой, то окружность минимальна, если точки A и B лежат на диаметре. Во всех остальных случаях поиск нужно продолжать. Переобозначим A и B точки, являющиеся вершинами нетупых углов треугольника, и найдем новую точку C. Этот процесс продолжается, пока не будет выполнено одно из сформулированных выше условий.

Оригинальный алгоритм начинается с очень большой окружности, охватывающей все точки и проходящей через две соседние точки, принадлежащие выпуклой оболочке заданного множества точек. Однако искать выпуклую оболочку и окружность, охватывающую все точки, долго; временная сложность этой процедуры составляет $O(n \log n)$, где n – число точек. В более

поздних версиях нахождение большой начальной окружности производится более эффективно, за время $O(n)$. Эта новая стратегия, названная методом Чакраборты–Чаудхури, ищет в P точку a, отстоящую дальше других от начала координат. Затем ищется другая точка, для которой достигает максимума выражение

$$\frac{d_a d(ab)^2}{|x_a(x_a - x_b) + y_a(y_a - y_b)|},$$

где d_a – расстояние от a до начала координат, а $d(ab)$ – расстояние между точками a и b. Окружность, проходящая через a и b, заведомо охватывает все точки P.

В листинге 11.2 приведен код алгоритма Кристала–Пирса. Он содержит четыре вспомогательные функции. Функция `get_angle` (строка 8) вычисляет угол, образованный первой точкой и остальными двумя. Функция `find_mini_angle` (строка 24) находит точку C, для которой угол ACB минимален. Функция `getfirsttwo` (строка 42) находит две точки, которые можно использовать для построения очень большой окружности на начальном шаге алгоритма со стратегией Чакраборты–Чаудхури. Функция `moveminimaxpoints` (строка 64) перемещает две начальные точки в начало списка.

Функция `oncenter1` – основа алгоритма поиска (строка 72). Поскольку мы используем первые две точки в P в качестве A и B, список необходимо предварительно подготовить. При поиске в P точки C мы не удаляем физически уже рассмотренные точки, а используем список целых чисел `unmarked`, в котором отмечаем, какие точки уже были просмотрены (строки 74). В строках 76 и 77 назначаются точки A и B. Условия оптимальности решения проверяются в строках 82 и 86. Для остальных случаев (строка 89) мы подставляем C вместо текущей A или B в зависимости от того, какая из них является вершиной тупого угла.

Листинг 11.2 ❖ Алгоритм Кристала–Пирса решения задачи о 1-центре (cp.py)

```
 1 from math import sqrt, atan2, fabs, pi
 2 from disc import *
 3 import sys
 4 sys.path.append('../geom')
 5 from point import *
 6 import random
 7
 8 def get_angle(p0, p1, p2):
 9     """
10     Возвращает угол между прямыми p0p1 и p0p2 (предполагаются
11     направления p0 -> p1, p0 -> p2)
12     """
13     a1 = atan2(p1.y-p0.y, p1.x-p0.x)
14     a2 = atan2(p2.y-p0.y, p2.x-p0.x)
15     if a1 < 0:
16         a1 = 2*pi - fabs(a1)
17     if a2 < 0:
18         a2 = 2*pi - fabs(a2)
```

```
19      degree = fabs(a1-a2)
20      if degree > pi:
21          degree = 2*pi - degree;
22      return degree;
23
24  def find_mini_angle(S1, A, B, um):
25      n = len(S1)
26      angle = 100
27      c = -1
28      for i in xrange(n):
29          if S1[i] == A or S1[i] == B:
30              continue
31          if not um[i]:
32              continue
33          angle1 = get_angle(S1[i], A, B)
34          if angle1 < angle:
35              angle = angle1
36              c = i                # D = S1[i]
37          if angle1 < pi/2.0:
38              continue
39          um[i] = 0
40      return angle, c
41
42  def getfirsttwo(S):
43      b = Point(0, 0)
44      dist = 0
45      ix = 0
46      for i, p in enumerate(S):    # самая далекая от начала координат точка
47          d1 = p.distance(b)
48          if d1 > dist:
49              dist = d1
50              ix = i
51              a = p
52      dist = 0
53      dd = sqrt(a.x*a.x+a.y*a.y)
54      for i, p in enumerate(S):
55          if i==ix: continue
56          d1 = (a.x-p.x)*(a.x-p.x) + (a.y-p.y)*(a.y-p.y)
57          d1 = d1*dd / fabs(a.x*(a.x-p.x) + a.y*(a.y-p.y))
58          if d1 > dist:
59              dist = d1
60              b = p
61              iy = i
62      return ix, iy
63
64  def moveminimaxpoints(points):
65      mini, maxi = getfirsttwo(points)
66      # положить minx = 0, maxx = 1
67      if maxi != 0:
68          points[0],points[mini] = points[mini], points[0]
69      if mini != 1:
70          points[1],points[maxi] = points[maxi], points[1]
71
```

```
 72 def onecenter1(P):
 73     n = len(P)
 74     unmarked = [1 for i in range(n)]
 75     done = False
 76     a = 0
 77     b = 1
 78     while not done:
 79         A, B = P[a], P[b]
 80         angle, c = find_mini_angle(P, A, B, unmarked)
 81         C = P[c]
 82         if angle > pi/2.0:
 83             done = True
 84             d = disc(points=[A, B])
 85         elif get_angle(B, A, C) < pi/2.0 and\
 86              get_angle(A, B, C) < pi/2.0:
 87             done = True
 88             d = disc(points=[A, B, C])
 89         else:
 90             if (get_angle(B, A, C)) > pi/2.0:
 91                 unmarked[b] = 0
 92                 b = c
 93         else:
 94             unmarked[a] = 0
 95             a = c
 96     return d
 97
 98 def test():
 99     npts = 5
100   points = []
101   for i in xrange(npts):
102       p = Point(random.random(), random.random())
103       points.append(p)
104   print points
105   print onecenter1(points)
106   moveminimaxpoints(points)
107   print points
108   print onecenter1(points)
109
110 if __name__ == '__main__':
111     test()
```

Теперь протестируем алгоритм Кристала–Пирса на пяти случайных точках (функция `test`). В первом эксперименте список точек не менялся, поэтому не гарантируется, что первые две точки определяют окружность, охватывающую все точки (строка 105). Затем использовалась стратегия выбора начальных точек (строка 106). Прогнав тест несколько раз, мы обнаружим, что для пяти точек алгоритм обычно находит лучшее решение и без использования стратегии инициализации. Но иногда можно столкнуться с такой ситуацией:

```
[(0.4, 0.8), (0.1, 0.3), (0.0, 0.8), (0.8, 0.4), (0.5, 0.4)]
((0.4, 0.4), 0.371971373225)
[(0.4, 0.8), (0.8, 0.4), (0.0, 0.8), (0.1, 0.3), (0.5, 0.4)]
((0.4, 0.6), 0.450773182422)
```

где, как мы видим, два результата не совпадают. Алгоритм не проверяет, все ли точки находятся внутри, поэтому важно начинать с большой окружности, чтобы гарантировать полный охват.

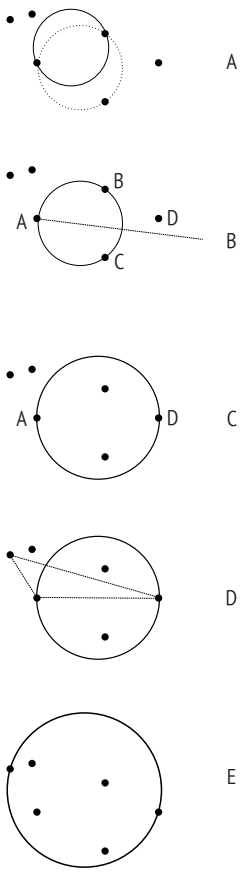

Рис. 11.1 ❖ Алгоритм Эльзинги–Хирна решения задачи о 1-центре: (A) две начальные точки и окружность (сплошная линия). Поскольку начальная окружность не охватывает все точки, добавляется еще одна точка, образующая вместе с предыдущими остроугольный треугольник. Окружность, проходящая через все три точки (пунктирная), по-прежнему не охватывает все точки; (B) выбирается находящаяся вне окружности точка (*D*), а три точки, лежащие на окружности, обозначаются соответственно *A*, *B* и *C*. Точка *C* является вершиной тупого угла и потому отбрасывается; (C) оставшиеся две точки (*A* и *D*) лежат на окружности, которая не охватывает все точки; (D) добавляется еще одна точка, но эти три точки образуют тупоугольный треугольник, и точка в вершине наибольшего угла отбрасывается; (E) окончательная окружность, проходящая через две точки, охватывает все точки

Из обсуждения выше ясно, что минимальная окружность, ограничивающая множество точек, определяется либо двумя точками, расположенными

на противоположных концах диаметра, либо тремя точками, образующими остроугольный треугольник. Алгоритм Эльзинги–Хирна разработан с учетом этого факта. Он начинается с построения окружности, проходящей через какие-либо две точки из *P*, а затем окружность итеративно увеличивается, пока не будет найдено оптимальное решение. На каждой итерации выполняется три шага: (1) если окружность, проходящая через две точки, не охватывает все точки, то выбирается точка, лежащая вне окружности; (2) если эти три точки образуют прямоугольный или тупоугольный треугольник, то точка в вершине прямого или тупого угла отбрасывается, и мы переходим к шагу 1, считая две оставшиеся точки начальными; (3) теперь имеется три точки, образующие остроугольный треугольник. Если окружность, проходящая через эти три точки, не охватывает все точки, то итерации не закончились, и мы выбираем точку, лежащую вне окружности. Обозначим эту точку *D*, а из трех предыдущих точек обозначим *A* ту, что отстоит дальше других от *D*. Затем проведем прямую через *A* и центр окружности. Ту из двух остальных точек, что находится от прямой по ту же сторону, что и *D*, обозначим *B*, а другую *C* (рис. 11.1B). Неоднозначность можно разрешать произвольно. Далее переходим к шагу 2, используя точки *A*, *C* и *D*. На рис. 11.1 этот алгоритм иллюстрируется наглядно.

Код алгоритма Эльзинги–Хирна представлен в листинге 11.3. Функция `right_obtuse_triangle` проверяет, является ли треугольник, построенный по трем точкам, прямоугольным или тупоугольным. Если да, то она возвращает `True` и переупорядочивает список точек, так что в начале оказываются вершины острых углов. Функция `find_three` принимает три точки, проходящую через них окружность и еще одну точку, лежащую вне этой окружности, и находит три точки *A*, *C* и *D*, которые будут использованы на следующем шаге. Цикл `for` в строке 36 находит точку *A*, отстоящую от *D* дальше, чем другая. Точка *D* лежит либо выше, либо ниже прямой, проходящей через *A* и центр окружности. Какой именно случай имеет место, легко проверить, подставив координаты *D* в уравнение прямой (строки 48 и 49). Цикл `for` в строке 52 находит точку, расположенную по ту же сторону, что и *D*, от прямой, проходящей через *A* и центр окружности. Если оказывается, что точка *A*, центр окружности и точка *D* в точности или почти коллинеарны (строка 59), то понять, по какую сторону от прямой лежит точка, невозможно, и тогда мы берем в качестве *C* точку с наименьшей абсолютной величиной. Функция `cover_all` проверяет, охватывает ли окружность все точки, при этом строка 72 необходима, потому в Python числа с плавающей точкой имеют конечную точность, из-за чего возможно небольшое расхождение между истинным и вычисленным расстоянием между двумя точками. Поэтому мы должны вручную исключать точки, по которым построены окружности, из числа проверяемых. Функция возвращает две переменные: признак `True` или `False`, сообщающий, все ли точки охвачены, и первую неохваченную точку, если таковые имеются. Найденная таким образом точка становится точкой, лежащей вне окружности, которая необходима алгоритму. Сама функция, реализующая алгоритм (`onecenter2`), короткая и в точности соответствует приведенному выше описанию (строка 78).

Листинг 11.3 ❖ Алгоритм Эльзинги–Хирна (elzinga_hearn.py)

```
from math import pi
import random
from disc import *
import sys
sys.path.append('../geom')
from point import *
from cp import get_angle

def right_obtuse_triangle(p3):
    """
    Если true, сделать так, что p3[0] и p3[1] определяют самую длинную
    сторону, а p3[2] - вершина угла >= 90
    """
    angle0 = get_angle(p3[0], p3[1], p3[2])
    angle1 = get_angle(p3[1], p3[0], p3[2])
    angle2 = pi - angle0 - angle1
    maxa = -1.0
    maxi = -1
    for i, a in enumerate([angle0, angle1, angle2]):
        if a > maxa:
            maxa = a
            maxi = i
    if maxa >= pi/2.0:
        if maxi != 2:
            p3[maxi], p3[2] = p3[2], p3[maxi]
        return True
    return False

def find_three(p3, D, d):
    """
    Получив три точки в p3, внешнюю точку D и окружность d,
    найти A, C, D и присвоить их элементам p3[0], p3[1] и p3[2]
    соответственно
    """
    maxd = 0
    for i in range(len(p3)):
        tmpd = p3[i].distance(D)
        if tmpd > maxd:
            maxd = tmpd
            iA = i
    x1 = p3[iA].x
    x2 = d.center.x
    y1 = p3[iA].y
    y2 = d.center.y
    a = y2-y1
    b = -(x2-x1)
    c = (x2-x1)*y1 - (y2-y1)*x1
    eqd = a*D.x + b*D.y + c
    positive = eqd > 0
    eq = [0 for i in range(3)]
    iC = -1
```

```
    for i in range(3):
        if i==iA:
            eq[i] = 0.0
        else:
            eq[i] = a*p3[i].x + b*p3[i].y + c
            if positive != ((eq[i]>0)):
                iC = i
    if iC == -1:
        tempf = 100000000.0
        for i in range(3):
            if i == iA:
                continue;
            if fabs(eq[i]) < tempf:
                tempf = fabs(eq[i])
                iC = i
    p3[0], p3[1], p3[2] = p3[iA], p3[iC], D
    return

def cover_all(points, d, pp):
    for p in points:
        if p in pp:                          # учесть ограниченную точность float
            continue
        if not d.inside(p):
            return False, p
    return True, None

def onecenter2(P):
    p3 = [Point(-1, -1) for i in range(3)]
    p3[0] = P[0]
    p3[1] = P[1]
    d = disc(points=[p3[0], p3[1]])
    n = len(P)
    cnt = 0
    stop, p3[2] = cover_all(P, d, p3[:2])
    while not stop:
        if right_obtuse_triangle(p3):       # прямоугольный/тупоугольный треугольник
            d = disc(points=[p3[0], p3[1]])
            stop, p3[2] = cover_all(P, d, p3[:2])
        else:                               # остроугольный треугольник
            d = disc(points=[p3[0], p3[1], p3[2]])
            stop, pd = cover_all(P, d, p3) # pd outside d
            if not stop:
                find_three(p3, pd, d)
        cnt += 1
    return d

def test():
    npts = 50
  points = []
  for i in xrange(npts):
      p = Point(random.random(), random.random())
      points.append(p)
```

```
104       print onecenter2(points)
105
106 if __name__== '__main__':
107     test()
```

Тестирование алгоритма не вызывает затруднений. Ниже приведен результат выполнения функции `test`:

```
[(0.1, 0.3), (0.9, 0.5), (0.6, 0.9), (0.1, 0.8), (0.6, 0.6)]
((0.5, 0.5), 0.440163316231)
```

Третий алгоритм решения задачи о 1-центре предложен Вельцлем. Вначале по первым двум точкам в списке строится небольшая окружность, которая затем расширяется путем включения новых точек. В листинге 11.4 приведен соответствующий код. Основная функция `minidisc` (строка 27) сначала создает минимальную окружность, содержащую первые две точки (строка 30), после чего в цикле добавляет новые точки (строка 31), пока все точки не будут охвачены. На каждой итерации цикла создается новая минимальная окружность, которая охватывает все прежние точки плюс дополнительную точку *q*. Это делается в функции `minidiscwithpoint` (строка 35). Мы знаем, что новая точка *q* не лежит внутри предыдущей окружности, а значит, она должна лежать на границе новой, большей окружности, поскольку эта окружность должна ее охватывать. Функция `minidiscwithpoint` гарантирует, что новая окружность включает *q* и все ранее охваченные точки. Ее работа начинается с построения окружности, включающей *q* и еще одну точку (строка 18). После этого мы рассматриваем дополнительные точки. Если точка уже находится внутри окружности (строка 21), то увеличивать окружность не нужно. В противном случае мы вызываем функцию `minidiscwith2points`, которая создает окружность, проходящую через три точки: *q*, новую точку и еще одну точку из числа еще не проанализированных (строка 24). Прогон функции `test` должен дать такие же результаты, как в случае алгоритма Эльзинги–Хирна.

Листинг 11.4 ❖ Алгоритм Вельцля для решения задачи о 1-центре (welzl.py)

```
 1 import random
 2 from disc import *
 3 import sys
 4 sys.path.append('../geom')
 5 from point import *
 6
 7 def minidiscwith2points(P, q1, q2, D):
 8     D[0] = disc(points = [q1, q2])
 9     n = len(P)
10     for k in range(n):
11         if D[k].inside(P[k]):
12             D[k+1] = D[k]
13         else:
14             D[k+1] = disc(points=[q1, q2, P[k]])
15     return D[n]
16
```

```
def minidiscwithpoint(P, q, D):
    D[0] = disc(points = [P[0], q])
    n = len(P)
    for j in range(1, n):
        if D[j-1].inside(P[j]):
            D[j] = D[j-1]
        else:
            D[j] = minidiscwith2points(P[:j], P[j], q, D)
    return D[n-1]

def minidisc(P):
    n = len(P)
    D = [ disc() for i in range(n)]
    D[1] = disc(points=[P[0], P[1]])
    for i in range(2, n):
        if D[i-1].inside(P[i]):
            D[i] = D[i-1]
        else:
            D[i] = minidiscwithpoint(P[:i], P[i], D)
    return D[n-1]

def test():
    n = 5
    points = []
    for i in xrange(n):
        p = Point(random.random(), random.random())
        points.append(p)
    print points
    print minidisc(points)

if __name__ == '__main__':
    test()
```

Теперь сравним производительность трех алгоритмов, описанных в этом разделе. Программа в листинге 11.5 проводит эксперименты на 50 000 точек. Тестируется два способа подготовки точек. В первом начальные точки во всех алгоритмах выбираются случайно. Во втором применяется метод Чакраборты–Чаудхури, когда вначале выбираются две точки, расположенные далеко друг от друга.

Листинг 11.5 ❖ Тестирование алгоритмов решения задачи о 1-центре (test_1center.py)

```
from elzinga_hearn import *
from welzl import *
from cp import *

import time

n = 50000
points = [ Point(random.random(), random.random())
```

```
 9 for i in range(n) ]
10
11 ########################################
12 #
13 # Тестирование производительности со случайно выбранными начальными точками
14 #
15 ########################################
16
17 time1 = time.time()
18 d1 = minidisc(points)
19 time2 = time.time()
20 d1t = time2-time1
21
22 d2 = onecenter2(points)
23 time3 = time.time()
24 d2t = time3-time2
25
26 d3 = onecenter1(points)
27 time4 = time.time()
28 d3t = time4-time3
29
30 print "Вельцль ", d1t, d1
31 print "Эльзинга-Хирн ", d2t, d2
32 print "Кристал-Пирс ", d3t, d3
33
34 ########################################
35 #
36 # Тестирование производительности с подготовкой данных
37 #
38 ########################################
39
40 time0 = time.time()
41 moveminimaxpoints(points)
42
43 time1 = time.time()
44 d1 = minidisc(points)
45 time2 = time.time()
46 d1t = time2-time1
47
48 d2 = onecenter2(points)
49 time3 = time.time()
50 d2t = time3-time2
51
52 d3 = onecenter1(points)
53 time4 = time.time()
54 d3t = time4-time3
55
56 print "Подготовка данных", time1-time0
57
58 print "Вельцль ", d1t, d1
59 print "Эльзинга-Хирн ", d2t, d2
60 print "Кристал-Пирс ", d3t, d3
```

Ниже приведен результат одного из многих прогонов. Первое число в каждой строке – время решения с помощью конкретного алгоритма. Хотя это лишь один пример, результат типичен для многих прогонов на моем компьютере. Мы видим, что без предварительной подготовки данных алгоритм Кристала–Пирса работает дольше, чем два других. Подготовка данных методом Чакраборты–Чаудхури занимает совсем небольшую часть времени работы алгоритма, но зато существенно уменьшает общее время, как показывают последние три строки. Отметим также, что этот пример еще раз подтверждает, что алгоритм Кристала–Пирса не всегда находит оптимальную окружность без предварительной подготовки данных.

```
Вельцль             0.856627941132 ((0.5, 0.5), 0.704372949832)
Эльзинга-Хирн       0.605397939682 ((0.5, 0.5), 0.704372949832)
Кристал-Пирс        0.903444051743 ((0.5, 0.5), 0.703718274146)
Подготовка данных   0.0939598083496
Вельцль             0.264666080475 ((0.5, 0.5), 0.704372949833)
Эльзинга-Хирн       0.301491975784 ((0.5, 0.5), 0.704372949832)
Кристал-Пирс        0.237596035004 ((0.5, 0.5), 0.704372949832)
```

11.2. Задачи размещения

Задача о 1-центре – это частный случай пространственной оптимизации, в котором ищется только одна точка размещения. Но нередко требуется найти несколько точек размещения объектов (пожарных депо, складов или библиотек).

В этом разделе нас будет интересовать один конкретный тип таких задач оптимизации, а именно задача о p-медиане, в которой цель – разместить p предприятий в сети из n узлов, так чтобы минимизировать сумму расстояний от любого узла до ближайшего к нему предприятия. Каждый из n узлов считается клиентом, и мы хотим выбрать из них p узлов для размещения предприятий.

Для нахождения точного решения задачи сначала поставим ее математически как задачу частично целочисленного программирования. Места возможного размещения в сети будем называть узлами или вершинами (оба термина употребляются как синонимы), обозначим d_{ij} расстояние между i-м и j-м узлом – индексом i будем обозначать предприятия, а индексом j – клиентов. В каждом узле существует спрос на услуги предприятий, обозначим a_i величину этого спроса в i-м узле. Чтобы описать, размещено ли предприятие в некотором узле, определим переменную x_{ij}, которая принимает значение 1, если спрос в вершине i удовлетворяется предприятием в вершине j, и 0 в противном случае. При этом x_{jj} равно 1, если в вершине j размещено предприятие, которое обслуживает само себя, и 0 в противном случае. В этих обозначениях задача о p-медиане формулируется как следующая задача частично целочисленного программирования:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} a_i d_{ij} x_{ij}, \tag{11.1}$$

при условии $$\sum_{j=1}^{n} x_{ij} = 1 \quad \forall i, \tag{11.2}$$

$$\sum_{j=1}^{n} x_{jj} = p, \tag{11.3}$$

$$x_{ij} - x_{jj} \leqslant 0 \quad \forall i, j \text{ таких, что } i \neq j, \tag{11.4}$$

$$x_{ij} = 0, 1 \quad \forall i, j. \tag{11.5}$$

Формула (11.1) описывает целевую функцию, минимизирующую суммарное взвешенное расстояние, в которой весами являются значения спроса. Каждое из ограничений (11.2) говорит, что спрос в узле i удовлетворяется одним и только одним из выбранных узлов. Ограничение (11.3) говорит, что всего для размещения предприятий выбрано ровно p узлов. Ограничение (11.4) означает, что если спрос в узле i удовлетворяется предприятием j (т. е. $x_{ij} = 1$), то должно существовать предприятие, размещенное в узле j (т. е. $x_{jj} = 1$) такое, что $x_{ij} - x_{jj} \leqslant 0$. А если в узле j нет предприятия ($x_{jj} = 0$), то спрос в узле i не должен удовлетворяться узлом j ($x_{ij} = 0$), откуда снова следует, что $x_{ij} - x_{jj} \leqslant 0$. Чтобы завершить описание ограничения (11.4), нужно рассмотреть еще два случая (см. упражнения в конце главы). Наконец, ограничение (11.5) означает, что параметры решения (x_{ij} и x_{jj}) двоичные, т. е. принимают только значения 0 или 1.

Стандартная постановка задачи о p-медиане позволяет применить для ее решения готовые программы. В частности, свою эффективность доказали коммерческие программы GUROBI и CPLEX. Можно также использовать программы решения с открытым исходным кодом, и в этой главе, как и во всех остальных, мы будем работать как раз с такой программой `lp_solve`[1]. Конечно, можно возразить, что программы с открытым исходным кодом, мол, плохо протестированы, но для демонстрации методов они вполне пригодны.

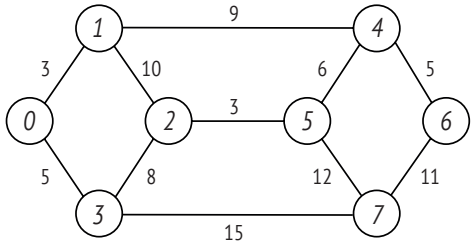

Рис. 11.2 ❖ Пример сети. Узлы помечены курсивными цифрами, а рядом с ребрами проставлены их длины (без учета масштаба)

[1] http://lpsolve.sourceforge.net/5.5/.

На рис. 11.2 повторен пример сети, на которой мы тестировали методы анализа в главе 10. Чтобы воспользоваться решателем, данные задачи нужно представить в формате, понятном программе. Мы воспользуемся популярным форматом файлов LP. Его структура проста: необходимо буквально записать целевую функцию и ограничения. Важно следить за тем, чтобы все параметры решения имели уникальные имена. В нашем простом примере параметр x_{ij} записывается в виде `x_i_j` (например, `x_5_16` для $i = 5$ и $j = 16$).

Представленная в листинге 11.6 программа формирует LP-файл для нашей задачи о p-медиане. Мы снова используем алгоритм нахождения кратчайших путей между всеми парами (листинг 10.7, импортированный в строке 6) для вычисления матрицы расстояний d_{ij}, участвующей в постановке задачи о p-медиане. В строке 19 читается входной файл, содержащий только данные об узлах и инцидентных им ребрах, а в строке 20 вычисляется матрица расстояний. Мы предполагаем, что для всех узлов сети спрос равен 1. Блок, начинающийся в строке 23, создает строковые представления параметров решения. Целевая функция и все ограничения хранятся в виде строк. В конце программы распечатывается LP-файл.

Листинг 11.6 ❖ Создание LP-файла для задачи о p-медиане (pmed_lpformat.py)

```
 1 import sys
 2 from string import atoi
 3 import sys
 4 sys.path.append('../networks')
 5 from network2listmatrix import network2distancematrix
 6 from allpairdist import *
 7
 8 if len(sys.argv) <=1:
 9     print sys.argv[0], "filename [ [True|False] p]"
10     sys.exit()
11
12 zerobased = True
13 p = 2
14 if len(sys.argv)>=3 and sys.argv[2]=="False":
15     zerobased = False
16 if len(sys.argv)>=4:
17     p = atoi(sys.argv[3])
18
19 a = network2distancematrix(sys.argv[1], zerobased)
20 allpairs(a)
21 n = len(a)
22
23 X = [['']*n for i in xrange(n)]
24 for i in xrange(n):
25     for j in xrange(n):
26         X[i][j] = 'x_%d_%d'%(i, j)
27
28 # целевая функция
29 obj='MIN: '
30 for i in range(n):
31     for j in range(n):
```

```
        obj += "%d"%(a[i][j])+"*"+X[i][j]      # %f, если числа вещественные
            if not (i == n-1 and j == n-1):
                obj += '+'
            else:
                obj=obj+';'

# ограничения
con1 = ['' for i in range(n)]
for i in range(n):
    con1[i] = X[i][0]
    for j in range(1,n):
        con1[i] += '+'+X[i][j]
    con1[i] += '=1;'

con2 = X[0][0]
for j in range(1, n):
    con2 += '+'+X[j][j]
con2 += '=%d;'%(p)

con3 = []
for i in range(n):
    for j in range(n):
        if i <> j:
            con3.append(X[i][j] + '-' + X[j][j] + '<=0;')

con4 = 'BIN '
for i in range(n):
    for j in range(n):
        con4 += X[i][j]
        if not (i == n-1 and j == n-1):
            con4 += ','
        else:
            con4 += ';'

# распечатать
print obj
for i in range(n):
    print con1[i]
print con2
for i in range(len(con3)):
    print con3[i]
print con4
```

Листинг 11.7 ❖ Часть LP-файла для простой сети

```
MIN: 0*x_0_0+3*x_0_1+13*x_0_2+5*x_0_3+...+11*x_7_6+0*x_7_7;
x_0_0+x_0_1+x_0_2+x_0_3+x_0_4+x_0_5+x_0_6+x_0_7=1;
...
x_0_0+x_1_1+x_2_2+x_3_3+x_4_4+x_5_5+x_6_6+x_7_7=2;
x_0_1-x_1_1<=0;
...
x_7_6-x_6_6<=0;
BIN x_0_0,x_0_1,x_0_2,x_0_3,...,x_7_6,x_7_7;
```

В листинге 11.7 представлена очень малая часть результата выполнения программы для нашей простой сети. Две приведенные ниже команды показывают, как запускаются программа создания LP-файла и программа `lp_solve`, которая решает задачу, описанную в этом LP-файле. Из результата видно, что при $p = 2$ (значение по умолчанию в нашей Python-программе) оптимально будет разместить предприятия в узлах 0 и 5, т. к. `x_0_0` и `x_5_5` равны 1. Также показано, какое предприятие должно обслуживать каждый узел. Например, узел 2 должен обслуживаться предприятием в узле 5, т. к. `x_2_5 = 1`. При взгляде на рис. 11.2 становится понятно, что такое решение разумно, потому что узел 2 ближе к узлу 5, чем узел 0.

```
$ python pmed_lpformat.py network-links > network-simple.lp
$ lp_solve -ia -S0 network-simple.lp
Value of objective function: 40.00000000

Actual values of the variables:
x_0_0                           1
x_1_0                           1
x_2_5                           1
x_3_0                           1
x_4_5                           1
x_5_5                           1
x_6_5                           1
x_7_5                           1
```

Теперь протестируем программу в листинге 11.6 на задаче из набора эталонных тестов – 40 задач о p-медиане из библиотеки OR-Library[1]. Каждая из этих задач хранится в текстовом файле, очень похожем на то, что мы использовали в листинге 10.1, но первая строка содержит три числа, а не одно, как у нас. Однако наша Python-программа все равно будет работать, потому что она читает только первое число, равное общему количеству узлов в сети. В первой из 40 задач имеется 100 узлов, а $p = 5$. Предположим, что файл для этой задачи называется pmed1.orlib. Ниже показано, как запускается программа, и ее результат – правильное значение целевой функции. Заметим, что в командной строке необходимо указать, что нумерация начинается не с нуля, а значение p равно 5, только тогда можно будет сравнить полученный результат с известным значением целевой функции.

```
$ python pmed_lpformat.py pmed1.orlib False 5 > pmed1.lp
$ lp_solve -S1 pmed1.lp
Value of objective function: 5819.00000000
```

11.3. Примечания

Согласно работе Sylvester (1860), именно Пирс разработал первый алгоритм решения задачи о минимальной ограничивающей окружности. Впослед-

[1] http://www.brunel.ac.uk/~mastjjb/jeb/info.html. Прямая ссылка на все 40 задач о p-медиане имеет вид http://people.brunel.ac.uk/~mastjjb/jeb/orlib/pmedinfo.html.

ствии Кристал в работе Chrystal (1885) формализовал алгоритм Пирса. Недавний теоретический анализ показал, что временная сложность алгоритма Кристала–Пирса составляет $O(n^2)$ (Chakraborty and Chaudhuri, 1981; Plastria, 2002). Стратегия инициализации для алгоритма Кристала–Пирса предложена в работе Chakraborty and Chaudhuri (1981). Алгоритм Эльзинги–Хирна описан в работе Elzinga and Hearn (1972), а его анализ, проведенный в работе Drezner and Shelah (1987), показал, что для завершения алгоритма требуется $O(n^2)$ итераций, а временная сложность составляет $\Omega(n^2)$. Ожидаемая сложность алгоритма Вельцля (Welzl, 1991) составляет $O(n)$ (см. также de Berg et al., 1998, стр. 89).

11.4. Упражнения

1. Функция `test` в коде алгоритма Кристала–Пирса выводит точки, в которых каждая координата является числом с плавающей точкой с точностью до одного десятичного знака после запятой. Из-за этого приведенная в тексте распечатка не помогает нарисовать точки и понять, в чем причина неоптимальности начальной окружности. Модифицируйте код, так чтобы при создании тестовых точек использовались только целые числа. Затем прогоните программу много раз, пока не окажется, что первая ограничивающая окружность не оптимальна. Нарисуйте эти точки и обсудите, почему начальная окружность не оптимальна.
2. В примере на рис. 11.1 при иллюстрации поиска показан тупоугольный треугольник. Поскольку на каждой итерации выбирается случайная точка вне окружности, попробуйте нарисовать ситуацию, когда в процессе поиска оптимальной ограничивающей окружности возникает хотя бы один остроугольный треугольник.
3. Мы обсудили два случая, возникающих в ограничении (11.4) в формулировке задачи о p-медиане. А какие еще два случая призван гарантировать этот набор ограничений? Как эти ограничения обеспечивают, что любой спрос будет удовлетворяться ближайшим предприятием?
4. У задачи о 1-центре, рассмотренной в этой главе, есть вариант с несколькими предприятиями, который называется задачей о p центрах. В ее непрерывной постановке требуется разместить p предприятий, так чтобы каждое находилось в центре множества достижимых из него точек. Мы хотим минимизировать максимальное расстояние от центра до любой из назначенных ему точек. Такого рода задачи называются минимаксными, потому что цель состоит в минимизации максимального расстояния. В дискретной постановке задачи о p центрах задается множество потенциальных мест размещения предприятий. Чтобы можно было вычислять расстояния, обозначим d_{ij} расстояние между вершинами i и j, где i принадлежит множеству I мест нахождения клиентов, формирующих спрос, а j – множеству J потенциальных мест размещения предприятий. В состав параметров решения включаются: y_j, равное 1, если выбрана вершина j, и 0 в противном случае; x_{ij}, равное 1, если спрос в точке i обслуживается пред-

приятием в точке j; и z – минимальное расстояние между узлом и ближайшим к нему предприятием. Тогда задача формулируется следующим образом:

$$\begin{aligned} &\min z, \\ &\text{при условии } \sum_j x_{ij} = 1 && \forall i, \\ &\sum_j y_j = p, \\ &x_{ij} \leqslant y_j && \forall i, j, \\ &z \geq \sum_j d_{ij} x_{ij} && \forall i, \\ &y_j = 0,1 && \forall j, \\ &x_{ij} = 0,1 && \forall i, j. \end{aligned}$$

Объясните смысл каждой строки в этой постановке и напишите на Python программу создания LP-файла для задачи о p центрах в случае нашей простой сети.

5. Мы рассматривали только задачи о медиане и центре, но существует большое семейство задач размещения, которые называются задачами о покрытии и имеют много применений. Пусть имеется множество клиентов, обозначенных буквой i. Требуется разместить предприятия в некотором множестве потенциальных точек, обозначенных буквой j. Стоимость размещения предприятия в точке j равна f_j. Наша цель – сделать так, чтобы любой клиент был покрыт хотя бы одним предприятием. Слово «покрыт» может трактоваться по-разному. Например, если мы возводим наблюдательную вышку в точке j и клиент в точке i виден с этой вышки, то мы говорим, что i покрыт j. Будем обозначать такое отношение покрытия a_{ij}, считая, что этот параметр равен 1, если предприятие в точке j может удовлетворить спрос со стороны клиента i, и 0 в противном случае. Обозначим N_i множество индексов потенциальных предприятий, которые могут покрыть клиента i. Формально $N_i = \{j | a_{ij} = 1\}$. Параметр решения x_j равен 1, если предприятие размещено в точке j, иначе 0. В этих обозначениях задача о покрытии формулируется следующим образом:

$$\begin{aligned} &\min \sum_{j=1} f_j x_j, \\ &\text{при условии } \sum_{j \in N_i} x_j \geq 1 && \forall i, \\ &xj = 0,1 && \forall j. \end{aligned}$$

Напишите на Python программу, которая создает файл с описанием задачи о покрытии множества в формате LP, который можно подать на вход решателю lp_solve.

Глава 12

Эвристические алгоритмы поиска

– Скажите, пожалуйста, куда мне отсюда идти?
– А куда ты хочешь попасть? – ответил Кот.
– Мне все равно… – сказала Алиса.
– Тогда все равно, куда и идти, – заметил Кот.
– …только бы попасть *куда-нибудь*, – пояснила Алиса.
– Куда-нибудь ты обязательно попадешь, – сказал Кот. – Нужно только достаточно долго идти.

Льюис Кэрролл
«Приключения Алисы в Стране чудес» (перевод Н. Демуровой)

В этой главе в фокусе нашего внимания будут эвристические методы – достаточно эффективные, но не гарантирующие нахождение оптимального решения. Мы будем говорить о двух видах эвристик: для решения конкретных задач и для решения широкого круга общих задач. Последние называются метаэвристиками, поскольку пригодны не для одного какого-то типа задач, а для задач многих видов. Часто эти алгоритмы называют алгоритмами поиска, потому что их цель – поиск в пространстве решений в надежде найти наилучшее.

12.1. Жадные алгоритмы

Мы можем строить решение задачи с нуля. Именно чтобы найти решение задачи о p-медиане в сети, можно сначала включить какой-нибудь узел предприятия в частичное решение и продолжать добавление узлов, пока не будет найдено полное решение. При этом должен существовать какой-то критерий, позволяющий решить, какой узел добавлять на каждой итерации. При каждом добавлении узла в частичное решение сумма расстояний до ближайшего предприятия по всем узлам должна уменьшаться. Таким образом, можно сформулировать эвристическое правило: добавлять тот узел, для которого сумма расстояний уменьшается больше всего. Можно вместо этого начать с «суперрешения», когда выбраны вообще все узлы, так что целевая функция оказывается равна нулю. А затем удалять узлы, пока не останется пригодное решение (содержащее p узлов).

Такие алгоритмы называются жадными, потому что на каждом шаге мы выбираем то, что представляется наилучшим в данный момент. В общем случае при проектировании жадного алгоритма решения задачи мы инкрементно модифицируем *частичное* решение, пока не получим полное. На протяжении всего процесса нужно знать, из каких компонентов состоит решение и как выбирать компонент, приближающий нас к полному решению. Цель жадного алгоритма – получить *одно* решение. Поэтому мы должны следить за тем, чтобы алгоритм порождал только пригодные решения. По ходу дела следует оценивать частичные решения, вычисляя целевую функцию.

Программа в листинге 12.1 реализует жадный алгоритм для задачи о *p*-медиане, применяя принцип добавления. Функция `evaluate` (строка 9) возвращает сумму расстояний до ближайшего предприятия по всем узлам, получив решение в параметре `median`. Здесь мы снова используем алгоритм нахождения кратчайших путей между всеми парами узлов для вычисления матрицы расстояний по файлу, содержащему описание сети. Основная идея этого алгоритма – начать с пустого списка узлов (строка 34) и на каждой итерации цикла `for` проверять, какой узел следует добавить в решение, чтобы суммарное расстояние было минимальным (строка 39). Найдя такой узел, мы удаляем его из списка кандидатов (строка 44), чтобы не рассматривать на следующих итерациях. Одновременно узел добавляется в текущее решение (строка 45).

Листинг 12.1 ❖ Жадный алгоритм решения задачи о *p*-медиане (pmed_greedy.py)

```
 1 import sys
 2 from string import atoi
 3 sys.path.append('../networks')
 4 from network2listmatrix import network2distancematrix
 5 from allpairdist import allpairs
 6
 7 INF = float('inf')
 8
 9 def evaluate(dist, median, n):
10     sumdist = 0.0
11     p = len(median)
12     for i in range(n):
13         dist0 = INF
14         for j in range(p):
15             if dist[i][median[j]] < dist0:
16                 dist0 = dist[i][median[j]]
17         sumdist += dist0
18     return sumdist
19
20 if len(sys.argv) <=1:
21     print sys.argv[0], "filename [ [True|False] p]"
22     sys.exit()
23
24 zerobased = True
25 p = 2
26 if len(sys.argv)>=3 and sys.argv[2]=="False":
```

```
    zerobased = False
if len(sys.argv)>=4:
    p = atoi(sys.argv[3])

a = network2distancematrix(sys.argv[1], zerobased)
allpairs(a)
n = len(a)
median = []
candidates = [i for i in range(n)]
for j in range(p):
    dmin = INF
    imin = -1
    for i in candidates:
        d = evaluate(a, median+[i], n)
        if d < dmin:
            dmin = d
            imin = i
    candidates.remove(imin)
    median.append(imin)

print median, dmin
```

Мы протестировали жадный алгоритм на двух задачах о *p*-медиане, упомянутых в предыдущей главе. Результат показывает, что решение ищется быстро, хотя оно и не оптимально. Итак, должно быть понятно, что жадные алгоритмы просты и работают быстро. Зачастую найденное ими решение не оптимально, но зато оно всегда пригодно, и на поиск уходит немного времени.

```
$ python pmed_greedy.py ../data/network-links
[2, 0] 49.0
$ python pmed_greedy.py ../data/pmed1.orlib False 5
[6, 12, 3, 90, 98] 5891.0
```

12.2. Алгоритм обмена вершин

Решения обеих задач о *p*-медиане, найденные жадным алгоритмом в предыдущем разделе, не оптимальны. А сейчас мы познакомимся с эвристическим алгоритмом, который спроектирован специально для задачи о *p*-медиане. Это алгоритм обмена вершин, который часто называют алгоритмом Тейца–Барта по именам авторов, предложивших его. Идея простая: начинаем со случайного решения и продолжаем обменивать одну из выбранных вершин с невыбранной, пока получается улучшать решение. В листинге 12.2 эта логика оформлена в виде псевдокода.

Листинг 12.2 ❖ Алгоритм обмена вершин для задачи о *p*-медианах

```
Пусть s – случайное решение
Повторять
    Для каждой невыбранной вершины i
        Вычислить выигрыш от замены выбранной вершины на i
```

```
5     Пусть s1 – решение с наилучшим выигрышем
6     Если выигрыш s1 > 0, то s = s1
7 Пока выигрыш s1 не станет < 0
```

Программа в листинге 12.3 реализует алгоритм обмена вершин. Решение задачи представлено строкой *p* целых чисел – индексов выбранных вершин. Основная часть алгоритма – проверка потенциального решения на предмет улучшения значения целевой функции. Для создания потенциального решения мы вставляем новую, ранее не выбранную вершину в текущее решение и смотрим, какую из прежних вершин можно заменить новой (строка 4 псевдокода). Для поиска наилучшей замены нужно просто вычислить для каждой вершины ближайшую, а затем просуммировать расстояния. Это реализуемо, но неэффективно.

Ниже описан алгоритм, который не пересчитывает все вообще. Он реализован в функции `findout` (строка 11). Предполагая, что дано решение, содержащее *p* выбранных для размещения предприятий вершин, мы вместо пересчета используем два списка. Для каждой вершины в первом списке (`d1`) хранятся индексы предприятий, ближайших к ней, а во втором – индексы вторых по удаленности от нее. При вставке новой вершины в текущее решение мы должны изъять одну из существующих вершин и хотим сделать это так, чтобы получить максимальный выигрыш в терминах значения целевой функции. Если добиться выигрыша невозможно, то мы хотим, чтобы потеря целевой функции была минимальной. Мы перебираем все вершины и проверяем возможность замены. Проверяемая на целесообразность вставки вершина названа `fi`. Для каждой вершины `i` (строка 19) если расстояние между `i` и `fi` меньше расстояния между `i` и текущей ближайшей к ней вершиной, то замена целесообразна, и мы увеличиваем текущий выигрыш (строка 21). В противном случае мы смотрим, какой будет потеря, если придется удалить текущее ближайшее предприятие, эта потеря зависит от того, какая вершина теперь ближе к `i`: вторая по удаленности или вновь вставляемая (строка 24). Мы подсчитываем потери вследствие удаления каждой вершины, где размещено предприятие (строка 24), выбираем вершину, для которой потеря минимальна (строка 27), и вычисляем полный выигрыш (строка 31).

Этот алгоритм очень эффективен, потому что нам удалось исключить необходимость пересчета всех вершин при вставке новой вершины в решение. Это типичный пример компромисса между памятью и временем: мы потребляем больше памяти, но выигрываем во времени вычислений. Недостаток в том, что приходится поддерживать списки `d1` и `d2`. Это делает функция `update_assignment` (строка 34), где для каждой вершины проверяется ближайшая к ней вершина и следующая по удаленности.

Функция `next` (строка 59) ищет наилучшую замену, проверяя все вершины, которые можно вставить в текущее решение (строка 73). Если найдено лучшее решение (строка 82), то мы производим замену и выполняем обновление и новое назначение. Первое возвращенное значение – признак, сообщающий, было найдено лучшее решение или нет.

Функция `teitz_bart` (строка 88) управляет всем итеративным алгоритмом замены вершин, описанным в псевдокоде. Первым делом создается начальное решение, включающее *p* случайно выбранных вершин из множества

N вершин сети (строка 96). Начальное решение обсчитывается в функции `update_assignment` (строка 99). Затем в цикле `while` проверяется, может ли обмен вершин улучшить решение.

Листинг 12.3 ❖ Алгоритм обмена вершин для решения задачи о p-медиане (teitz_bart.py)

```
 1 from bisect import bisect_left
 2 from string import atoi, split
 3 import random
 4 import sys
 5 sys.path.append('../networks')
 6 from network2listmatrix import network2distancematrix
 7 from allpairdist import allpairs
 8
 9 INF = float('inf')
10
11 def findout(median, fi, dist, d1, d2, N):
12     """
13     Получив кандидата на вставку (fi), находит в текущем решении
14     лучшего кандидата на замену или удаление
15     (fr).
16     При этом значения в median, d1 и d2 не изменяются.
17     """
18     w = 0.0
19     v = [0.0 for i in range(N)]
20     for i in range(N):
21         if dist[i][fi] < dist[i][d1[i]]:   # выигрыш
22             w += dist[i][d1[i]] - dist[i][fi]
23         else:                              # потеря
24             v[d1[i]] += min(dist[i][fi],
25                             dist[i][d2[i]]) - dist[i][d1[i]]
26     fmin = INF
27     fr = 0
28     for i in median:
29         if v[i] < fmin:
30             fmin = v[i]
31             fr = i
32     fmin = w-fmin
33   return fmin, fr                          # выигрыш и подлежащая замене вершина
34
35 def update_assignment(dist, median, d1, d2, p, N):
36     """
37     Зная median, обновляем d1 и d2, так чтобы в d1 хранились
38     ближайшие предприятия, а в d2 – вторые по удаленности
39     """
40     dist1, dist2 = 0.0, 0.0
41     node1, node2 = -1, -1
42     for i in range(N):
43         dist1, dist2 = INF, INF
44         for j in range(p):
45             if dist[i][median[j]] < dist1:
```

```
                dist2 = dist1
                node2 = node1
                dist1 = dist[i][median[j]]
                node1 = median[j]
            elif dist[i][median[j]] < dist2:
                dist2 = dist[i][median[j]]
                node2 = median[j]
        d1[i] = node1
        d2[i] = node2
    dist1 = 0
    for i in range(N):
        dist1 += dist[i][d1[i]]
    return dist1

def next(dist, median, d1, d2, p, N):
    """
    Вход
      dist: матрица рассояний
      median: список индексов выбранных вершин
      d1: список ближайших предприятий для каждой вершины
      d2: список вторых по удаленности предприятий
      p: сколько предприятий нужно разместить
      N: число вершин в сети
    Выход
      r: суммарное рассояние
      median: список индексов выбранных вершин
    """
    bestgain = -INF
    for i in range(N):
        gain, fr1 = findout(median, i, dist, d1, d2, N)
        if i in median:
            continue
        if gain>bestgain:
            bestgain = gain
            fr = fr1
            fi = i
    r = 0
    if bestgain > 0:
        i = median.index(fr)
        median[i] = fi
        r = update_assignment(dist, median, d1, d2, p, N)
    return bestgain>0, r, fr, fi

def teitz_bart(dist, p, verbose=False):
    """
    Вход
      dist: матрица рассояний
      p: сколько предприятий нужно разместить
      verbose: печатать ли промежуточные результаты
    """
    N = len(dist)
    median = random.sample(range(N), p)
```

```
 98     d1 = [-1 for i in range(N)]
 99     d2 = [-1 for i in range(N)]
100     r = update_assignment(dist, median, d1, d2, p , N)
101     if verbose: print r
102     while True:
103         result = next(dist, median, d1, d2, p, N)
104         if result[0]:
105             r = result[1]
106             if verbose: print r
107         else:
108             break
109     return r, median
110
111 if __name__ == "__main__":
112     print 'Задача: простая сеть'
113     a = network2distancematrix('../data/network-links', True)
114     allpairs(a)
115     teitz_bart(a, 2, True)
116     print 'Задача: pmed1 в OR-lib'
117     a = network2distancematrix('../data/orlib/pmed1.orlib',
118                                False)
119     allpairs(a)
120     teitz_bart(a, 5, True)
```

Как и в случае жадного алгоритма, протестируем алгоритм обмена вершин на простой сети (строка 114) и на первой задаче о *p*-медиане из библиотеки OR-Library (строка 119). Приведенный ниже результат показывает, что мы смогли найти оптимальные решения обеих задач. Мы запускали программу многократно, и всегда она находила оптимальные решения в обоих случаях.

```
Задача: простая сеть
53
43
40
Задача: pmed1 в OR-lib
11030
7971
6598
6024
5847
5821
5819
```

А как насчет других задач? Проверим. На этот раз мы используем все 40 задач из библиотеки OR-Library и для каждой прогоним тест только один раз (для иллюстрации). А чтобы повысить эффективность многократного прогона теста, лучше преобразовать файл описания сети в матрицу расстояний единожды. Заранее сохраним все матрицы расстояний и будем загружать их при каждом запуске программы. Это легко сделать с помощью функции `list2distancematrix`, которую мы обсуждали в главе 10 в контексте анализа сетей. Программа в листинге 12.4 включает функцию `load_distance_matrix`,

которая загружает матрицу расстояний из файла с указанным именем. Описания всех 40 задач хранятся в переменных, начиная со строки 21. Функция `testall` один раз прогоняет тест для каждой задачи, а функция `testpmed` прогоняет тест на указанный задаче по умолчанию 20 раз.

Листинг 12.4 ❖ Тестирование алгоритма обмена вершин на задачах из библиотеки OR-Library (test_orlib_pmed.py)

```
 1 import time
 2 from teitz_bart import *
 3
 4 def load_distance_matrix(fname):
 5     # загрузить матрицу расстояний
 6     f = open(fname)
 7     l = f.readline()
 8     N = atoi(split(l.strip())[0])    # получить число узлов
 9     dist=[[0]*N for x in xrange(N)]  # заполнить 2-мерный список значениями INF
10     l = f.readline()
11     row = 0
12     for l in f:
13         dline = split(l.strip())
14         if len(dline)==N:
15             for i in range(N):
16                 dist[row][i] = atoi(dline[i])
17         row += 1
18     f.close()
19     return dist
20
21 N = [100, 100, 100, 100, 100, 200, 200, 200, 200, 200,
22      300, 300, 300, 300, 300, 400, 400, 400, 400, 400,
23      500, 500, 500, 500, 500, 600, 600, 600, 600, 600,
24      700, 700, 700, 700, 800, 800, 800, 900, 900, 900]
25 pp = [5, 10, 10, 20, 33, 5, 10, 20, 40, 67,
26       5, 10, 30, 60, 100, 6, 10, 40, 80, 133,
27       5, 10, 60, 100, 167, 5, 10, 60, 120, 200,
28       5, 10, 70, 140, 5, 10, 80, 5, 10, 90]
29 known = [5819, 4093, 4250, 3034, 1355,
30          7824, 5631, 4445, 2734, 1255,
31          7696, 6634, 4374, 2968, 1729,
32          8162, 6999, 4809, 2845, 1789,
33          9138, 8579, 4619, 2961, 1828,
34          9917, 8307, 4498, 3033, 1989,
35          10086, 9297, 4700, 3013, 10400,
36          9934, 5057, 11060, 9423, 5128]
37
38 def testall():
39     for i in range(len(N)):
40         fn = format("../data/orlib/pmed%d.distmatrix"%(i+1))
41         dist = load_distance_matrix(fn)
42         t1 = time.time()
43         r = teitz_bart(dist, pp[i])[0]
44         t2 = time.time()
```

```
45          print format("pmed%d:"%(i+1)),\
46             N[i], pp[i], r, known[i], t2-t1,
47          if r==known[i]: print "(*)"
48          else: print
49
50 def testpmed(i, numtest=20):
51      fn = format("../data/orlib/pmed%d.distmatrix"%(i+1))
52      dist = load_distance_matrix(fn)
53      t1 = time.time()
54      sumdiff = 0
55      nummiss = 0
56      for j in range(numtest):
57          r = teitz_bart(dist, pp[i], verbose=True)[0]
58          print r,
59          if r==known[i]: print "(*)"
60          else:
61              print
62              sumdiff += r - known[i]
63              nummiss += 1
64      t2 = time.time()
65      print "Время:", (t2-t1)/numtest
66      print "Найдено:", numtest-nummiss
67      if nummiss:
68          print "Средняя разность:", float(sumdiff)/nummiss
69
70      if __name__ == "__main__":
71          testall()
72          testpmed(16)
```

Результаты выполнения функции `testall` приведены ниже. Видно, что алгоритм обмена вершин действительно может находить оптимальные решения многих задач: в этом эксперименте они были найдены в 14 из 40 случаев. Интересно отметить, что при разных значениях p для одного и того же n алгоритм чаще находит оптимальные решения, когда p относительно мало. Это «более простые» задачи при заданной конфигурации (значении n), поскольку в них меньше возможных комбинаций. В каждой строке результата мы печатаем имя задачи (например, pmed1), общее число вершин (n), число подлежащих размещению предприятий (p), лучшее из найденных значений целевой функции, известное оптимальное решение и время работы (в секундах). Звездочка означает, что найдено оптимальное решение.

```
pmed1: 100 5 5819 5819 0.0466809272766 (*)
pmed2: 100 10 4102 4093 0.0773859024048
pmed3: 100 10 4250 4250 0.0820319652557 (*)
pmed4: 100 20 3049 3034 0.122255086899
pmed5: 100 33 1355 1355 0.150933027267 (*)
pmed6: 200 5 7824 7824 0.213330984116 (*)
pmed7: 200 10 5639 5631 0.414077043533
pmed8: 200 20 4454 4445 0.76794219017
pmed9: 200 40 2764 2734 1.02096009254
pmed10: 200 67 1265 1255 1.57539701462
```

```
pmed11: 300 5 7696 7696 0.4705991745 (*)
pmed12: 300 10 6634 6634 1.02756404877 (*)
pmed13: 300 30 4374 4374 1.90632295609 (*)
pmed14: 300 60 2971 2968 4.46003508568
pmed15: 300 100 1737 1729 5.15211892128
pmed16: 400 6 7788 8162 1.28738808632
pmed17: 400 10 7014 6999 1.3513071537
pmed18: 400 40 4813 4809 5.61389303207
pmed19: 400 80 2860 2845 8.77476096153
pmed20: 400 133 1805 1789 9.67293000221
pmed21: 500 5 9138 9138 1.22582006454 (*)
pmed22: 500 10 8579 8579 2.59184718132 (*)
pmed23: 500 60 4212 4619 13.2033700943
pmed24: 500 100 2962 2961 17.4130480289
pmed25: 500 167 1844 1828 23.4592969418
pmed26: 600 5 9917 9917 1.53462600708 (*)
pmed27: 600 10 8307 8307 3.74222993851 (*)
pmed28: 600 60 4508 4498 15.5038609505
pmed29: 600 120 3041 3033 24.4620049
pmed30: 600 200 2012 1989 32.912113905
pmed31: 700 5 10087 10086 2.17640995979
pmed32: 700 10 9301 9297 4.40583491325
pmed33: 700 70 4715 4700 24.5156989098
pmed34: 700 140 3025 3013 41.5150740147
pmed35: 800 5 10400 10400 3.16665506363 (*)
pmed36: 800 10 9951 9934 7.37093496323
pmed37: 800 80 5067 5057 38.7697281837
pmed38: 900 5 11060 11060 5.15535211563 (*)
pmed39: 900 10 9423 9423 7.04349589348 (*)
pmed40: 900 90 5134 5128 56.506018877
```

Возникает вопрос: может ли этот алгоритм найти оптимальное решение любой задачи, пусть даже для этого потребуется много попыток? Очевидно, что существует много задач, для которых алгоритм не может найти оптимальное решение с первого раза. Например, в приведенном выше тесте оптимальное значение целевой функции в задаче pmed17 равно 6999, но алгоритм вернул только 7014. Есть ли надежда, что мы все-таки сможем найти для нее оптимальное решение? Попробуем прогнать алгоритм еще несколько раз, скажем 20. Это делает функция `testpmed`, которой нужно указать номер задачи (заметим, что нумерация начинается с нуля). Оказывается, что алгоритм находит оптимальное решение в 10 из 20 попыток:

```
6999 (*)
6999 (*)
7037
6999 (*)
6999 (*)
7003
7010
6999 (*)
6999 (*)
6999 (*)
```

```
7014
7037
6999 (*)
7014
7014
6999 (*)
6999 (*)
7003
7014
7003
Время: 1.27441074848
Найдено: 10
Средняя разность: 15.9
```

12.3. Имитация отжига

Алгоритм обмена вершин в задаче о p-медиане «жадный»[1], потому что на каждой итерации рассматривает лучшую замену и принимает только те, которые улучшают решение. Из-за этого процесс поиска легко может застрять в локальном оптимуме, где все возможные решения хуже текущего. Но что, если дать худшим решениям шанс? Люди действуют жадно, но при этом могут вырабатывать стратегию. Например, в шахматах мы часто жертвуем малоценную фигуру, чтобы в конечном итоге поставить мат сопернику. Такой подход порождает широкий спектр новых эвристических методов.

В этом разделе мы рассмотрим еще один эвристический метод – имитацию отжига. Подобно другим эвристическим методам, ему необходима стратегия создания множества потенциальных решений, рассматриваемых на каждой итерации. Эти решения называются соседями текущего, потому что обычно они создаются с помощью операций, изменяющих текущее решение. После того как соседи созданы, мы применяем стратегию выбора одного из них. Можно почти всегда выбирать лучшего среди соседей, но время от времени отдавать предпочтение случайному решению. Затем нужно решить, примем ли мы выбранное решение в качестве следующего. Это зависит от того, как обрабатываются худшие решения и как алгоритм избегает локальных «ям», пытаясь найти оптимальные решения. Может случиться, особенно в начале поиска, что алгоритм примет решение, которое хуже текущего. Обозначим Δ разность значений целевой функции для нового и текущего решений. Пусть требуется минимизировать целевую функцию, а отрицательное Δ означает, что новое решение лучше текущего и будет принято на данной итерации. Но и у худшего решения есть шанс оказаться принятым, и эта вероятность вычисляется согласно алгоритму Метрополиса:

$$\Pr(\Delta, T) = \begin{cases} 1, & \text{если } \Delta < 0 \\ e^{-\Delta/T} & \text{в противном случае} \end{cases},$$

[1] Не совсем точно называть этот алгоритм жадным, потому что данный термин зарезервирован для другого класса алгоритмов.

где T – параметр, который часто называют температурой. В процессе поиска значение T постепенно уменьшается. Ясно, что чем больше T, тем выше вероятность принять худшие решения. Мы называем уменьшение температуры отжигом по аналогии с одноименным физическим процессом. Стратегия такого уменьшения называется методом охлаждения. Простой метод охлаждения заключается в том, чтобы умножать T на коэффициент r ($0 < r < 1$) в конце каждой итерации. Вычислив вероятность принять решение, мы производим случайную выборку из равномерного распределения в интервале от 0 до 1. Если выбрано число, меньшее вероятности, то новое решение принимается, в противном случае отвергается, и процесс останавливается.

В листинге 12.5 приведен псевдокод общей процедуры имитации отжига. Мы предполагаем, что значение целевой функции для решения s равно $f(s)$, а функция $R(0,1)$ возвращает случайное число, выбранное из равномерного распределения. Алгоритм продолжает принимать новые решения, пока не окажется выполнено условие в строке 9.

Листинг 12.5 ❖ Общий алгоритм имитации отжига

```
 1 Обозначим s начальное решение задачи
 2 Обозначим f(s) значение целевой функции для s
 3 Обозначим T начальное значение
 4 Повторять
 5   Найти множество соседей s
 6   Пусть s1 – наилучшее решение среди соседей
 7   Δ = f(s1) – f(s)
 8 P = Pr(Δ, T)
 9 Если P > R(0, 1)
10   s = s1
11   T = r * T
12 Иначе
13   Остановиться и сообщить о лучшем найденном решении
```

Теперь реализуем алгоритм имитации отжига для задачи о p-медиане (листинг 12.6). Главная функция `simulated_annealing` (строка 117) управляет логикой процедуры в целом, как описано в псевдокоде. Как и в алгоритме обмена вершин, мы инициализируем решение, представив его множеством случайно выбранных индексов (строка 125). Используется та же функция `update_assignment`, что и раньше (строка 126). Мы не отказываемся от идеи хранить списки ближайших и вторых по удаленности вершин для каждой вершины, т. к. это позволяет более эффективно оценивать результат обмена вершин. Начальная температура задается равной 100 (строка 132). В цикле `while` (строка 133) мы вызываем функцию `next` (строка 134), которая находит наилучшее решение для следующей итерации.

Функция `next` ищет наилучшее решение среди соседей текущего и проверяет, следует ли его принять. Мы реализовали два способа создания соседей данного решения, поэтому можем выбирать среди них. Способ выбора задается параметром `neighbormethod` функции `next`. Первый вариант – создать соседа путем замены одной из выбранных вершин на другую, находящуюся от исходной не дальше, чем величина параметра `dthreshold`. Мы создаем p

соседей с помощью функции `bestGeoNeighbor` (строка 64), которая возвращает лучшего соседа для последующей проверки в функции `next` (строка 89). Заметим, что функция `bestGeoNeighbor` возвращает не само решение, а новую вершину (`fi`), которой нужно заменить текущую выбранную вершину (`fr`). Поскольку этот способ создания соседей основан на замене вершины в существующем решении, нет необходимости пересчитывать все назначения, а достаточно воспользоваться информацией о ближайших и вторых по удаленности предприятиях (строка 74), т. е. тем же механизмом, что в функции `findout`, рассмотренной ранее. Второй вариант – случайным образом сгенерировать *p* новых решений и вернуть лучшее из них – реализован в функции `bestRandomNeighbor` (строка 50). Поскольку теперь каждое соседнее решение является новым, придется вычислять целевую функцию заново (строка 55), а хранить списки ближайших и вторых по удаленности не нужно.

В нашей реализации алгоритма имитации отжига стоит обратить внимание еще на несколько моментов. Во-первых, мы храним в переменной `accepted_same` (строка 131), сколько раз одно и то же лучшее решение было принято после принятия худшего решения (строка 141). Во-вторых, мы принимаем лишь примерно две трети замен для каждой выбранной вершины (строка 72). То и другое призвано предотвратить ненужные осцилляции между лучшим и вторым по качеству найденным решением. Мы знаем, что по самой своей природе алгоритм имитации отжига может принимать решения, худшие, чем лучшее из найденных. Предположим, что алгоритм уже нашел оптимальное решение, так что любая альтернатива будет хуже. Если применить жадный подход и выбирать только лучшего из соседей, то всегда будет выбираться одно и то же худшее решение, а затем на следующей итерации будет происходить возврат к одному и тому же лучшему решению. Разумеется, то же самое может происходить и для локально оптимального решения. Поэтому мы произвольно выбрали пороговое значение – три – числа возвратов к одному и тому же лучшему решению. Это также позволяет закончить алгоритм раньше, потому что в противном случае итерации продолжались бы, пока вероятность принятия решения не стала бы достаточно малой. А разрешая выбирать только две трети замен, мы еще сильнее снижаем шансы возврата к одному и тому же решению.

Листинг 12.6 ❖ Алгоритм имитации отжига (simulated_annealing.py)

```
 1 from bisect import bisect_left
 2 from string import atoi, split
 3 import math
 4 import random
 5 from copy import deepcopy
 6 from teitz_bart import update_assignment
 7 import sys
 8 sys.path.append('../networks')
 9 from network2listmatrix import network2distancematrix
10 from allpairdist import allpairs
11
12 INF = float('inf')
13
```

```
def evaluate(dist, median, p, N):
    sumdist = 0.0
    for i in range(N):
        dist0 = INF
        for j in range(p):
            if dist[i][median[j]] < dist0:
                dist0 = dist[i][median[j]]
        sumdist += dist0
    return sumdist

# проверить замену fr на fi в медиане, не пересчитывая
# все узлы
def test_replacement(fi, fr, dist, d1, d2, p, N):
    total = 0.0
    for i in range(N):
        if dist[i][fi]<dist[i][d1[i]]:
            total += dist[i][fi]
        else:
            if fr==d1[i]:
                if dist[i][fi]<dist[i][d2[i]]:
                    total += dist[i][fi]
                else:
                    total += dist[i][d2[i]]
            else:
                total += dist[i][d1[i]]
    return total

def get_new_median(N, p, median):
    samples = []
    for i in range(p):
        s = -1
        while s in median or s in samples:
            s = random.sample(range(N), 1)[0]
        samples.append(s)
    return samples

def bestRandomNeighbor(median, dist, N, p):
    candidates = []
    r_mins = []
    for j in range(p):
        candidates.append(get_new_median(N, p, median))
        r_mins.append(evaluate(dist, candidates[j], p , N))
    r_min = INF
    i_min = -1
    for j in range(p):
        if r_mins[j] < r_min:
            r_min = r_mins[j]
            i_min = j
    return r_min, candidates[i_min]

def bestGeoNeighbor(median, dist, d1, d2, N, p, dthreshold):
    r_min = INF
```

```
    for j in range(p):
        r_min_temp = INF
        fi_temp = -1
        for i in range(N): # пытаемся заменить j на i
            if dist[median[j]][i]>dthreshold or i in median:
                continue
            if random.random() > 0.67:
                continue
            r1 = test_replacement(i, median[j], dist,
                                  d1, d2, p, N)
            if r1 < r_min_temp:
                r_min_temp = r1
                fi_temp = i
        if r_min_temp < r_min:
            r_min = r_min_temp
            fi = fi_temp
            fr = j
    return r_min, fi, fr

def next(r, T, median, dist, d1, d2, p, N,
            dthreshold, neighbormethod):
    r1 = r
    if neighbormethod == 0:
        r_min, fi, fr = bestGeoNeighbor(median, dist, d1,
                                        d2, N, p, dthreshold)
        test = acceptable(10*(r_min-r)/r, T)
        if test[0] > 0:
            median[fr] = fi
            r1 = update_assignment(dist, median,d1,d2,p,N)
        return test[0], r1, median
    else:
        r_min, candidate = bestRandomNeighbor(median,
                                              dist, N, p)
        test = acceptable(10*(r_min-r)/r, T)
        if test[0] > 0:
            median = candidate
            r1 = update_assignment(dist,
                                   median, d1, d2, p, N)
        return test[0], r1, median

def acceptable(delta, T):
    if delta<0:                    # лучшее решение
        return 1, 1
    if delta==0:                   # то же решение, ничего не изменилось
        return 0, 0
    prob = math.exp(-delta/T)
    if random.random() < prob:     # худшее решение, принять
        return 2, prob
    return 0, prob                 # худшее решение, отвергнуть

def simulated_annealing(dist, p,
                        neighbormethod=0, verbose=False):
```

```
118     N = len(dist)
119     dmax = max([max(dist[i]) for i in range(N)])
120     dthreshold = dmax/2
121     d1 = [-1 for i in range(N)]
122     d2 = [-1 for i in range(N)]
123
124 ## Инициализация
125 median = random.sample(range(N), p)
126 r = update_assignment(dist, median, d1, d2, p , N)
127 first = [deepcopy(r), deepcopy(median)]
128 best = [r, median]
129 if verbose: print first[0]
130
131 accepted_same = 0
132 T = 100.0
133 while True:
134     result = next(r, T, median, dist, d1, d2, p, N,
135                   dthreshold, neighbormethod)
136     if result[0]>0:
137         r = result[1]
138         if r < best[0]:
139             best = [r, deepcopy(result[2])]
140             accepted_same = 0
141         if result[0]==2:
142             accepted_same += 1
143         if r == best[0] and accepted_same > 2:
144             break
145         T = 0.9*T
146         if verbose:
147             print r
148             if result[0] == 2:
149                 print '*',
150             print median
151     else: break
152 return first, best
153
154 if __name__ == "__main__":
155     print 'Задача: простая сеть'
156     a = network2distancematrix('../data/network-links', True)
157     allpairs(a)
158     result = simulated_annealing(a, 2, verbose=True)
159     print result[0][0], result[1][0]
160
161     print 'Задача: pmed1 из OR-lib'
162     a = network2distancematrix('../data/orlib/pmed1.orlib',
163                                False)
164     allpairs(a)
165     result = simulated_annealing(a, 5, verbose=True)
166     print result[0][0], result[1][0]
```

Программа в листинге 12.6 включает тесты на двух задачах: простой сети и первой задаче о *p*-медиане из библиотеки OR-Library, как и для алгоритма

обмена вершин. Мы приводим результаты выборочного прогона, чтобы проиллюстрировать некоторые интересные особенности алгоритма. Во-первых, видно, что были найдены оптимальные решения обеих задач. Позже мы проверим, как часто такое случается. Во-вторых, важно отметить, как именно (точнее, когда) встретилось оптимальное решение в обоих случаях. Мы печатаем звездочку (*), если принятое решение хуже текущего. Так, в прогоне для простой сети оптимальное решение (целевая функция равна 40) встретилось *после* того, как алгоритм принял худшее решение (целевая функция равна 45). Это означает, что, по крайней мере в этом случае, выбор худшего решения помог алгоритму найти оптимальное, отступив назад всего лишь на один шаг. Та же тенденция наблюдается для задачи pmed1, в этом случае было принято три худших решения и лишь затем найдено оптимальное, 5819. В-третьих, видно, что лучшие решения посещаются несколько раз, перед тем как алгоритм завершается. Очевидно, что наши ухищрения позволили избежать лишних итераций, уменьшив тем самым время вычислений.

```
Задача: простая сеть
[52, [2, 1]]
43 [5, 1]
45 * [5, 3]
40 [5, 0]
44 * [4, 0]
40 [5, 0]
43 * [5, 1]
45 * [5, 3]
43 [5, 1]
52 40
Задача: pmed1 из OR-lib
[9462, [11, 13, 12, 81, 38]]
7175 [11, 3, 12, 81, 38]
6589 [11, 3, 12, 34, 38]
6144 [11, 3, 12, 34, 90]
5897 [98, 3, 12, 34, 90]
5907 * [24, 3, 12, 34, 90]
5920 * [24, 2, 12, 34, 90]
5899 [24, 65, 12, 34, 90]
5905 * [24, 65, 41, 34, 90]
5856 [24, 65, 41, 6, 90]
5827 [24, 64, 41, 6, 90]
5821 [98, 64, 41, 6, 90]
5819 [98, 64, 12, 6, 90]
5843 * [25, 64, 12, 6, 90]
5821 [24, 64, 12, 6, 90]
5819 [98, 64, 12, 6, 90]
5821 * [24, 64, 12, 6, 90]
5827 * [24, 64, 41, 6, 90]
5821 [24, 64, 12, 6, 90]
9462 5819
```

Теперь посмотрим, как вероятность принятия изменяется в процессе поиска. Вероятность зависит от температуры и изменения значения целевой

функции. Мы напишем коротенькую программу для изучения динамики температуры и вероятности принятия при трех увеличивающихся значениях целевой функции:

```
from simulated_annealing import acceptable
T = 100.0
for i in range(100):
    print T, acceptable(1, T)[1], \
        acceptable(5, T)[1], \
        acceptable(10, T)[1]
    T = T*0.9
```

Результаты представлены на рис. 12.1. В самом начале температурная кривая резко убывает. Вероятности для трех величин изменения целевой функции демонстрируют различные тренды, наибольший спад имеет место между 20-й и 50-й итерациями. На одной же итерации малая величина дельты дает значительно большую вероятность, чем большая.

Теперь протестируем алгоритм имитации отжига на всех 40 задачах о *p*-медиане в библиотеке OR-Library (листинг 12.7).

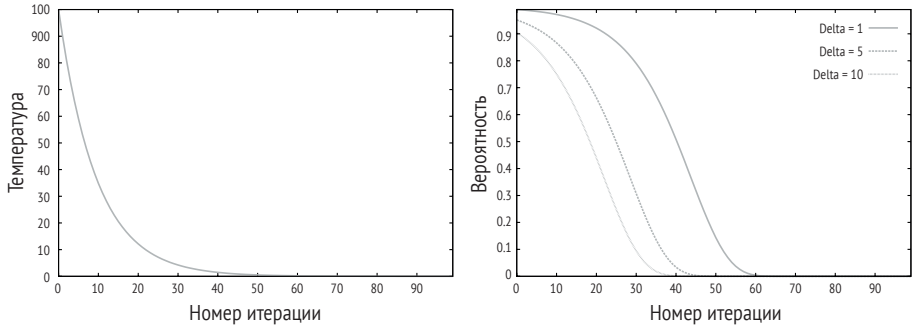

Рис. 12.1 ❖ Температура (слева) и вероятность принятия (справа) в алгоритме имитации отжига

Листинг 12.7 ❖ Тестирование алгоритма имитации отжига на 40 задачах о *p*-медиане из библиотеки OR-Library (test_orlib_pmed_simann.py)

```
import time
from test_orlib_pmed import load_distance_matrix
from simulated_annealing import *
from test_orlib_pmed import *

def testall():
    for i in range(len(N)):
        fn = format("../data/orlib/pmed%d.distmatrix"%(i+1))
        dist = load_distance_matrix(fn)
        t1 = time.time()
        result = simulated_annealing(dist, pp[i])
        t2 = time.time()
        print format("pmed%d:"%(i+1)), N[i], pp[i],\
```

```
                    result[1][0], known[i], t2-t1,
        if result[1][0]==known[i]: print "(*)"
        else: print

def testpmed(i, numtest=20, neighbormethod=0):
    fn = format("../data/orlib/pmed%d.distmatrix"%(i+1))
    dist = load_distance_matrix(fn)
    t1 = time.time()
    sumdiff = 0
    nummiss = 0
    for j in range(numtest):
        result = simulated_annealing(dist, pp[i],\
             neighbormethod=neighbormethod)
        print result[1][0],
        if result[1][0]==known[i]:
            print "(*)"
        else:
            print
            sumdiff += result[1][0] - known[i]
            nummiss += 1
    t2 = time.time()
    print "Время:", (t2-t1)/numtest
    print "Найдено:", numtest-nummiss
    if nummiss:
        print "Средняя разность:", float(sumdiff)/nummiss

if __name__ == "__main__":
    testall()
    testpmed(16)
    testpmed(16, neighbormethod=1)
```

Мы нашли оптимальные решения 15 из 40 задач, это похоже на результаты, показанные алгоритмом обмена вершин, который с первой попытки нашел 14 оптимальных решений. Но вот что интересно – для задач, в которых оптимальные решения не были найдены, расстояния больше, чем в случае обмена вершин. Например, для задач pmed37 и pmed40 в лучших решениях, найденных алгоритмом имитации отжига, целевая функция принимала значения 5080 и 5143, тогда как алгоритм обмена вершин давал для них значения 5067 и 5134 соответственно.

```
pmed1: 100 5 5819 5819 0.0839600563049 (*)
pmed2: 100 10 4093 4093 0.287177085876 (*)
pmed3: 100 10 4270 4250 0.191607952118
pmed4: 100 20 3098 3034 0.457482099533
pmed5: 100 33 1358 1355 0.779225826263
pmed6: 200 5 7824 7824 0.80565905571 (*)
pmed7: 200 10 5631 5631 0.725255012512 (*)
pmed8: 200 20 4445 4445 3.69324994087 (*)
pmed9: 200 40 2754 2734 9.48669195175
pmed10: 200 67 1265 1255 13.189332962
pmed11: 300 5 7696 7696 1.77202391624 (*)
pmed12: 300 10 6634 6634 4.31804585457 (*)
```

```
pmed13: 300 30 4374 4374 14.6342201233 (*)
pmed14: 300 60 2972 2968 34.3586940765
pmed15: 300 100 1734 1729 63.2703919411
pmed16: 400 6 7823 8162 1.8658220768
pmed17: 400 10 6999 6999 8.24238801003 (*)
pmed18: 400 40 4852 4809 45.7127890587
pmed19: 400 80 2870 2845 92.6540749073
pmed20: 400 133 1798 1789 164.285640001
pmed21: 500 5 9138 9138 3.01183795929 (*)
pmed22: 500 10 8579 8579 13.4972910881 (*)
pmed23: 500 60 4219 4619 126.247814894
pmed24: 500 100 2978 2961 262.334873199
pmed25: 500 167 1845 1828 508.916893959
pmed26: 600 5 9917 9917 7.942497015 (*)
pmed27: 600 10 8310 8307 15.1935739517
pmed28: 600 60 4524 4498 248.925246
pmed29: 600 120 3048 3033 497.998163939
pmed30: 600 200 2017 1989 977.196017981
pmed31: 700 5 10087 10086 7.5910961628
pmed32: 700 10 9301 9297 14.325248003
pmed33: 700 70 4727 4700 378.281183958
pmed34: 700 140 3020 3013 1355.75469995
pmed35: 800 5 10400 10400 11.208812952 (*)
pmed36: 800 10 9965 9934 22.7055418491
pmed37: 800 80 5080 5057 734.290642023
pmed38: 900 5 11060 11060 14.7149581909 (*)
pmed39: 900 10 9423 9423 35.8160691261 (*)
pmed40: 900 90 5143 5128 1251.52289701
```

Складывается впечатление, что имитация отжига хуже, чем алгоритм обмена вершин. Случайно ли это? Трудно сказать определенно. Пойдем дальше и исследуем конкретную задачу о *p*-медиане. Сначала проверим эффективность имитации отжига на задаче pmed17 из библиотеки OR-Library. Мы прогнали тест 20 раз и получили такой результат:

```
7010
7003
7003
7033
6999 (*)
7003
6999 (*)
7014
7010
7010
6999 (*)
7010
6999 (*)
6999 (*)
7003
7025
7003
```

```
6999 (*)
7010
6999 (*)
Время: 4.70761389732
Найдено: 7
Средняя разность: 11.5384615385
```

В семи из 20 прогонов на одной и той же задаче мы смогли найти оптимальное решение. Это меньше, чем показал алгоритм обмена вершин, зато среднее отклонение от оптимального значения целевой функции меньше, т. е. обе эвристики демонстрируют очень похожую эффективность нахождения оптимальных решений, по крайней мере если судить по выполненным тестам.

Во всех тестах имитации отжига мы использовали для порождения соседних решений функцию `bestGeoNeighbor`. А теперь кратко рассмотрим другой подход. Показанный ниже результат говорит о том, что метод случайного порождения не сумел найти ни одного оптимального решения за 20 прогонов (приведена только часть результатов). Поэтому во всех остальных тестах мы будем использовать только функцию `bestGeoNeighbor`.

```
8899
8628
...
8540
8555
0.323373091221
```

Оба алгоритма продемонстрировали сходимость к оптимальному или почти оптимальному решению. На рис. 12.2 представлен общий тренд обоих алгоритмов при 20 прогонах на одной задаче. Поскольку в основу алгоритмов положены сходные механизмы замены вершин, следовало ожидать, что и тренды будут похожи. Однако мы также видим, что для сходимости алгоритма обмена вершин требуется меньше итераций. Было бы разумно объяснить такую быструю сходимость тем, что этот алгоритм рассматривает все возможные замены непосредственно соседней вершиной (т. е. в качестве кандидатов на замену рассматриваются только вершины, соединенные ребром с выбранными вершинами), тогда как в алгоритме имитации отжига рассматривается только подмножество возможных замен, но принимаются во внимание как непосредственно соседствующие, так и более отдаленные вершины.

Сравнение сходимости обоих алгоритмов обнаруживает также значительное различие во времени вычислений: алгоритм имитации отжига в его текущем виде работает намного дольше алгоритма обмена вершин. Очевидно, что временем работы имитации отжига можно управлять с помощью порогового расстояния. Мы выбрали порог, при котором рассматриваются даже вершины, не соединенные ребрами ни с одной из выбранных. Это значит, что мы рассматриваем больше замен, из чего и проистекает увеличение времени работы. Это также означает, что имитация отжига, возможно, «сведется» к обмену вершинами, если мы потребуем, чтобы на каждой итерации

рассматривались только непосредственно соседствующие вершины. Иными словами, реализацию алгоритма имитации отжига, приведенную в этой главе, можно считать обобщением алгоритма обмена вершин.

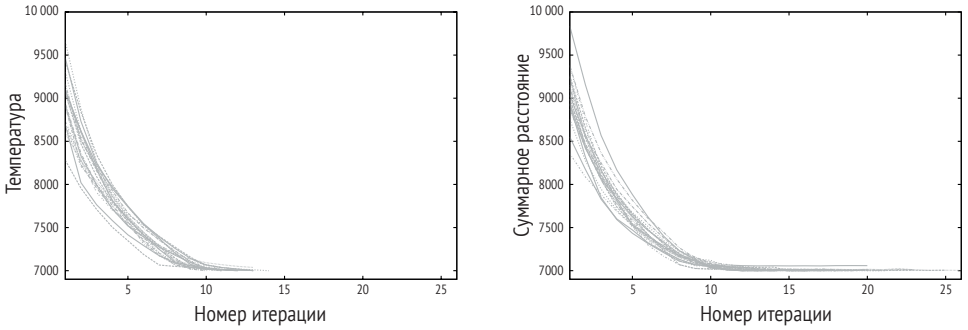

Рис. 12.2 ❖ Сходимость алгоритмов обмена вершин (слева) и имитации отжига (справа) для задачи pmed17

Теперь прогоним тест на задаче pmed1. Алгоритм обмена вершин нашел оптимальное решение во всех 20 прогонах (среднее время составило 0.058 секунды). Результат алгоритма имитации отжига выглядит так:

```
5868
5868
5868
5868
5868
5819 (*)
5868
5819 (*)
5868
5819 (*)
5819 (*)
5821
5819 (*)
5819 (*)
5819 (*)
5819 (*)
5868
5868
5868
5819 (*)
Время: 0.101140642166
Найдено: 9
Среднее расстояние: 44.7272727273
```

В связи с этим возникает вопрос: верно ли, что алгоритм имитации отжига менее устойчив, чем алгоритм обмена вершин? Ясно одно: объем проведенного тестирования не позволяет прийти к определенным выводам. Но

мы на этом остановимся. В упражнениях обсуждение данного вопроса будет продолжено.

12.4. Примечания

Задачам размещения и размещения-распределения посвящено огромное количество работ (Daskin, 1995; Drezner, 1995). Это краеугольный камень того, что мы в этой книге назвали пространственной оптимизацией. Алгоритмы, рассмотренные в этой и предыдущей главах, – лишь вершина айсберга. Слово «эвристический» происходит от греческого *eureka*, означающего «нашел». Легенда гласит, что когда Архимед открыл, как измерить объем тела неправильной формы, он голым выбежал на улицу с криком «Эврика!». В нашем обсуждении алгоритмов поиска мы называли эвристическим метод, позволяющий находить приближенные решения задач оптимизации.

Многие методы заимствованы из обширного пласта научных исследований, особенно из литературы по информатике. Помимо алгоритма имитации отжига (Kirkpatrick et al., 1983; Cerny, 1985), рассмотренного в этой главе, к числу метаэвристических методов можно отнести генетические алгоритмы (Goldberg, 1989), поиск с запретами (Gendreau et al., 1997), колонии муравьев (Dorigo and Gambardella, 1997) и другие (Michalewicz and Fogel, 2000). Генетические алгоритмы принадлежат к семейству так называемых эволюционных алгоритмов. У алгоритма имитации отжига имеются и другие вариации, например использование порога вместо температуры (Dueck and Scheuer, 1990). Подобные алгоритмы всесторонне исследовались в различных приложениях пространственной оптимизации (Krzanowski and Raper, 2001; Xiao et al., 2002, 2007; Xiao, 2006, 2008).

Метод обмена вершин был предложен в работе Teitz and Bart (1968). Хотя идея, лежащая в основе его эвристики, проста, многие исследователи продемонстрировали, что он весьма эффективно находит по крайней мере почти оптимальные решения задачи о *p*-медиане (Rosing et al., 1979; Rosing, 1997; Xiao, 2012). После появления метода обмена вершин было создано много его вариантов (Densham and Rushton, 1992; Rosing and Hodgson, 2002; Resende and Werneck, 2004). Все они тестировались на разного рода эталонных задачах, в т. ч. включенных в библиотеку OR-Library (Beasley, 1985). Но оригинальная реализация этого алгоритма неэффективна. Алгоритм `findout`, используемый в методе обмена вершин, разработан в работе Whitaker (1983); он позволил заметно улучшить общую производительность и использовался также во многих других алгоритмах (Hansen and Mladenović, 1997; Resende and Werneck, 2003, 2004).

Поскольку выбор методов столь широк, рано или поздно возникает вопрос, какой из них лучше. Теорема об отсутствии «бесплатных завтраков» (Wolpert and Macready, 1997) предостерегает нас от выводов, выходящих за пределы протестированного. Мы были осторожны и не стали выносить определенное заключение по поводу двух алгоритмов решения задачи о *p*-медиане. Хотя мы и провели тщательное тестирование на нескольких задачах, всегда най-

дется много других, и у нас нет никаких оснований для прогноза того, как наши эвристики поведут себя в новых условиях.

Один из способов справиться с разнообразием алгоритмов – объединить их в гибридные алгоритмы. Алгоритм мультистарта (Resende and Werneck, 2004) представляет собой гибридный алгоритм, содержащий в качестве компонентов много полезных алгоритмов. Можно также рассматривать любой существующий алгоритм как программный агент и организовать взаимодействие между разными агентами, так чтобы они решали задачу совместно (Xiao, 2012). Эти агенты могут обмениваться решениями, применяя различные механизмы. Их можно также распределить между несколькими компьютерами в сети и тем самым ускорить работу гибридных процессов, которые вообще-то потребляют очень много времени.

12.5. Упражнения

1. Мы спроектировали жадный алгоритм решения задачи о p-медиане на основе принципа наращивания. В разделе 12.1 мы упомянули принцип удаления, который также позволяет построить пригодное решение. Реализуйте жадный алгоритм, основанный на удалении, и протестируйте его на задачах из этой главы. Следует ли ожидать существенно других результатов по сравнению с наращиванием?
2. Спроектируйте и реализуйте жадный алгоритм решения дискретной задачи о p-центрах, описанной в упражнениях к главе 11.
3. Спроектируйте и реализуйте жадный алгоритм решения задачи о покрытии множества, описанной в упражнениях к главе 11.
4. Рассмотренная выше задача о 1-центре – это дискретная задача размещения с конечным множеством клиентов, но ее пространство решений непрерывно. Мы можем решить ее точно, воспользовавшись одним из описанных методов. Но если требуется выбрать несколько центров, то задача усложняется. Поскольку центры непрерывны, невозможно написать частично целочисленную программу, т. к. такая программа способна работать только с дискретными решениями. Это так называемая непрерывная задача о p центрах (где $p > 1$). Модифицируйте алгоритм k средних, описанный в главе 9, так чтобы его можно было использовать в качестве эвристического алгоритма решения непрерывной задачи о p центрах.
5. Протестируйте жадный алгоритм на всех 40 задачах о p-медиане из библиотеки OR-Library. Распечатайте время, затраченное на решение каждой задачи.
6. При проектировании эвристических методов очень важен вопрос о том, как выбраться из локального оптимума. Сравните стратегии, принятые в трех алгоритмах решения задачи о p-медиане, описанных в этой главе. Придумайте свои собственные стратегии и изучите их эффективность. Попробуйте несколько стратегий, чтобы иметь основания для выводов в этом вопросе.

7. Существуют ли заведомо простые задачи о p-медиане? Иными словами, можно ли спроектировать сеть с такой структурой, что алгоритм поиска всегда будет находить оптимальное решение, каким бы ни было начальное приближение? С другой стороны, можно задаться вопросом, существуют ли заведомо трудные сети, структура которых такова, что обязательно заводит процесс поиска в тупик. Верно ли, что такие особенности зависят от алгоритма, т. е. могут обмануть один алгоритм, но не могут – другой? Размышляя на эти темы, спроектируйте модельную сеть, обладающую некоторыми из таких свойств.
8. Мы использовали 40 задач о p-медиане из библиотеки OR-Library, для которых оптимальные решения известны. А что, если нам понадобится больше тестовых задач? Попробуйте самостоятельно спроектировать несколько задач с известными оптимальными решениями. Для их решения используйте методы частично целочисленного программирования.

Послесловие

Осел: Мы уже приехали?
Шрек: Нет.
Осел: Мы уже приехали?
Фиона: Нет.
Осел: Мы уже приехали?
Фиона: Нет.
Осел: Мы уже приехали?
Шрек: Нет!
Осел: Мы уже приехали?
Шрек: Да.
Осел: Правда?
Шрек: Нет!!!

Шрек-2 (2004)

Мы начали эту книгу с введения в пространственные данные и некоторые базовые геометрические алгоритмы. Затем перешли к обсуждению методов индексирования, которые позволяют ускорить обработку пространственных данных. После этого мы рассмотрели ряд приложений пространственного анализа. В каждой главе мы уделяли внимание теории, методам и реальному коду. Мы на примерах продемонстрировали, не претендуя на слишком многое, что написание кода дает нам свободу делать полезные вещи, не полагаясь на большие программные пакеты. Но приехали ли мы?

Каждую главу можно было бы развернуть в отдельную книгу, а то и не одну. Мы рассмотрели некоторые наиболее важные темы, ставя во главу угла обсуждение идей и написание простых программ на Python, которые реализуют различные методы. В примечаниях мы старались показать направления развития, но были ограничены местом. Понадобится больше усилий, в т. ч. коллективных, чтобы охватить вопросы, связанные с геопространственными данными, во всей широте и глубине. Кроме того, мы, конечно, могли бы сделать больше с включенным в книгу кодом. Например, все программы использовались отдельно одна от другой, и на то были причины – мы хотели обсуждать программы для каждого описываемого алгоритма независимо от других. Но теперь, когда мы познакомились со всеми программами, можно собрать их в один Python-пакет, чтобы с ними было проще работать. Однако следует отметить, что программы в этой книге написаны в минималистском стиле, чтобы не отвлекать внимание от алгоритмов. Все они *работают*, но, возможно, не при любых условиях. Мы не проверяли ни тип, ни допустимость параметров, если только это не было абсолютно необходимо. Например, не проверялось, что в *k*D-дерево вставляются только точки (объекты класса Point). Это может привести к дефектам, из-за которых программа аварийно завершается. Кто-то скажет, что это внутреннее ограничение слаботипизированного языка программирования, каковым является Python. Но зато это

позволяет нам размышлять об алгоритмической стороне вопроса на более интуитивном уровне, не слишком вдаваясь в детали.

Между местом, где мы находимся, и местом, куда хотим попасть, лежит обширная область науки геоинформатики и пространственного анализа, а также строгие процессы программной инженерии. Но это не умаляет ценности книги: все это часть процесса, который поможет нам обрести свободу в работе с пространственными данными.

Приложение A

Введение в Python

Красивое лучше, чем уродливое.
Явное лучше, чем неявное.
Простое лучше, чем сложное.
Сложное лучше, чем запутанное.
Плоское лучше, чем вложенное.
Разреженное лучше, чем плотное.
Читаемость имеет значение.

Тим Питерс «Дзен Питона»

Это приложение предназначено в основном читателям, незнакомым с некоторыми средствами Python, используемыми в данной книге. Оно даст возможность быстро познакомиться с языком, но не следует считать его основательным введением. Тем, что хочет по-настоящему понять Python, лучше обратиться к другим источникам. Различных пособий хватает, и многие из них бесплатны. Например, в официальном руководстве по Python[1] освещается широкий круг тем и приводится множество примеров, которые легко перенести на более сложные случаи. В других пособиях, например «How to Think Like a Computer Scientist»[2], представлен более специализированный материал. Но так или иначе, ниже мы рассмотрим некоторые вопросы, тесно связанные с алгоритмами, описанными в этой книге. Наша цель – простота, поскольку более подробную информацию легко найти в сети или в других публикациях.

Python – кросс-платформенный язык программирования. В этой книге мы предполагаем, что Python и многие модули, входящие в состав дистрибутива, уже успешно установлены. Мы пользуемся только командной строкой оболочки Python. Прежде чем переходить к конкретным темам, опишем некоторые базовые команды. Отметим, что любая команда вводится после приглашения >>>. В этом приложении предполагается версия Python 2.7.6.

Python можно использовать как мощный калькулятор. Однако нужно внимательно следить за типами объектов. Значение `2` считается целым числом; вычисление `2/3` рассматривается как операция над двумя целыми числами, и ее результат – целое число (то есть 0). Для получения числа с плавающей точкой необходимо, чтобы хотя бы один операнд был числом с плавающей

---

[1] https://docs.python.org/2.7/tutorial/.

[2] http://openbookproject.net/thinkcs/python/english3e/index.html.

точкой (например, `2.0/3`). Можно также использовать функцию `float`, которая преобразует свой аргумент в число с плавающей точкой.

```
>>> 10 * 10
100
>>> 2/3
0
>>> 2.0/3
0.6666666666666666
>>> 2/3.0
0.6666666666666666
>>> float(2)/3
0.6666666666666666
>>> 2/float(3)
0.6666666666666666
```

Переменные играют важную роль в Python (как и в любом языке программирования). Однако у переменных в Python нет типов. Тип переменной определяется интерпретатором Python во время выполнения. В следующем примере мы сначала присваиваем переменной `b` целое значение, а затем преобразуем его в текстовую строку (заключенную в двойные или одиночные кавычки). Если мы попробуем прибавить строку к числу, Python сообщит об ошибке. Заметим также, что символ # в Python обозначает комментарий, и все следующее за ним до конца строки интерпретатор игнорирует.

```
>>> a = 12              # переменная, имеющая целое значение
>>> b = 10              # еще одна переменная
>>> a + b               # сложение двух переменных
22
>>> b = 'mystring'      # теперь значением b является строка
>>> a + b
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Если обе переменные строковые, то их можно складывать. Заметим, что строку можно заключать как в одиночные, так и в двойные кавычки, лишь бы одинаковые.

```
>>> a = "astring"
>>> a + ' ' + b
'astring mystring'
```

Операция * дублирует строку:

```
>>> a * 5
'astringastringastringastringastring'
>>> (a + ' ') * 5
'astring astring astring astring astring '
```

Можно также подставлять значения переменных в строку, добиваясь выразительного форматирования, для этого предназначен встроенный метод

format, имеющийся у любой строки. Первая часть команды – частичная строка, в которой фигурные скобки обозначают места, куда будут подставлены значения, перечисленные в круглых скобках после имени метода – слова format. Параметры метода нумеруются, начиная с нуля, и эти номера указываются в частичной строке.

```
>>> n1, n2, n3 = 10, 1, 2
>>> 'I have {0} apples, {1} pears, and {2} others'.\
... format(n1, n2, n3)
'I have 10 apples, 1 pears, and 2 others'
```

Первая строка в этом примере иллюстрирует еще одну полезную возможность Python – одновременное присваивание, это позволяет в одной операции присвоить значения сразу нескольким переменным. Благодаря подобной возможности функции в Python могут возвращать сразу несколько значений.

```
>>> a,b = 3,4
>>> a,b
(3, 4)
>>> a,b = b,a
>>> a,b
(4, 3)
```

Еще один способ форматирования строки дает операция %. Она не настолько интуитивно очевидна, но может быть полезной. Символы % внутри кавычек отмечают места, куда подставляются значения, вслед за % указывается символ, обозначающий тип переменной или значения. Буквы d, f и s обозначают соответственно типы: целый, с плавающей точкой и строковый. Для значений с плавающей точкой можно также писать %.2f – тогда будут выведены только две цифры после запятой. Символ % после конечной кавычки означает, что далее следуют переменные или значения, подставляемые вместо спецификаторов формата внутри кавычек.

```
>>> n1, n2, n3 = 10, 1, 2
>>> "n1 = %d, n2 = %d, n3 = %d"%(n1, n2, n3)
'n1 = 10, n2 = 1, n3 = 2’
>>> name = "Dr. P"
>>> age = 40
>>> height = 6.1
>>> "%s is %d years old and %.2f feet tall"%(name,age,height)
'Dr. P is 40 years old and 6.10 feet tall'
```

Наконец, функция help возвращает полезные пояснения и примеры использования функций Python. Ее параметры, являющиеся символами Python, необходимо заключать в кавычки.

```
>>> help(format)
>>> help("<")
>>> help("==")
>>> help("%")
```

A.1. Списковое включение

В списке Python можно хранить несколько элементов любого типа, это самая мощная структура данных в Python. В следующем примере `a` – список, содержащий шесть разных значений. Функция `len` возвращает длину списка. Каждый элемент списка определяется индексом, нумерация начинается с 0.

```
>>> a = ['str1', 10, 'str2', 'str3', 10, 5]
>>> a
['str1', 10, 'str2', 'str3', 10, 5]
>>> len(a)
6
>>> a[0] 'str1'
>>> a[0] + ' ' + a[2]
'str1 str2'
>>> a[1] + a[5]
15
```

Очень полезным средством является функция `range`, возвращающая список чисел. Если задан только один аргумент (например, 10), то он расценивается как ограничитель, и функция отдает целые числа, начиная с 0 и кончая значением, на единицу меньшим аргумента.

```
>>> range(9)
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> range(2, 9)
[2, 3, 4, 5, 6, 7, 8]
>>> range(2, 9, 3)
[2, 5, 8]
```

Операция создания списка называется в Python списковым включением (list comprehension). Список можно создать разными способами. Поскольку список содержит набор элементов, традиционный способ – добавлять значения в цикле `for`. Сначала создается пустой список ([ ]), а потом в него помещаются элементы с помощью функции `append`. В следующем примере создается список 10 значений, равных степеням 2 для *i*, изменяющегося от 0 до 9. (Оператор ** означает «возведение в степень».)

```
>>> exps = []
>>> for i in range(10):
...     exps.append(2**i)
...
>>> exps
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

Прежде чем двигаться дальше, обратим внимание на очень важное синтаксическое правило Python: отступы. В Python отступы используются для обозначения блоков кода – вместо скобок или иных символов. В примере выше код, логически находящийся внутри цикла `for`, должен начинаться с нескольких пробелов (в нашем случае – четырех). Предшествующая отсту-

пу строка должна заканчиваться двоеточием (:). Это правило применяется всякий раз, как нужно обособить группу строк – мы часто встречаем такую ситуацию в этом приложении и в основном тексте книги.

Воспользовавшись списковым включением, этот пример можно заменить одной строчкой кода (см. ниже), возвращающей точно такой же список. Заодно мы показали сравнение двух списков с помощью логического оператора ==, который возвращает True, только если элементы обоих списков в точности совпадают. В любом списковом включении есть по меньшей мере два вида операций: присваивание и перечисление. В примере ниже присваивание (2**i) применяется к результату перечисления (организованного с помощью предложения for) значений i, выбираемых из списка 10 целых чисел, возвращаемого функцией range.

```
>>> exps1 = [2**i for i in range(10)]
>>> exps1 == exps
True
```

В списковое включение можно также вставлять условия, разнообразив механизм создания списков. В следующем примере создается список, содержащий только те значения из exps1, которые нацело делятся на 32. Здесь символ % – оператор деления по модулю, который возвращает остаток от деления первого аргумента на второй (например, 5%3 возвращает 2).

```
>>> [x for x in exps1 if x%32==0]
```

Можно даже подставить вместо exps1 выражение, получив вложенное списковое включение:

```
>>> [x for x in [2**i for i in range(10)] if x%32==0]
```

В разделе 2.8 используются следующие два цикла для создания списка точек, образующих параллели, где каждая параллель определяется 37 точками в диапазоне от –180 до 180 с шагом 10.

```
>>> points = []
>>> linenum = 0
>>> for lat in range(-90, 91, 10):
...     for lon in range(-180, 181, 10):
...         points.append([linenum, lon, lat])
...     linenum += 1
...
```

Этот код можно упростить, воспользовавшись списковым включением:

```
>>> points = [ [(lat+90)/10, lat, lon]
...            for lat in range(-90, 91, 10)
...            for lon in range(-180, 181, 10) ]
```

Координаты *X* и *Y* точек на 17-й параллели (80° с.ш.) также можно получить с помощью спискового включения:

```
>>> [[p[1], p[2]] for p in points if p[0]==17]
```

A.2. Функции, модули и рекурсивные функции

Как и в любом языке программирования, группу строк можно оформить в виде функции, которая будет вызываться многократно. В Python для определения функции служит ключевое слово `def`. Сразу после него указывается имя функции, а за ним – множество аргументов в круглых скобках. Строка, содержащая заголовок определения функции, должна заканчиваться двоеточием. Продемонстрируем работу функций на простом примере.

Мы определяем функцию, которая вычисляет расстояние между двумя точками с координатами (`x1`, `y1`) и (`x2`, `y2`). Для извлечения квадратного корня из суммы квадратов нам понадобится функция `sqrt`. Однако эта функция не встроена в Python, а находится в Python-модуле `math`, который нужно явно импортировать (в первой строке). Все строки, принадлежащие функции, должны начинаться отступом. В примере ниже мы опустили приглашения Python, потому что вводить несколько строчек кода в интерактивной оболочке неудобно, т. к. невозможно редактировать ранее введенные строки. Вместо этого мы напишем код Python в текстовом редакторе и сохраним его в файле. Хороших редакторов много, выбирать есть из чего. Я больше всего люблю Emacs, но Notepad++ – также популярный и бесплатный редактор. Предположим, что показанный ниже код сохранен в файле `dist.py`. Тогда он становится частью модуля, который можно будет при необходимости импортировать.

```
import math
def distance(x1, y1, x2, y2):
   dx = x1-x2
   dy = y1-y2
   d = math.sqrt(dx*dx + dy*dy)
   print "Мое имя:", __name__
   return d
```

Ниже показано, как использовать этот код для вычисления расстояния.

```
>>> from dist import *
>>> distance(1.5, 2.5, 3.5, 4.0)
Мое имя: dist
2.5
>>> print __name__
__main__
```

Таким образом, мы создали Python-модуль `dist`. В общем случае Python-модуль включает код, который можно использовать вне исходного файла. В примере выше показана также важная встроенная в Python переменная `__name__`. В процессе, исполняющем интерпретатор Python (в нашем случае это командная оболочка Python), ей присваивается значение `__main__`. А когда сам Python загружает модуль, он записывает в переменную `__name__` его имя.

В следующем примере показано вычисление факториала целого числа по формуле $n! = n \times (n - 1) \times (n - 2) \times \ldots \times 3 \times 2 \times 1$. Здесь мы просто перемножаем целые числа, взятые из списка от 1 до `n`.

```
def factorial_i(n):
   f = 1
   for i in range(1,n+1):
      f = f*i
   return f
```

Но вычисление *n*! можно организовать и по-другому, сделав так, что функция будет вызывать саму себя. Такая функция называется рекурсивной.

```
def factorial_r(n):
   if n == 1:
      return 1
   return n * factorial_r(n-1)
```

Рекурсивные функции часто просты и элегантны. Однако нужно внимательно следить за тем, чтобы рекурсия не оказалась бесконечной. Любая рекурсивная функция должна когда-то останавливаться. В примере выше функция вызывает саму себя в последней строке. Если ничего не предпринять, то она будет вызывать себя до бесконечности. Поэтому перед вызовом функции мы обязательно размещаем условие. В данном случае, когда `n` становится равным 1, мы возвращаем константу, а не вызываем себя еще раз. Это и есть база рекурсии. Кроме того, рекурсивная функция должна каким-то образом продвигаться в направлении базы. В нашем примере функция гарантированно достигает базы, потому что мы всякий раз вызываем ее, задавая меньшее значение `n`. В основном тексте книги немало примеров рекурсивных функций, особенно при работе с деревьями, где благодаря рекурсии легко организовать спуск по дереву до листового узла. Но у рекурсивных функций есть серьезный недостаток: низкая производительность. Например, если `n` равно 10 и мы вызываем `factorial_r(10-1)`, то ответ будет получен не сразу – нам придется подождать, пока `n` не достигнет 1, и только тогда, на обратном пути, мы сможем вычислить все факториалы при разных значениях `n`. Такой возврат по пути выполнения требует большого объема памяти для хранения еще не завершенных вызовов функций, да и времени тратится больше, чем на перебор в цикле.

A.3. Лямбда-функции и сортировка

Иногда нам нужна функция, но мы не хотим заводить настоящую функцию, имеющую имя. Для создания таких анонимных функций в Python применяются лямбда-выражения. Например, ниже определена функция, которая возвращает `True`, если на вход подано четное число, и `False` в противном случае. Это быстрый способ определить функцию, не используя ключевое слово `def` со всеми его синтаксическими премудростями. Тут все просто: переменная, предшествующая запятой, – это входной параметр, а все, что после запятой, вычисляется и возвращается.

```
>>> f = lambda x: x%2==0
>>> f(10)
```

```
True
>>> f(1)
False
```

Такой способ определения функций может показаться ненужным и только сбивающим с толку. Однако же он обладает огромной выразительностью в плане абстрагирования кода. Продемонстрируем это на примере сортировки списка. В Python список можно отсортировать, воспользовавшись встроенным методом `sort`.

```
>>> exps1 = [2**i for i in range(10)]
>>> exps1.sort(reverse=True)
>>> exps1
[512, 256, 128, 64, 32, 16, 8, 4, 2, 1]
```

Однако этот подход не работает, когда элементы списка имеют более сложную структуру. Например, если каждый элемент списка сам является списком, содержащим координаты *X* и *Y* точки, то при сортировке мы получим результаты, основанные на первом значении каждого элемента. А если мы хотим отсортировать список по второму значению элемента? Тут-то и приходит на помощь лямбда-функция. Мы используем параметр `key` функции `sort`, чтобы указать, что сортировать нужно по второму значению каждого элемента.

```
>>> points=[[10,1], [1,5], [1,2], [3,7], [10,8], [7,8], [9,8]]
>>> points.sort()
>>> points
[[1, 2], [1, 5], [3, 7], [7, 8], [9, 8], [10, 1], [10, 8]]
>>> points.sort(key=lambda points: points[1])
>>> points
[[10, 1], [1, 2], [1, 5], [3, 7], [7, 8], [9, 8], [10, 8]]
```

A.4. NumPy и Matplotlib

NumPy – это мощный Python-модуль, разработанный специально для операций с массивами и матрицами. В главе 8 мы то и дело пользовались функциями из NumPy при обсуждении методов интерполяции. Например, мы легко можем преобразовать список Python в массив NumPy. Многие функции NumPy позволяют напрямую работать сразу с несколькими элементами входного объекта. Продемонстрируем эти возможности на простом примере. Здесь функция NumPy `sin` применяется непосредственно к списку `angles`. На внутреннем уровне NumPy преобразует список в массив, вычисляет синус каждого элемента и формирует из этих значений выходной массив. Однако функция `sin`, входящая в состав модуля `math`, так работать не умеет, она ожидает получить на входе одно значение с плавающей точкой.

```
>>> import numpy as np
>>> angles = [i*np.pi/10 for i in range(1, 10)]
>>> np.sin(angles)
```

```
array([0.30901699, 0.58778525, 0.80901699, 0.95105652, 1.,
       0.95105652, 0.80901699, 0.58778525, 0.30901699])
>>> import math
>>> math.sin(angles)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: a float is required
```

Matplotlib – исключительно полезный Python-пакет, предназначенный для построения двумерных графиков. Несмотря на интерфейс в духе командной строки, этот пакет довольно просто использовать для создания впечатляющих графиков и диаграмм. Ниже приведено три примера его применения в различных областях, связанных с обработкой пространственных данных. Пакет Matplotlib может работать в двух режимах: IPython и pylab. Режим pylab предоставляет основной интерфейс к функциональности Matplotlib, и мы импортируем его в следующем примере. Отметим, что мы также импортировали NumPy под псевдонимом `np`. Мы создаем список 60 углов, покрывающих диапазон, больший 2π. Функция `plot` создает график, в первых двух аргументах ей передаются списки координат *X* и *Y*. Дополнительно у функции `plot` существует множество параметров, управляющих внешним видом графика. На экране ничего не появится, пока не будет вызвана функция `show`. На рис. А.1 показан результат.

```
from pylab import * # или: import matplotlib.pyplot as plt
angles = [i*pi/20 for i in range(60)]
plot(angles, np.sin(angles), color='grey')
plot(angles, np.cos(angles), linewidth=2, color='lightblue')
show()
```

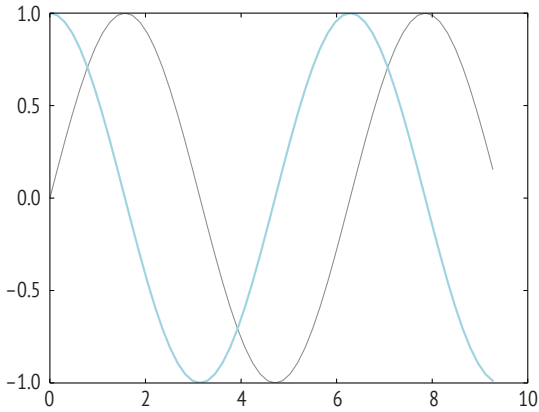

Рис. А.1 ❖ Построение графика с помощью Matplotlib

Функция `plot` применяется для построения линейных карт. В главе 2 мы использовали ее при изображении трех картографических проекций, где присутствовали сетка параллелей и меридианов и очертания материков. Для рисования многоугольников в Matplotlib имеется функция `Polygon`.

Во втором примере иллюстрируется рисование диаграммы рассеяния, состоящей из отдельных точек. Ее можно назвать эквивалентом «точечной карты». Мы создаем 100 точек, из которых 50 рассредоточены вокруг точки (0,0) в декартовой системе координат, а другие 50 – вокруг точки (10,10). Координаты *X* и *Y* выбираются из нормального распределения со стандартным отклонением 1. Для этого мы воспользовались функцией NumPy `normal`, которой передали среднее, стандартное отклонение и объем выборки. Координаты *X* и *Y* для каждой группы точек создаются по отдельности, а затем объединяются с помощью функции NumPy `concatenate`. Точки, принадлежащие первой группе, – вокруг (0,0) – рисуются красным цветом, а принадлежащие второй группе – синим. Для этого создается список цветов `col`, содержащий 100 элементов: первые 50 равны `red`, оставшиеся 50 – `blue`. Наконец, вызывается функция `scatter` для построения диаграммы, ей передаются не только сами координаты точек, но и цвет каждой точки (`col`) и степень прозрачности. Прозрачность 0.5 позволяет изображать перекрывающиеся точки. На рис. А.2 показан результат работы этой программы. Заметим, что мы экспортировали рисунок в формате PDF, который преобразуется в формат публикации EPS отдельно, потому что Matplotlib не поддерживает прозрачность в EPS.

```
import matplotlib.pyplot as plt
import numpy as np
n = 100
X1 = np.random.normal(0, 1, n/2)
X2 = np.random.normal(10, 5, n/2)
Y1 = np.random.normal(0, 1, n/2)
Y2 = np.random.normal(10, 5, n/2)
X = np.concatenate((X1, X2))
Y = np.concatenate((Y1, Y2))
col = ['red' if i<n/2 else 'blue' for i in range(n)]
plt.scatter(X, Y, color=col, alpha=.5)
plt.axis('scaled')
plt.savefig('matplotlib-scatter.pdf',
            bbox_inches='tight',pad_inches=0)
plt.show()
```

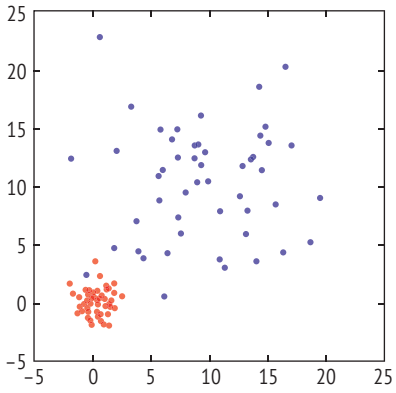

Рис. А.2 ❖ Рисование точек с помощью Matplotlib

И в последнем примере мы повторно воспользуемся разработанной ранее функцией `distance` для создания растрового изображения с двумя центрами: в центрах левого нижнего и правого верхнего прямоугольников. Для каждой точки изображения мы находим ближайший к ней центр и задаем значение в этой точке с помощью бивесового ядра (см. раздел 9.4). Функция `K` возвращает нужное значение.

```
from dist import distance
def K(x, y, c1, c2, r):
   d1 = distance(x, y, c1[0], c1[1])
   d2 = distance(x, y, c2[0], c2[1])
   d, i = d1, 0
   if d2<d1:
      d,i = d2,1
   if d>r:
      v = 0
   else:
      u = float(d/r)
      v = 15.0/16.0 * (1 - u**2)**2
   return v
```

Приведенный ниже код создает растровое изображение размера 20×20 с шириной полосы ядерной функции 10. Для создания списка из 20 значений мы применяем списковое включение. Переменные `c1` и `c2` описывают положения центров. `Z` – это двумерный список (список списков), содержащий значения пикселей, генерируемые функцией `K`. Функция Matplotlib `imshow` создает двумерное изображение по значениям в списке `Z`. Мы используем цветовую карту `Greys`, чтобы создать черно-белое изображение. По умолчанию функция `imshow` располагает данные с наименьшим значением на одном конце карты (белом), а с наибольшим – на другом (черном). Однако при этом карта получится слишком контрастной. Поэтому мы принудительно заставляем функцию использовать в качестве минимального значение –0.5, а в качестве максимального – 1.2, хотя они и выходят за границы диапазона данных (от 0 до 0.9375). В результате фактически использованные цвета окажутся в середине цветовой карты. Параметр `origin` функции `imshow` располагает начало координат в левом верхнем углу изображения, что соответствует нашим данным. Функция `colorbar` отображает пояснительную надпись, при этом мы указываем, что высота планки должна составлять 99 % от высоты карты. Функциям `xticks` и `yticks` передаются пустые списки, поэтому риски скрываются, и рисунок больше напоминает настоящую карту. Получившийся результат показан на рис. А.3.

```
from pylab import *
n = 20
r = n/2.0
c1 = [n/4, n/4]
c2 = [3*n/4, 3*n/4]
Z = [ [K(x, y, c1, c2, r) for x in range(1, n+1) ]
      for y in range(1, n+1)]
imshow(Z, cmap='Greys', vmin=-0.5, vmax=1.2, origin='lower')
```

```
colorbar(shrink=.99)
xticks([]), yticks([])
show()
```

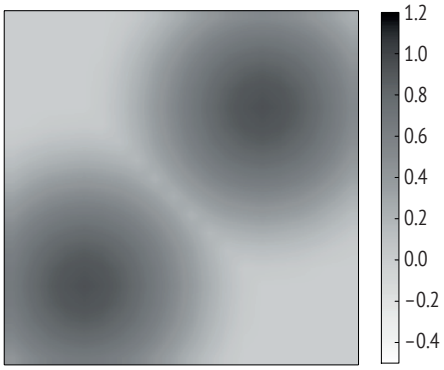

Рис. А.3 ❖ Построение растрового изображения с помощью Matplotlib

А.5. Классы

Данные и функции, предназначенные для работы с этими данными, можно сгруппировать в класс. В Python данные, инкапсулированные в классе, обычно называют атрибутами, а функции – методами. Создадим класс `Shape` для хранения нескольких точек. Первая строка содержит объявление класса. Мы хотим определить в этом классе два метода, оба встроенных. В Python метод называется встроенным, если он изначально является частью любого класса и предназначен для вполне определенной цели. Например, метод `__init__` всегда вызывается в момент создания экземпляра класса, поэтому его также называют конструктором класса, и предполагается, что код этого метода должен написать пользователь. В частности, при создании экземпляра класса `Shape` необходимо задать множество точек. Любому методу класса передается в качестве первого аргумента переменная, называемая `self`. Это позволяет передать методу (функции) ссылку на сам класс (со всеми его данными и методами), что очень удобно. Так, в методе `__init__` мы можем присвоить переданные аргументы атрибутам класса `Shape`. Определить атрибут очень просто: нужно лишь написать его имя, предпослав ему префикс `self.`, где точка означает, что следующая за ним строка является именем члена класса. Еще один встроенный метод `__repr__` определяет, как экземпляр класса представляется в текстовом виде. Естественно, он должен возвращать строку. В данном случае мы просто преобразуем список точек в строку.

```
class Shape():
   def __init__(self, points, id=None, t=None):
      self.points = points
      self.id = id
```

```
      self.type = t
   def __repr__(self):
      return str(self.points)
```

Чтобы воспользоваться этим классом, мы определяем переменную и присваиваем ей выражение, состоящее из имени класса и списка аргументов – как при вызове функции. Такая переменная называется объектом класса. Класс можно рассматривать как чертеж для изготовления объектов. С самим классом ничего делать нельзя, это не более чем определение, но можно создать объект класса и работать с ним.

```
>>> s = Shape([[1,1], [2,3], [3,7], [2,8]])
>>> print s
[[1, 1], [2, 3], [3, 7], [2, 8]]
```

С объектом `Shape` мало что можно сделать, потому что фигура – нечто более сложное, чем просто множество точек. Например, есть прямые, и есть многоугольники, и ведут они себя по-разному. Поэтому определяются подклассы существующего класса, уточняющие его поведение. Это называется наследованием: подкласс наследует возможности от своих родительских классов. Сначала рассмотрим подкласс `Polyline` (ломаная). Мы по-прежнему используем ключевое слово `class`, но дополнительно указываем имя родительского класса в скобках. В конструкторе (`__init__`) этого класса мы сначала вызываем конструктор родительского класса, передавая ему необходимую информацию, которая будет храниться в родительском классе. Кроме того, мы добавили в подкласс два новых атрибута: `name` и `color`. Оба атрибута получают значения по умолчанию, которые можно будет изменить позже. Мы включили также новый метод `draw`, в котором сначала вызываем функцию `Polygon` из пакета Matplotlib для создания объекта ломаной, а затем добавляем этот объект в график. Заметим, что для рисования незамкнутой ломаной параметры `closed` и `fill` функции `Polygon` нужно задать равными `False`. Необходимо также импортировать пакет Matplotlib.

```
import matplotlib.pyplot as plt
class Polyline(Shape):
   def __init__(self, points, id, name=None):
      Shape.__init__(self, points, id, "polyline")
      self.name = name
      self.color = 'grey'
   def draw(self):
      l = plt.Polygon(self.points, closed=False, fill=False,
      edgecolor=self.color)
      plt.gca().add_line(l)
```

Далее мы определим еще один подкласс, `Polygon`. В этом подклассе имеется два свойства, определяющих цвет: заливки (`fillcolor`) и обводки (`edgecolor`). В классе `Polygon` определена своя функция `draw`, учитывающая особенности многоугольника. Степень прозрачности зададим равной 0.5, воспользовавшись параметром `alpha` функции `Polygon`.

```
class Polygon(Shape):
   def __init__(self, points, id, name=None):
      Shape.__init__(self, points, id, "polygon")
      self.name = name
      self.fillcolor = 'white'
      self.edgecolor = 'grey'
   def draw(self):
      poly = plt.Polygon(self.points, closed=True,
                         fill=True,
                         facecolor=self.fillcolor,
                         edgecolor=self.edgecolor,
                         alpha=0.5)
      plt.gca().add_line(poly)
```

Наконец, создадим еще класс `Shapes`, который будет служить контейнером для хранения нескольких фигур. Его ключевая особенность – список `shapes` и функция, которая добавляет фигуры в этот список. Функция `draw` этого класса сначала вызывает функции `draw` всех фигур в списке, чтобы поместить их на рисунок (но пока еще не выводит рисунок на экран). Затем она добавляет оси координат и, наконец, вызывает функцию Matplotlib `show`, чтобы показать результат.

```
class Shapes():
   def __init__(self):
      self.shapes = []
   def add_shape(self, shape):
      self.shapes.append(shape)
   def draw(self):
      for s in self.shapes:
         s.draw()
      if len(self.shapes):
         plt.axis('scaled')
         plt.xticks([])
         plt.yticks([])
         plt.show()
```

Создадим три очень простые фигуры и нарисуем их. На рис. А.4 показан результат.

```
shapes = Shapes()
l1 = Polyline([[1,2], [5,3], [7,2]], 0)
l2 = Polyline([[6,2], [1,1], [5,5]], 1)
l2.color = 'red'
p1 = Polygon([[1,1], [2,3], [3,7], [2,8]], 3)

shapes.add_shape(l1)
shapes.add_shape(l2)
shapes.add_shape(p1)
shapes.draw()
```

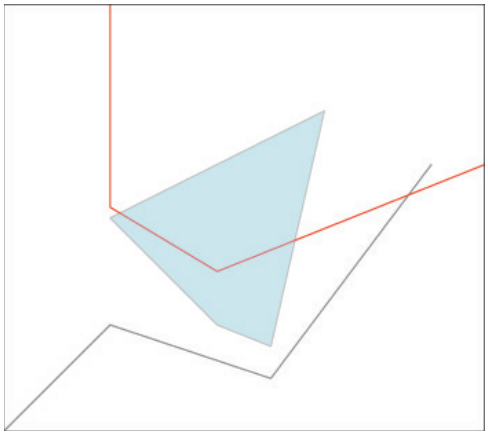

Рис. А.4 ❖ Рисование фигур с помощью Matplotlib

Приложение B

GDAL/OGR и PySAL

Для работы с геопространственными данными разработано много пакетов на Python. В этом приложении мы познакомимся с двумя библиотеками, предназначенными для разных целей. GDAL (Geospatial Data Abstraction Library) – мощная библиотека, поддерживающая широкий спектр операций с наборами геопространственных данных. GDAL проектировалась для работы с растровыми данными и поддерживает около 100 растровых форматов[1]. GDAL знаменует большой шаг вперед в развитии свободных ГИС с открытым исходным кодом. Один из компонентов GDAL, OGR, поддерживает векторные форматы данных. Акроним OGR означает OpenGIS Simple Features Reference Implementation, но на самом деле библиотека OGR не полностью совместима со спецификацией OpenGIS Simple Features Specification, разработанной консорциумом Open GIS Consortium (OGC), и потому не получила одобрения OGC. Поэтому в настоящее время акроним OGR технически не значит ничего. OGR поддерживает многочисленные векторные форматы[2], в т. ч. файлы фигур, персональные базы геоданных, ArcSDE, MapInfo, GRASS, TIGER/Line, SDTS, GML и KML. Поддерживаются также различные базы данных, включая MySQL, PostgreSQL и Oracle Spatial. Дополнительную информацию о GDAL и OGR см. на сайте[3]. Изначально GDAL/OGR разрабатывалась не как библиотека на Python, но интерфейсы к ней реализованы для многих языков программирования, в т. ч. Python. В реальности мы пользуемся написанной на Python оберткой двоичной библиотеки GDAL/OGR, что в какой-то мере отражается на производительности.

Вторая библиотека – пакет PySAL с открытым исходным кодом (Python Spatial Analysis Library), который использовался для создания широкого круга приложений для работы с пространственными данными[4]. В отличие от GDAL/OGR, библиотека PySAL написана на Python и поддерживает многочисленные приложения для анализа пространственных данных. В PySAL имеется собственная главная структура данных, поддерживающая файловый ввод-вывод. Основным компонентом PySAL являются пространственные

---

[1] Полный перечень поддерживаемых форматов см. на странице http://www.gdal.org/formats_list.html.

[2] Полный перечень поддерживаемых OGR форматов см. на странице http://www.gdal.org/ogr/ogr_formats.html.

[3] http://www.gdal.org.

[4] http://pysal.org.

веса, играющие главенствующую роль во многих методах пространственного анализа.

В этом приложении мы рассмотрим некоторые основные средства, чтобы продемонстрировать, как эти мощные библиотеки могли бы помочь в понимании и реализации многих описанных в книге алгоритмов. В основном приложение состоит из кода, снабженного подробными комментариями. Некоторые строки включены в педагогических целях, поэтому не всегда необходимы на практике. Но они создают контекст для освоения функциональности GDAL/ORG и PySAL. Мы не собираемся пояснять каждую строчку, поскольку сложностей в понимании не должно возникнуть (особенно при наличии развернутых комментариев к коду). Дополнительные сведения об устройстве и применении библиотеки GDAL/OGR есть в онлайновой книге «Python GDAL/ORG Cookbook»[1]. Для PySAL также имеется пособие, в котором описаны детали работы с библиотекой.

Библиотеки GDAL/OGR и PySAL способны на очень многое, поскольку предоставляют средства для преобразования геопространственных в любой формат. В них реализованы почти все рассмотренные в книге алгоритмы. Следует воспользоваться этой возможностью, чтобы протестировать и исследовать эти алгоритмы на собственных данных.

Как GDAL/OGR, так и PySAL можно установить в разных операционных системах. Здесь мы не станем обсуждать процедуру установки, потому что она подробно описана на сайтах библиотек.

B.1. OGR

Использование OGR обычно начинается с приведенных ниже строк, в которых показано, как OGR работает с файлом фигур. В следующем примере предполагается, что в каталоге `data` на верхнем уровне текущего каталога расположен файл `uscnty48area.shp`, содержащий описание многоугольника. Для работы нам нужно знать путь к этому файлу. Здесь мы будем заниматься файлами фигур для многоугольников. Для файлов фигур других типов (например, ломаных) процедура похожа, но детали несколько отличаются. Пример использования файла фигур для ломаной (с очертаниями материков) был приведен в главе 2. Выражение в последней строке приведенного ниже кода должно быть равно `False`, это означает, что файл фигур был прочитан успешно.

```
from osgeo import ogr                    # импортировать OGR
drvName = "ESRI Shapefile"               # тип драйвера/файла
driver = ogr.GetDriverByName(drvName)    # драйвер формата файла фигур
fname = '../data/uscnty48area.shp'       # путь к файлу
vector = driver.Open(fname, 0)           # 0 - читать, 1 - записывать
vector is None                           # проверить, был ли открыт файл
```

[1] http://pcjericks.github.io/py-gdalogr-cookbook/.

B.1.1. Атрибуты

Переменная `vector` в приведенном выше коде позволяет получить доступ к информации в файле фигур. Для доступа к информации разных типов нужно выполнить общую процедуру: спуститься по иерархии файл > слой > свойство. Файл фигур может содержать только один слой, который содержит несколько свойств. Показанный ниже код получает атрибуты, ассоциированные со свойством.

```
layer = vector.GetLayer(0)            # в файле фигур только 0
layer.GetFeatureCount()               # число свойств
feature = layer.GetFeature(0)         # первое свойство
feature.items()                       # атрибуты свойства
feature.GetFieldCount()               # число полей
fldnref = feature.GetFieldDefnRef(7)  # получить ссылку на поле
fldn = fldnref.GetName()              # получить имя поля
feature.GetField(fldn)                # получить значения поля по имени
feature.GetField(7)                   # получить значения поля по идентификатору
```

Мы можем написать коротенькую функцию (`getattr.py`), которая возвращает список значений атрибута в файле фигур. Эта функция понадобится нам в разделе B.3.

```
from osgeo import ogr
import sys
def get_shp_attribute_by_name(shpfname, attrname):
   """
   Получить из файла фигур значения атрибута по имени столбца
   """
   driver = ogr.GetDriverByName("ESRI Shapefile")
   vector = driver.Open(shpfname, 0)
   layer = vector.GetLayer(0)
   f = layer.GetFeature(0)
   val = []
   for i in range(layer.GetFeatureCount()):
      f = layer.GetFeature(i)
      val.append(f.GetField(attrname))
   return val
```

B.1.2. Геометрия и координаты

Теперь посмотрим, как получить геометрическое описание и координаты из только что загруженного файла фигур. Мы покажем только получение вершин одного многоугольника, но, конечно, таким же образом можно в цикле получить вершины всех многоугольников, как будет продемонстрировано ниже.

```
layer.GetExtent()                # экстент слоя
feature = layer.GetFeature(19)   # получить свойство
geom = feature.GetGeometryRef()  # ссылка на геометрию свойства
```

```
geom.GetEnvelope()              # получить огибающую (экстент)
geom.Centroid()                 # центроид многоугольника
geom.GetGeometryName()          # "POLYGON"
geom.GetPoint(0)                # ERR 6 в фигуре есть кольца
geom.GetGeometryCount()         # в этой фигуре 1 кольцо
ring = geom.GetGeometryRef(0)   # получить первое кольцо
ring.GetPointCount()            # в нем 31 точка
p0 = ring.GetPoint(0)           # первая точка в кольце
pn = ring.GetPoint(30)          # последняя точка в кольце
p0 == pn                        # True: они совпадают
pn==ring.GetPoint(29)           # False
```

B.1.3. Проецирование точек

Мы можем перейти от географической системы координат (ГСК) – широта и долгота – к системе координат проекции. Пусть имеются координаты нескольких точек в Северном Камеруне, записанные в формате WGS84, и мы хотим преобразовать их в формат зоны 33N универсальной поперечной проекции Меркатора (UTM). Для этого мы воспользуемся библиотекой OSR (OGRSpatialReference) – модулем OGR для работы с системами пространственной привязки. OSR основана на библиотеке картографических проекций PROJ.4[1]. Процедура начинается с определения исходной системы координат, которая в приведенном ниже коде названа `gcs`. Затем мы определяем целевую систему координат, названную ниже `utmsr`. Есть разные способы определить систему координат проекции, мы выберем из них самый простой, воспользовавшись кодом EPSG (European Petroleum Survey Group), однозначно идентифицирующим зону UTM 33N. Имея такие две системы координат, мы можем определить преобразование с помощью функции OSR `CoordinateTransformation`, это преобразование названо ниже `coordTransPcsUtm33`. Известные координаты четырех точек в Камеруне хранятся в списке списков, в котором строка в каждом внутреннем списке является названием места. Сначала нужно преобразовать эти координаты в точечную геометрию в OGR, а затем просто вызвать функцию `Transform` для выполнения преобразования координат.

```
from osgeo import ogr                   # импортировать OGR
import osr                              # импортировать OSR
#
gcs = osr.SpatialReference()            # определить ГСК
gcs.SetGeogCS("My GCS", "WGS_1984",
              "WGS84 Spheroid",
              osr.SRS_WGS84_SEMIMAJOR,
              osr.SRS_WGS84_INVFLATTENING,
              "Greenwich", 0.0, "degree")
utmsr = osr.SpatialReference()          # пустая пространственная привязка
utmsr.ImportFromEPSG(32633)             # код EPSG: UTM 33N
                                        # из gcs в utmsr
```

[1] http://trac.osgeo.org/proj/.

```
coordTransPcsUtm33 = osr.CoordinateTransformation(gcs, utmsr)
                                        # четыре точки, координаты в ГКС
places = [ ["Mindif", 14.433333, 10.4],  # Name, lon, lat
           ["Pette", 14.5, 10.966667 ],
           ["Moulvoudaye", 14.85385, 10.40588],
           ["Diamare", 10.89510, 14.80438]]
point = ogr.Geometry(ogr.wkbPoint)      # пустая точечная геометрия
for p in places:
   point.AddPoint(p[1], p[2])           # X (долгота), Y (широта)
   point.Transform(coordTransPcsUtm33)  # преобразовать
   print p[0], point                    # напечатать результат
```

B.1.4. Проецирование и запись геопространственных данных

Вернемся к нашему файлу фигур, который сейчас записан в координатах ГКС, и преобразуем его в равновеликую коническую проекцию Альберса, которая широко используется в США, вытянутых в широтном направлении. Наш файл фигур имеет расширение .prj, и информацию о пространственной привязке можно читать непосредственно из него. Идея заключается в том, чтобы взять каждую точку, преобразовать ее в новую систему координат и записать в кольцо, которое, в свою очередь, добавляется в свойство многоугольника. Преобразованные данные записываются в новый файл фигур.

Многоугольник общего вида в файле фигур может иметь дырки, каждая дырка и граничный многоугольник называется кольцом. Однако во многих округах, представленных в наших данных, имеются также острова посередине реки или озера, которые не являются дырками. Такие округа организуются как мультимногоугольники. Как следует из названия, мультимногоугольник содержит набор многоугольников, каждый из которых может содержать дырки. Чтобы упростить обработку таких данных, мы написали функцию `trans_rings`, она выбирает из входного описания геометрии кольца, которые должны быть многоугольниками, возможно, с дырками, но не мультимногоугольниками. (Наличие дырок не влияет на работу этой функции, потому что дырки и граничные многоугольники рассматриваются как кольца, и многоугольник должен содержать по крайней мере одно кольцо.) Эта функция возвращает новый многоугольник, содержащий преобразованные точки. Затем этот многоугольник добавляется в свойство нового файла фигур. Если свойство является простым многоугольником (с дырками или без), то эта функция вызывается только один раз и создает новый многоугольник. Если же свойство является мультимногоугольником, то функцию придется вызвать несколько раз, по разу для каждого частичного многоугольника. Перед добавлением каждого нового многоугольника мы создаем новое свойство, в которое этот многоугольник и добавляется.

```
import os                               # импортировать модуль OS
from osgeo import ogr                   # импортировать OGR
import osr                              # импортировать OSR
```

```
#
drvName = "ESRI Shapefile"                  # имя драйвера
driver = ogr.GetDriverByName(drvName)       # драйвер файла фигур
fname = '../data/uscnty48area.shp'          # имя входного файла
vector = driver.Open(fname, 0)              # открыть входной файл
layer = vector.GetLayer(0)                  # для файлов фигур 0
ofname = 'uscnty48_proj.shp'                # имя выходного файла
if os.path.exists(ofname):                  # если файл существует
   driver.DeleteDataSource(ofname)          # удалить его
#
ovector = driver.CreateDataSource(ofname)   # выходной вектор
olayer = ovector.CreateLayer('POLYGONS',    # точечный файл фигур
                             geom_type=ogr.wkbPolygon)
fieldDefn = ogr.FieldDefn('id',
                          ogr.OFTInteger)   # целочисленный ид поля
olayer.CreateField(fieldDefn)               # поле выходного файла
featureDefn = olayer.GetLayerDefn()         # свойство для последующего исп-ния
gcs = layer.GetSpatialRef()                 # географическая система координат
pcs = osr.SpatialReference()                # система координат пустой проекции
pcs.SetProjCS("Albers Conic Equal Area")    # США, вытянутые в широтном напр.
pcs.SetACEA(29.5, 45.5,                     # стд параллели: 29.5, 45.5
            23, -96, 0, 0)                  # средний меридиан : 23 с.ш. 96 з.д.
pcs.SetWellKnownGeogCS("NAD83")             # задать реперную точку (NAD83)
# преобразовать из gcs в pcs
coordTrans = osr.CoordinateTransformation(gcs, pcs)

pcs.MorphToESRI()                           # преобразовать pcs в .prj
fprj = open(ofname.strip('.shp')+'.prj', 'w')
fprj.write(pcs.ExportToWkt())
fprj.close()

def trans_rings(geom):                      # возвращает многоугольник
   polygon = ogr.Geometry(
   ogr.wkbPolygon)                          # пустой многоугольник
   point = ogr.Geometry(ogr.wkbPoint)       # временная точка
   for ring in geom:                        # цикл по всем кольцам
      ring1 = ogr.Geometry(
      ogr.wkbLinearRing)                    # создать пустое кольцо
      for p in ring.GetPoints():            # цикл по точкам кольца
         point.AddPoint(p[0], p[1])         # создать точку
         point.Transform(coordTrans)        # преобразовать GCS в PCS
         ring1.AddPoint(
            point.GetX(),
            point.GetY(),
            point.GetZ())                   # создать новое кольцо
   polygon.AddGeometry(ring1)               # добавить кольцо в многоугольник
   return polygon

k = 0
for f in layer:                             # цикл по свойствам слоя
   geom = f.GetGeometryRef()                # получить геометрию свойства
   ofeat = ogr.Feature(featureDefn)         # выходное свойство
   ofeat.SetField('id', k)                  # задать ид свойства
```

```
   geomtype = geom.GetGeometryName()
   if geomtype == "MULTIPOLYGON":
      for geom1 in geom:
         polygon = trans_rings(geom1)
         ofeat.SetGeometry(polygon)          # сделать ofeat многоугольником
         olayer.CreateFeature(ofeat)         # присоединить ofeat к слою
   elif geomtype == "POLYGON":
      polygon = trans_rings(geom)
      ofeat.SetGeometry(polygon)             # то же, что и выше
      olayer.CreateFeature(ofeat)            # то же, что и выше
   else:
      pass                                   # не тип многоугольника
   k += 1

vector.Destroy()                             # закрыть входной файл фигур
ovector.Destroy()                            # закрыть выходной файл фигур
```

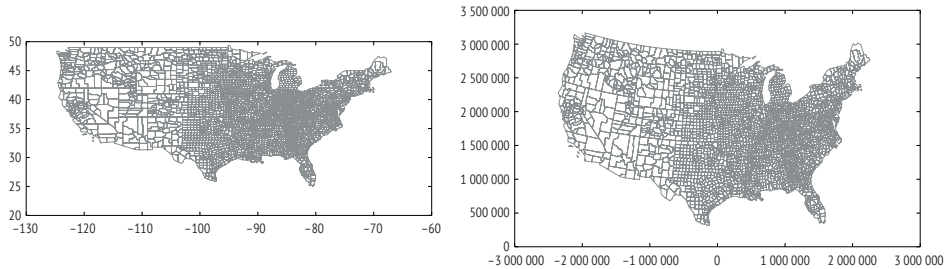

Рис. В.1 ❖ Проецирование пространственных данных.
Слева: исходные данные в ГСК.
Справа: данные в равновеликой конической проекции Альберса

Результат изображен на рис. В.1. Обе карты нарисованы с помощью Matplotlib следующей Python-программой (draw_shp_polygons.py).

```
from osgeo import ogr
import matplotlib.pyplot as plt
import sys

def plot_rings(geom):                        # нарисовать
   ptsx = []
   ptsy = []
   for ring in geom:                         # цикл по всем кольцам
      points = ring.GetPoints()              # получить точки кольца
      ptsx += [p[0] for p in points]
      ptsy += [p[1] for p in points]
   plt.plot(ptsx, ptsy, color='grey')

   drvName = "ESRI Shapefile"
   driver = ogr.GetDriverByName(drvName)     # драйвер файла фигур

   driver = ogr.GetDriverByName("ESRI Shapefile")
   if len(sys.argv) == 2:
```

```
      fname = sys.argv[1]                  # имя входного файла
   else:
      print "Usage:", sys.argv[0], "FILE.shp"
      sys.exit()

   vector = driver.Open(fname, 0)          # открыть входной файл
   layer = vector.GetLayer(0)              # в файле фигур всегда 0

   for f in layer:
      geom = f.GetGeometryRef()            # геометрия свойства
      geomtype = geom.GetGeometryName()
      if geomtype == "MULTIPOLYGON":
         for geom1 in geom:
            plot_rings(geom1)
      elif geomtype == "POLYGON":
            plot_rings(geom)
      else:
         pass                              # не тип многоугольника

f.Destroy()                               # удалить входное свойство
vector.Destroy()                          # закрыть файл фигур
plt.axis('scaled')
plt.savefig('us48prj.eps', bbox_inches='tight', pad_inches=0)
plt.show()
```

B.1.5. Матрица смежности

Последний наш проект с использованием OGR – создание матрицы смежности для файла фигур, содержащего описание многоугольников. Процедура простая: для каждой пары многоугольников мы проверяем, верно ли, что они соприкасаются или один находится внутри другого. Для этого используются функции OGR `Touches` и `Contains`. Чтобы ускорить процесс, мы включили функцию `env_touch`, которая проверяет, соприкасаются ли ограничивающие прямоугольники (огибающие) двух многоугольников. Можно написать функцию `geom_touch`, проверяющую, соприкасаются ли два геометрических объекта в указанном (в параметре `n0`) числе точек.

Функция `adjacency_matrix` принимает три параметра: путь к файлу фигур, формат результата (`M` – матрица, `L` – список) и количество общих точек, необходимых для того, чтобы многоугольники считались смежными (по умолчанию 1). Для тестирования этой функции мы взяли файл фигур с описанием округов США, а результирующую матрицу сохранили в pickle-файле. В Python pickle – это стандартный способ сериализации сложных объектов, позволяющий сохранить их в текстовом файле для будущего использования. Сохраненный здесь pickle-файл используется в основном тексте книги (раздел 9.2) при вычислении индекса *I* Морана.

```
from osgeo import ogr
import numpy as np
import pickle

XMIN = 0 # envelope[0]: xmin
```

```
XMAX = 1 # envelope[1]: xmax
YMIN = 2 # envelope[2]: ymin
YMAX = 3 # envelope[3]: ymax

def env_touch(e1, e2):
   if e1[XMAX]<e1[XMIN] or e1[XMIN]>e2[XMAX] or\
      e1[YMAX]<e2[YMIN] or e1[YMIN]>e2[YMAX]:
      return False
   return True

def geom_share(g1, g2, n0):
   pts1 = []
   if g1 is None or g2 is None:
      return False
   for g in g1:
      if g.GetGeometryCount()>0:
         for gg in g:
            pts1.extend(gg.GetPoints())
      else:
         pts1.extend(g.GetPoints())
   pts2 = []
   for g in g2:
      if g.GetGeometryCount()>0:
         for gg in g:
            pts2.extend(gg.GetPoints())
      else:
         pts2.extend(g.GetPoints())
   np = 0
   for p1 in pts1:
      for p2 in pts2:
         if p1==p2:
            np+=1
         if np>=n0:
            return True
   return False

def adjacency_matrix(shpfname, output="M",
                     num_shared_points=1):
   driver = ogr.GetDriverByName("ESRI Shapefile")
   vector = driver.Open(shpfname, 0)
   layer = vector.GetLayer(0)
   n = layer.GetFeatureCount()
   if output=="M":
      adj = np.array([[0]*n for x in range(n)])
   elif output=="L":
      adj = []
   else:
      return None
   for i in range(n):
      feature1 = layer.GetFeature(i)
      geom1 = feature1.GetGeometryRef()
      env1 = geom1.GetEnvelope()
      for j in range(i):
```

```
            feature2 = layer.GetFeature(j)
            geom2 = feature2.GetGeometryRef()

            env2 = geom2.GetEnvelope()
            if not env_touch(env1, env2):
               continue
            is_adj = False
            if geom1.Touches(geom2):
               if geom_share(geom1, geom2,
                             num_shared_points):
                  is_adj = True
               elif geom1.Contains(geom2):
                  is_adj = True
               if is_adj:
                  if output=="M":
                     adj[i][j] = adj[j][i] = 1
                  elif output=="L":
                     adj.append([i, j])
               else: # не определено
                  pass
      return adj

if __name__ == "__main__":
      shpfname = '../data/uscnty48area.shp'
      shpadj = adjacency_matrix(shpfname)
      pickle.dump(shpadj, open('uscnty48area.adj.pickle', 'w'))
```

B.2. GDAL

Теперь обратимся к библиотеке GDAL и рассмотрим несколько способов доступа к растровым данным. В этом примере для демонстрации используется Национальная база данных о земляном покрове (National Land Cover Database – NLCD) за 2011 год. Как и в OGR, для доступа к растровым данным нужен драйвер, но в GDAL необходимо также вызвать функцию `Register`, перед тем как открывать растровый файл. Имена драйверов приведены по адресу http://www.gdal.org/formats_list.html. База данных NLCD хранится в формате Erdas Imagine (.img), драйвер GDAL для этого формата называется HFA. Наша цель – загрузить данные NLCD в массив NumPy с помощью функции `ReadAsArray`. После того как данные представлены в виде массива – это естественный формат для растровых данных, – мы можем обработать их самыми разными способами, например вычислить индекс распространенности (раздел 9.4).

```
from osgeo import gdal                    # импортировать GDAL
driver = gdal.GetDriverByName('HFA')      # драйвер формата Erdas Imagine
driver.Register()                         # зарегистрировать драйвер
fname = "nlcd_2011_landcover_2011_edition_2014_03_31.img"
raster = gdal.Open(fname,
                   gdal.GA_ReadOnly)
```

```
raster is not None                      # файл загружен?
ncols = raster.RasterXSize              # размер растра по оси x
nrows = raster.RasterYSize              # размер растра по оси y
raster.RasterCount                      # число полос
geotransform = raster.GetGeoTransform() # инфо о пространственной привязке
ul_x = geotransform[0]                  # абсцисса левого верхнего угла
x_res = geotransform[1]                 # разрешение по оси запад-восток
geotransform[2]                         # поворот по X
ul_y = geotransform[3]                  # ордината левого верхнего угла
geotransform[4]                         # поворот по Y
y_res = geotransform[5]                 # разрешение по оси север-юг
band = raster.GetRasterBand(1)          # получить полосу (нумерация с 1)
x = 835000                              # X (центр Огайо)
y = 1900000                             # Y (центр Огайо)
Xoff = int((x - ul_x) / x_res)          # смещение по X (110934)
Yoff = int((y - ul_y) / y_res)          # смещение по Y (47000)
data1 = band.ReadAsArray(Xoff, Yoff,
                         1, 1)          # получить фрагмент данных 1×1
data1[0, 0]                             # значение в (0, 0)
data=band.ReadAsArray(0,0,ncols,nrows)  # создать массив питру
data[47000, 110934]                     # значение в точке
band=None                               # освободить память
data=None
```

B.3. PySAL

Библиотека PySAL охватывает ряд задач пространственного анализа, и не только (Rey and Anselin, 2010). Мы продемонстрируем, как воспользоваться ей для вычисления индекса *I* Морана для площадного набора данных. PySAL предоставляет ряд способов задать тип пространственных весов. Мы остановимся на двоичных весах, когда вес пары многоугольников равен 1 (смежные) или 0 (несмежные). Существует также несколько способов вычислить смежность многоугольников, мы будем использовать критерий ферзя, при котором многоугольники считаются смежными, если имеют одну или больше общих точек (с другой стороны, критерий ладьи требует, чтобы было общее ребро). При таком задании параметров мы сможем сравнить результаты PySAL с теми, что дает код, написанный нами в книге.

```
import pysal                            # импортировать PySAL
import numpy as np                      # импортировать NumPy
fname = '../data/franklin.shp'          # данные об округе Франклин
w = pysal.queen_from_shapefile(fname)   # соседи и веса
w.neighbors[0]                          # соседи первого многоугольника
sum([len(w.neighbors[i])
   for i in range(w.n)])                # двунаправленные ребра
dbname = '../data/franklin.dbf'         # таблица атрибутов
db = pysal.open(dbname, 'r')            # получить атрибуты
db.header                               # показать имена атрибутов
y = np.array(db.by_col('Franklin_b'))   # использовать столбец
```

```
mi = pysal.Moran(y, w,
                 transformation='B')    # индекс I Морана, двоичные веса
mi.I                                    # 0.31583739278406564
mi.EI                                   # ожидаемый I

from adjacency_matrix import *          # код вычисления матрицы смежности
from getattr import *
import sys
sys.path.append('../spatialanalysis')
from moransi2 import *                  # импортировать moransi2
adj = adjacency_matrix(fname,           # вычислить матрицу смежности
                       output="L")      # вывести в виде списка
len(adj)                                # неориентированные ребра
y1 = get_shp_attribute_by_name(
     fname, "Franklin_b")               # получить атрибуты
moransi2(y1, adj)                       # 0.3158373927840651
```

В разделе 9.2 мы познакомились с вычислением индекса Морана на основе матрицы (или списка) смежности, что эквивалентно двоичным весам, определенным по критерию ферзя. Приведенный выше код также сравнивает этот результат с тем, что был получен ранее написанной программой, – совпадение почти полное (интересующиеся читатели могут изучить исходный код и объяснить, чем вызвано очень небольшое различие). Чтобы еще раз подтвердить правильность результатов, мы можем применить к тем же данным пакет spdep, написанный на R. С той точностью, которую обеспечивает этот пакет, результат равен 0.3158373928, что согласуется с результатами, которые дают программы на Python.

Приложение C

Список программ

Приведенные в этой книге программы по большей части хранятся в отдельных файлах, чтобы их можно было импортировать как модули или выполнять из командной строки. Для большинства программ заведены отдельные каталоги, поэтому если нам нужен модуль из другого каталога, необходимо добавить путь к нему, воспользовавшись функцией `sys.path.append` из модуля `sys`. В предположении, что мы находимся в каталоге `interpolation` и хотим импортировать модуль `point.py` из каталога `geom`, надо будет включить такие три строчки:

```
import sys
sys.path.append('../geom')
from point import *
```

В табл. C.1 перечислены все именованные файлы в том порядке, в каком они встречаются в тексте. Помимо файлов, в тексте встречаются фрагменты кода; они здесь не упоминаются, потому что предназначены в основном для интерактивного выполнения. В таблицу также включены наборы данных, используемые в книге.

Таблица C.1. Программы и данные

Каталог	Имя файла	Раздел
geom	point.py	2.1
	spherical_distance.py	2.2
	point2line.py	2.3
	centroid.py	2.4
	sideplr.py	2.5
	linesegment.py	2.6
	intersection.py	2.7
	point_in_polygon.py	2.8
	point_in_polygon_winding.py	2.9
	neville.py	2.10
	transform1.py	2.11
	worldmap.py	2.12
	test_projection.py	2.13
	transform2.py	2.14
	line_seg_eventqueue.py	3.2
	line_seg_intersection.py	3.3
	test_line_seg_intersection.py	3.4
	overlay.py	3.5
	test_overlay_1.py	3.6

Таблица С.1 (продолжение)

Каталог	Имя файла	Раздел
indexing	kdtree1.py	5.1
	kdtree2a.py	5.2
	kdtree2b.py	5.3
	kdtree3.py	5.4
	prkdtree1.py	5.5
	prkdtree2.py	5.6
	prkdtree3.py	5.7
	kdtree_performance.py	5.8
	quadtree1.py	6.1
	pointquadtree1.py	6.2
	pointquadtree2.py	6.3
	pointquadtree3.py	6.4
	xdcel.py	7.1
	extent.py	7.2
	pmquadtree.py	7.3
	pm1quadtree.py	7.4
	pm2quadtree.py	7.5
	pm3quadtree.py	7.6
	rtree1.py	7.7
	rtree2.py	7.8
	use_rtree.py	7.9
interpolation	idw.py	8.1
	read_data.py	8.2
	prepare_interpolation_data.py	8.3
	test_idw.py	8.4
	idw_cross_validation.py	8.5
	semivariance.py	8.6
	covariance.py	8.8
	fitsemivariance.py	8.9
	test_fitsemivariance.py	8.10
	okriging.py	8.11
	ordinary_kriging_test.py	8.12
	skriging.py	8.13
	simple_kriging_test.py 8.14	8.14
	interpolate_surface.py 8.15	8.15
	kriging_cross_validation.py 8.16	8.16
	midpoint2d.py	8.17
spatialanalysis	nnd.py	9.1
	nnd_monte_carlo.py	9.2
	oval_trees.py	9.3
	test_oval_trees_nnd.py	9.4
	kfunction.py	9.5
	test_oval_trees.py	9.6
	moransi.py	9.7
	moransi2.py	9.8
	test_moransi.py	9.9
	test_moransi_mc.py	9.10
	test_moransi_grid.py	9.11
	kmeans.py	9.12
	contagion.py	9.13
	test_contagion.py	9.14
	test_contagion_kernel.py	9.15

Таблица С.1 (окончание)

Каталог	Имя файла	Раздел
networks	network2listmatrix.py bfs.py dfs.py dijkstra.py gateway.py allpairdist.py allpairpaths.py	10.2 10.3 10.4 10.5 10.6 10.7 10.8
optimization	disc.py cp.py elzinga_hearn.py welzl.py test_1center.py pmed_lpformat.py pmed_greedy.py teitz_bart.py test_orlib_pmed.py simulated_annealing.py test_orlib_pmed_simann.py	11.1 11.2 11.3 11.4 11.5 11.6 12.1 12.3 12.4 12.6 12.7
gdal	getattr.py transform_cameroon_points.py transform_coordinate_systems.py draw_shp_polygons.py adjacency_matrix.py gdal1.py pysal1.py	B.1.1 B.1.3 B.1.4 B.1.4 B.1.5 B.2 B.3
data	uscnty48area.* franklin.* necoldem.dat necoldem250.dat	
data/orlib	pmed*.orlib pmed*.distmatrix	

Список литературы

Aluru, S. (2005). Quadtrees and octrees. In D. Metha and S. Sahni (eds), *Handbook of Data Structures and Applications*. Boca Raton, FL: Chapman & Hall/CRC.

Anselin, L. (1995). Local indicators of spatial association – LISA. *Geographical Analysis 27* (2), 93–115.

Bailey, T. C. and A. C. Gatrell (1995). *Interactive Spatial Data Analysis*. Harlow, UK: Pearson Education.

Bayer, R. and E. McCreight (1972). Organization and maintenance of large ordered indexes. *Acta Informatica 1*, 173–189.

Beasley, J. E. (1985). A note on solving large *p*-median problems. *European Journal of Operational Research 21*, 270–273.

Beckmann, N., H. Kriegel, R. Schneider, and B. Seeger (1990). The R*-tree: An efficient and robust method for points and rectangles. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Atlantic City, pp. 322–331. New York: Association for Computing Machinery.

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM 18* (9), 509.

Bentley, J. L. and T. A. Ottmann (1979). Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers C 28* (9), 643–647.

Bentley, J. L., D. F. Stanat, and J. E. H. Williams (1977). The complexity of finding fixed-radius near neighbors. *Information Processing Letters 6* (6), 209–212.

Boots, B. N. and A. Getis (1988). *Point Pattern Analysis*. Newbury Park, CA: Sage.

Burrough, P. A. and R. A. McDonnell (1998). *Principles of Geographical Information Systems*. Oxford: Oxford University Press.

Carr, J. R. (1995). *Numerical Analysis for the Geological Sciences*. Englewood Cliffs, NJ: Prentice Hall.

Cerny, V. (1985). Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Application 45* (1), 41–51.

Chakraborty, R. K. and P. K. Chaudhuri (1981). Note on geometrical solution for some minimax location problems. *Transportation Science 15*, 164–166.

Chan, T. M. (1994). A simple trapezoid sweep algorithm for reporting red/blue segment intersections. In *Canadian Conference on Computational Geometry*, Saskatoon, Saskatchewan, Canada, pp. 263–268.

Chrystal, G. (1885). On the problem to construct the minimum circle enclosing *n* given points in the plane. *Proceedings of Edinburgh Mathematical Society 3*, 30–33.

Clark, P. J. and F. C. Evans (1954). Distance to nearest neighbor as a measure of spatial relationships in populations. *Ecology 35*, 445–453.

Cliff, A. D. and J. K. Ord (1981). *Spatial Processes: Models and Applications*. London: Pion.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein (2001). *Introduction to Algorithms* (2nd edn). Cambridge, MA: MIT Press.

Cressie, N. (1990). The origins of kriging. *Mathematical Geology 22*, 239–253.

Cressie, N. A. C. (1991). *Statistics for Spatial Data*. New York: John Wiley & Sons.

Daskin, M. S. (1995). *Network and Discrete Location: Models, Algorithms, and Applications*. New York: John Wiley & Sons.

de Berg, M., M. van Kreveld, M. Overmars, and O. Schwarzkopf (1998). *Computational Geometry: Algorithms and Applications* (2nd edn). Berlin: Springer.

Densham, P. J. and G. Rushton (1992). A more efficient heuristic for solving large *p*-median problems. *Papers in Regional Science 41* (3), 307–329.

Deza, M. M. and E. Deza (2010). *Encyclopedia of Distances* (2nd edn). Berlin: Springer.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik 1*, 269–271.

Dorigo, M. and L. M. Gambardella (1997). Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation 1* (1), 53–66.

Drezner, Z. (ed.) (1995). *Facility Location: A Survey of Applications and Methods*. New York: Springer.

Drezner, Z. and S. Shelah (1987). On the complexity of the Elzinga–Hearn algorithm for the 1-center problem. *Mathematics of Operations Research 12* (2), 255–261.

Dueck, G. and T. Scheuer (1990). Threshold accepting: A general purpose optimization algorithm appearing superior to simulated annealing. *Journal of Computational Physics 90*, 161–175.

Elzinga, J. and D. W. Hearn (1972). Geometrical solutions for some minimax location problems. *Transportation Science 6*, 379–394.

Finkel, R. and J. Bentley (1974). Quad trees: A data structure for retrieval on composite keys. *Acta Informatica 4* (1), 1–9.

Floyd, R. W. (1962). Algorithm 97: Shortest path. *Communications of the ACM 5* (6), 345.

Fotheringham, A. S., C. Brunsdon, and M. Charlton (2000). *Quantitative Geography: Perspectives on Spatial Data Analysis*. London: Sage.

Fournier, A., D. Fussell, and L. Carpenter (1982). Computer rendering of stochastic models. *Communication of the ACM 25* (6), 371–384.

Frank, A. U. (1987). Overlay processing in spatial information systems. In *Proceedings, Eighth International Symposium on Computer-Assisted Cartography (Auto-Carto 8)*, pp. 12–31. Baltimore, MD: ASPRS, ACSM.

Franklin, W. R., C. Narayanaswami, M. Kankanhalli, D. Sun, M.-C. Zhou, and P. Y. Wu (1989). Uniform grids: A technique for intersection detection on serial and parallel machines. In *Ninth International Symposium on Computer-Assisted Cartography (Auto-Carto 9)*, pp. 100–109. Baltimore, MD: ASPRS, ACSM.

Geary, R. C. (1954). The contiguity ratio and statistical mapping. *Incorporated Statistician 5*, 115–141.

Gendreau, M., G. Laporte, and F. Semet (1997). Solving an ambulance location model by tabu search. *Location Science 5* (2), 75–88.

Getis, A. and J. K. Ord (1992). The analysis of spatial association by use of distance statistics. *Geographical Analysis 24* (3), 189–206.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley.

Goovaerts, P. (1997). *Geostatistics for Natural Resources Evaluation*. Oxford: Oxford University Press.

Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD 84, Boston, pp. 47–57. New York: ACM.

Haines, E. (1994). Point in polygon strategies. In P. S. Heckbert (ed.), *Graphics Gems IV*, pp. 24–46. San Francisco: Morgan Kaufmann.

Hansen, P. and N. Mladenovič (1997). Variable neighborhood search for the *p*-median. *Location Science 5* (4), 207–226.

Healey, R. G., S. Sowers, B. M. Gittings, and M. J. Mineter (1998). *Parallel Processing Algorithms for GIS*. London: Taylor and Francis.

Huang, C.-W. and T.-Y. Shih (1997). On the complexity of point-in-polygon algorithms. *Computers & Geociences 23* (1), 109–118.

Ipbuker, C. (2004). Numerical evaluation of the Robinson projection. *Cartography and Geographic Information Science 31* (2), 79–88.

Jordan, C. (1887). *Cours d'analyse de l'École polytechnique*. Paris: Gauthier-Villars.

Kamel, I. and C. Faloutsos (1994). Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pp. 500–509. San Francisco: Morgan Kaufmann.

Kirkpatrick, S., C. D. Gelatt, and M. P. Vecchi, Jr. (1983). Optimization by simulated annealing. *Science 220*, 671–680.

Krige, D. G. (1951). A statistical approach to some basic mine valuation problems on the Witwatersrand. *Journal of the Chemical and Metallurgical Mining Society of South Africa 52*, 119–139.

Krzanowski, R. M. and J. Raper (2001). *Spatial Evolutionary Modeling*. Oxford: Oxford University Press.

Lee, D. T. and C. K. Wong (1977). Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica 9* (1), 23–29.

Lombard, K. and R. L. Church (1993). The gateway shortest path problem: Generating alternative routes for a corridor location problem. *Geographical Systems 1* (1), 25–45.

Mandelbrot, B. B. (1967). How long is the coast of Britain? Statistical self-similarity and fractional dimension. *Science 155*, 636–638.

Mandelbrot, B. B. (1977a). *The Fractal Geometry of Nature*. San Francisco: W. H. Freeman.

Mandelbrot, B. B. (1977b). *Fractals: Form, Chance, and Dimension*. San Francisco: W. H. Freeman.

Manolopoulos, Y., A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis (2006). *R-Trees: Theory and Applications*. London: Springer.

Matheron, G. (1963). Principles of geostatistics. *Economic Geology 58*, 1246–1266.

McGarigal, K. and B. J. Marks (1995). FRAGSTATS: Spatial pattern analysis program for quantifying landscape structure. Technical Report PNW-GTR-351, U.S. Department of Agriculture, Forest Service, Pacific Northwest Research Station.

Meagher, D. (1980). Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-D objects by computer. Technical Report IPL-TR-80-111, Rensselaer Polytechnic Institute.

Meagher, D. (1982). Geometric modeling using octree encoding. *Computer Graphics and Image Processing 19*, 129–147.

Michalewicz, Z. and D. B. Fogel (2000). *How to Solve It: Modern Heuristics*. Berlin: Springer.

Miller, H. J. and S.-L. Shaw (2001). *Geographic Information Systems for Transportation: Principles and Applications*. New York: Oxford University Press.

Moran, P. A. P. (1950). Notes on continuous stochastic phenomena. *Biometrika 37* (1), 17–23.

O'Neill, R., J. Krummel, R. Gardner, G. Sugihara, B. Jackson, D. DeAngelis, B. Milne, M. Turner, B. Zygmunt, S. Christensen, V. Dale, and R. Graham (1988). Indices of landscape pattern. *Landscape Ecology 1* (3), 153–162.

O'Rourke, J. (1998). Point in polygon. In *Computational Geometry in C* (2nd edn), pp. 279–285. Cambridge: Cambridge University Press.

Peitgen, H.-O., H. Jurgens, and D. Saupe (1992). *Chaos and Fractals: New Frontiers of Science*. New York: Springer.

Peitgen, H.-O. and D. Saupe (1988). *The Science of Fractal Images*. New York: Springer.

Plastria, F. (2002). Continuous covering location problems. In Z. Drezner and H. W. Hamacher (eds), *Facility Location: Applications and Theory*, pp. 37–79. Berlin: Springer.

Press, W. H., S. A. Teukosky, W. T. Vetterling, and B. P. Flannery (2002). *Numerical Recipes in C++* (2nd edn). Cambridge: Cambridge University Press.

Resende, M. G. C. and R. F. Werneck (2003). On the implementation of a swap-based local search procedure for the *p*-median problem. In R. E. Ladner (ed.), *Proceedings of the Fifth Workshop on Algorithm Engineering and Experiments*, pp. 119–127. Philadelphia: SIAM.

Resende, M. G. C. and R. F. Werneck (2004). A hybrid heuristic for the *p*-median problem. *Journal of Heuristics 10*, 59–88.

Rey, S. J. and L. Anselin (2010). PySAL: A Python library of spatial analytical methods. In M. Fischer and A. Getis (eds), *Handbook of Applied Spatial Analysis: Software Tools, Methods and Applications*, pp. 175–193. Berlin: Springer.

Richardson, R. T. (1989). Area deformation on the Robinson projection. *American Cartographer 16*, 294–296.

Riitters, K. H., R. V. O'Neill, J. D. Wickham, and K. B. Jones (1996). A note on contagion indices for landscape analysis. *Landscape Ecology 11* (4), 197–202.

Ripley, B. D. (1981). *Spatial Statistics*. New York: John Wiley & Sons.

Robinson, A. H. (1974). A new map projection: Its development and characteristics. *International Yearbook of Cartography 14*, 145–155.

Rosing, K. E. (1997). An empirical investigation of the effectiveness of a vertex substitution heuristic. *Environment and Planning B: Planning and Design 24* (1), 59–67.

Rosing, K. E., E. L. Hillsman, and H. Rosing-Vogelaar (1979). A note comparing optimal and heuristic solutions to the *p*-median problem. *Geographical Analysis 11* (1), 86–89.

Rosing, K. E. and M. J. Hodgson (2002). Heuristic concentration for the *p*-median: An example demonstrating how and why it works. *Computers & Operations Research 29*, 1317–1330.

Samet, H. (1990a). *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Reading, MA: Addison-Wesley.

Samet, H. (1990b). *The Design and Analysis of Spatial Data Structures*. Reading, MA: Addison-Wesley.

Samet, H. (2006). *Foundations of Multidimensional and Metric Data Structures*. San Francisco: Morgan Kaufmann.

Samet, H. and R. E. Webber (1985). Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics 4* (3), 182–222.

Sellis, T., N. Roussopoulos, and C. Faloutsos (1987). The R+-tree: A dynamic index for multi-dimensional objects. Technical report, VLDB Endowments.

Shamos, M. and D. Hoey (1976). Geometric intersection problems. In *17th Annual Symposium on Foundations of Computer Science*, pp. 208–215. IEEE.

Sinnott, R. (1984). Virtues of the Haversine. *Sky and Telescope 68*, 159.

Snyder, J. P. (1987). Map projections – a working manual. Professional paper 1395, U.S. Geological Survey.

Snyder, J. P. (1990). The Robinson projection: A computation algorithm. *Cartography and Geographic Information Systems 17* (4), 301–305.

Sylvester, J. J. (1860). On Poncelet's approximate linear valuation of Surd forms. *Philosophical Magazine 20*, 203–222.

Teitz, M. B. and P. Bart (1968). Heuristic methods for estimating the generalized vertex median of a weighted graph. *Operations Research 16*, 955–961.

Tobler, W. R. (1970). A computer movie simulating urban growth in the Detroit region. *Economic Geography 46*, 234–240.

Webster, R. and M. A. Oliver (1990). *Statistical Methods in Soil and Land Resource Survey*. Oxford: Oxford University Press.

Welzl, E. (1991). Smallest enclosing disks (balls and ellipsoids). In H. Maurer (ed.), *New Results and New Trends in Computer Science*, pp. 359–370. Berlin: Springer.

Whitaker, R. (1983). A fast algorithm for the greedy interchange of large-scale clustering and median location problems. *INFOR 21*, 95–108.

Wilford, J. N. (1988). The impossible quest for the perfect map. *New York Times*, October 25. http://nyti.ms/1BcUqAt.

Wolpert, D. and W. Macready (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation 1* (1), 67–82.

Xiao, N. (2006). An evolutionary algorithm for site search problems. *Geographical Analysis 38* (3), 227–247.

Xiao, N. (2008). A unified conceptual framework for geographical optimization using evolutionary algorithms. *Annals of the Association of American Geographers 98* (4), 795–817.

Xiao, N. (2012). A parallel cooperative hybridization approach to the *p*-median problem. *Environment and Planning B: Planning and Design 39*, 755–774.

Xiao, N., D. A. Bennett, and M. P. Armstrong (2002). Using evolutionary algorithms to generate alternatives for multiobjective site search problems. *Environment and Planning A 34* (4), 639–656.

Xiao, N., D. A. Bennett, and M. P. Armstrong (2007). Interactive evolutionary approaches to multiobjective spatial decision making: A synthetic review. *Computers, Environment and Urban Systems 30*, 232–252.

Xiao, N. and A. T. Murray (2015). Interpolation. In M. Monmonier (ed.), *History of Cartography, Volume 6: Cartography in the Twentieth Century*. Chicago: University of Chicago Press.

Young, C., D. Martin, and C. Skinner (2009). Geographically intelligent disclosure control for flexible aggregation of census data. *International Journal of Geographical Information Science 23* (4), 457–482.

Предметный указатель

Примечание. Номера страниц, выделенные **полужирным** шрифтом, относятся к развернутому изложению темы.

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
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.a-planeta.ru**.

Нинчуань Сяо

Алгоритмы ГИС

Главный редактор *Мовчан Д. А.*
Заместитель главного редактора *Сенченкова Е. А.*
dmkpress@gmail.com
Перевод *Слинкин А. А.*
Корректор *Абросимова Л. А.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 26,65. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**