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.

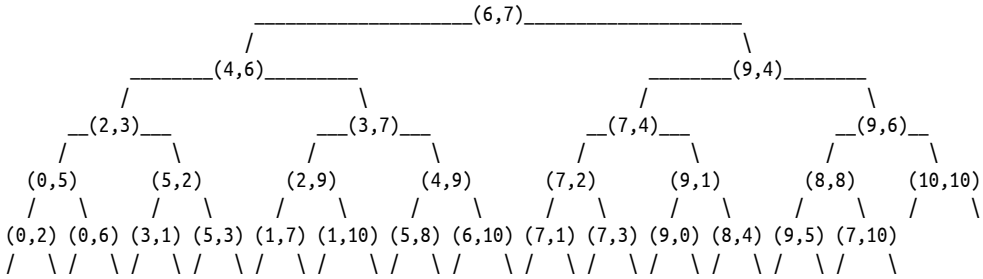

Рис. 1.1 ❖ Дерево, в котором хранится 29 случайно выбранных точек. Каждый узел помечен координатами X и Y, изменяющимися в диапазоне от 0 до 10

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.

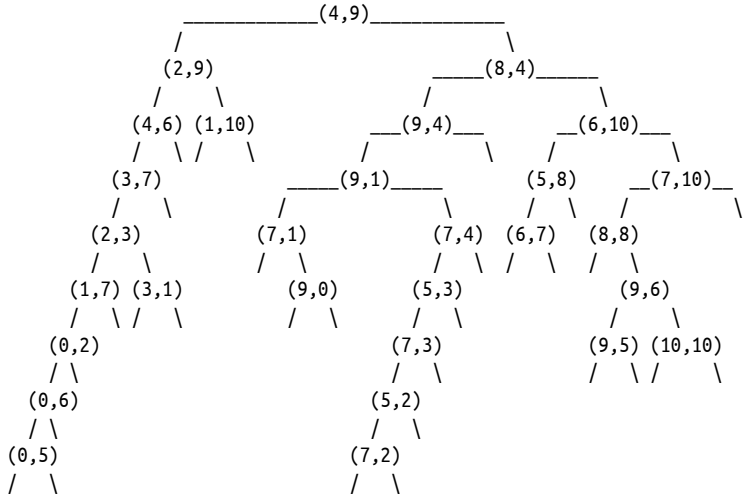

Рис. 1.2 ❖ Несбалансированное дерево, в котором хранится 29 случайных точек

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
```