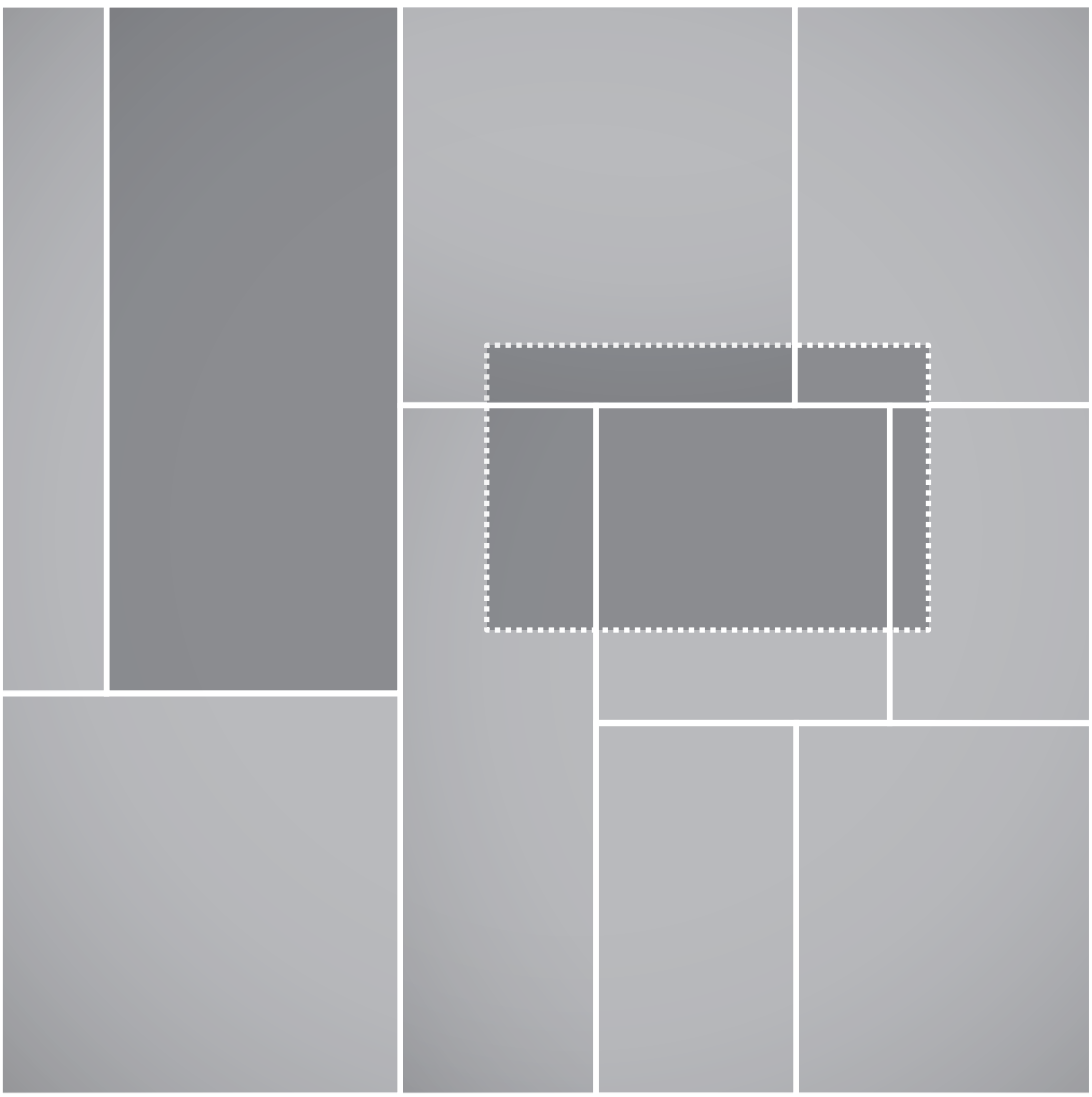
GIST

GIS Algorithms

Ningchuan Xiao



SAGE

Los Angeles | London | New Delhi
Singapore | Washington DC

Los Angeles | London | New Delhi
Singapore | Washington DC

SAGE Publications Ltd
1 Oliver's Yard
55 City Road
London EC1Y 1SP

SAGE Publications Inc.
2455 Teller Road
Thousand Oaks, California 91320

SAGE Publications India Pvt Ltd
B 1/I 1 Mohan Cooperative Industrial Area
Mathura Road
New Delhi 110 044

SAGE Publications Asia-Pacific Pte Ltd
3 Church Street
#10-04 Samsung Hub
Singapore 049483

First published 2016

Editor: Robert Rojek
Editorial assistant : Matt Oldfield
Production editor: Katherine Haw
Copyeditor: Richard Leigh
Proofreader: Richard Hutchinson
Indexer: Bill Farrington
Marketing manager: Michael Ainsley
Cover design: Francis Kenney
Typeset by: C&M Digitals (P) Ltd, Chennai, India
Printed and bound by CPI Group (UK) Ltd,
Croydon, CR0 4YY

Library of Congress Control Number: 2015940434

British Library Cataloguing in Publication data

A catalogue record for this book is available from
the British Library

ISBN 978-1-4462-7432-3
ISBN 978-1-4462-7433-0 (pbk)

1

Introduction

Algorithms[1] are designed to solve computational problems. In general, an algorithm is a process that contains a set of well designed steps for calculation. For example, to correctly calculate the sum of 18 and 19, one must know how to deal with the fact that 8 plus 9 is more than 10, though different cultures have different ways of processing that. Even for a simple problem like this, we expect that the steps used can help us get the answer quickly and correctly. There are many problems that are more difficult than the simple problem of addition, and solving these problems, again efficiently and correctly, requires more careful design of the computational steps.

In GIS development and applications, algorithms are important in almost every aspect. When we click on a map, for example, we expect a quick response from the computer system so that we can pull out relevant information about the point or area we just clicked on. Such a fundamental daily routine for almost every GIS application involves a variety of algorithms to ensure a satisfying response. It starts from searching for the object (point, line, polygon, or pixel) underneath the clicking point. An efficient search algorithm will allow us to narrow down to the area of interest quickly. While a brute-force approach may work by physically checking every object in our data, it will not be useful for a large data set that will make the time of finding the target impractically long. Many spatial indexing and query algorithms are designed to address this issue. While the search is ongoing, we must check whether the object in our data matches the point clicked. For polygons, we must decide whether the click point is within a polygon in our data, which requires a special algorithm to quickly return a yes or no answer to decide whether the point is in the polygon. Geospatial data normally come from many different sources, and it has been common practice to transform them into the same coordinate system so that different data sets can be processed consistently. Another common application of the multiple data sources is to overlay them to make the information more useful together.

There are many aspects of an algorithm to be examined. It is straightforward to require that an algorithm solve the problem correctly. For some algorithms, it is easy to prove their

[1]The word "algorithm" itself comes from medieval Latin *algorismus*, which is from the name of *al-Khwārizmī*, a Persian mathematician, astronomer, and geographer, who made great contributions to the knowledge of algebra and world geography.

correctness. For example, we will introduce two search algorithms later in this chapter, and their correctness should be quite straightforward. Other algorithms, however, are not so obvious, and proving their correctness will require more formal analysis. A second feature of algorithms is their efficiency or running time. Of course we always want an algorithm to be fast, but there are theoretical limits on how fast or efficient an algorithm can be, as determined by the problem. We will discuss some of those problems at the end of the book under topics of spatial optimization. Besides correctness and running time, algorithms are often closely related to how the data are organized to enable the processes and how the algorithms are actually implemented.

1.1 Computational concerns for algorithms

Let us assume we have a list of *n* points and the list does not have any order. We want to find a point from the list. How long will we take to find the point? This is a reasonable question. But the actual *time* is highly related to a lot of issues such as the programming language, the skill of the person who codes the program, the platform, the speed and number of the CPUs, and so on. A more useful way to examine the time issue is to know how many *steps* we need to finish the job, and then we analyze the total cost of performing the algorithm in terms of the number of steps used. The cost of each step is of course variable and is dependent on what constitutes a step. Nevertheless, it is still a more reliable way of thinking about computing time because many computational steps, such as simple arithmetic operations, logical expressions, accessing computer memory for information retrieval, and variable value assignment, can be identified and they only cost a constant amount of time. If we can figure out a way to count how many steps are needed to carry out a procedure, we will then have a pretty good idea about how much time the entire procedure will cost, especially when we compare algorithms.

Returning to our list of points, if there is no structure in the list – the points are stored in an arbitrary order – the best we can do to find a point from the list is to test all the points in the list, one by one, until we can conclude it is in or not in the list. Let us assume the name of the list is `points` and we want to find if the list includes point `p0`. We can use a simple algorithm to do the search (Listing 1.1).

Listing 1.1: Linear search to find point `p0` in a list.

```
1 for each point p in points:
2     if p is the same as p0:
3         return p and stop
```

The algorithm in Listing 1.1 is called a linear search; in it we simply go through all the points, if necessary, to search for the information we need. How many steps are necessary in this algorithm? The first line is a loop and, because of the size of the list, it will run as many as *n* times when the item we are looking for happens to be the last one in the list. The cost of running just once in the loop part in line 1 is a constant because the list is stored in the computer memory, and the main operation steps here are to access the information at a fixed location in the memory and then to move on to the next item in the

memory. Suppose that the cost is c_1 and we will run it up to to n times in the loop. The second line is a logic comparison between two points. It will run up to n times as well because it is inside the loop. Suppose that the cost of doing a logic comparison is c_2 and it is a constant too. Line 3 simply returns the value of the point found; it has a constant cost of c and it will only run once. For the best case scenario, we will find the target at the first iteration of the loop and therefore the total cost is simply $c_1 + c_2 + c$, which can be generalized as a constant $b + c$. In the worst case scenario, however, we will need to run all the way to the last item in the list and therefore the total cost becomes $c_1n + c_2n + c$, which can be generalized as $bn + c$, where b and c are constants, and n is the size of the list (also the size of the problem). On average, if the list is a random set of points and we are going to search for a random point many times, we should expect a cost of $c_1n/2 + c_2n/2 + c$, which can be generalized as $b'n + c$, and we know $b' < b$, meaning that it will not cost as much as the worse case scenario does.

How much are we interested in the actual values of b, b', and c in the above analysis? How will these values impact the total computation cost? As it turns out, not much, because they are constants. But adding them up many times will have a real impact and n, the problem size, generally controls how many times these constant costs will be added together. When n reaches a certain level, the impact of the constants will become minimal and it is really the magnitude of n that controls the *growth* of the total computation cost.

Some algorithms will have a cost related to n^2, which is significantly different from the cost of n. For example, the algorithm in Listing 1.2 is a simple procedure to compute the shortest pairwise distance between two points in the list of n points. Here, the first loop (line 2) will run n times at a cost of t_1 each, and the second loop (line 3) will run exactly n^2 times at the same cost of t_1 each. The logic comparison (line 4) will run n^2 times and we assume each time the cost is t_2. The calculation of distance (line 5) will definitely be more costly than the other simple operations such as logic comparison, but it is still a constant as the input is fixed (with two points) and only a small finite set of steps will be taken to carry out the calculation. We say the cost of each distance calculation is a constant t_3. Since we do not compute the distance between the point and itself, the distance calculation will run $n^2–n$ times, as will the comparison in line 6 (with time t_4). The assignment in line 7 will cost a constant time of t_5 and may run up to $n^2–n$ times in the worst case scenario where every next distance is shorter than the previous one. The last line will only run once with a time of c. Overall, the total time for this algorithm will be $t_1n + t_1n^2 + t_2n^2 + t_3(n^2–n) + t_4(n^2–n) + t_5(n^2 – n) + c$, which can be generalized as $an^2 + bn + c$. Now it should be clear that this algorithm has a running time that is controlled by n^2.

Listing 1.2: Linear search to find shortest pairwise distance in a list of points.

```
1  let mindist be a very large number
2  for each point p1 in points:
3      for each point p2 in points:
4          if p1 is not p2:
5              let d be the distance between p1 and p2
6              if d < mindist:
7                  mindist = d
8  return mindist and stop
```

In the two example algorithms we have examined so far, the order of n indicates the total cost and we say that our linear search algorithm has a computation cost in the order of n and the shortest pairwise distance algorithm in the order of n^2. For the linear search, we also know that, when n increases, the total cost of search will always have an upper bound of bn. But is there a lower bound? We know the best case scenario has a running time of a constant, or in the order of n^0, but that does not apply to the general case. When we can definitely find an upper bound but not a lower bound of the running time, we use the O-notation to denote the order. In our case, we have $O(n)$ for the average case, and the worst case scenario as well (because again the constants do not control the total cost). In other words, we say that the running time, or time complexity, of the linear search algorithm is $O(n)$. Because the O-notation is about the upper bound, which is meant to be the worst case scenario, we also mean the time complexity of the worst case scenario.

There are algorithms for which we do not have the upper bound of their running time. But we know their lower bound and we use the Ω-notation to indicate that. A running time of $\Omega(n)$ would mean we know the algorithm will at least cost an order of n in its running time, though we do not know the upper bound of the running time. For other algorithms, we know both upper and lower bounds of the running time and we use the Θ-notation to indicate that. For example, a running time of $\Theta(n^2)$ indicates that the algorithm will take an order of n^2 in running time in all cases, best and worst. This is the case for our shortest distance algorithm because the process will always run n^2 times, regardless of the outcome of the comparison in line 6. It is more accurate to say the time complexity is $\Theta(n^2)$ instead of $O(n^2)$ because we know the lower bound of the running time of pairwise shortest distance is always in the order of n^2.

Now we reorganize our points in the previous list in a particular tree structure as illustrated in Figure 1.1. This is a binary tree because each node on the tree can have at most two branches, starting from the root. Here the root of the tree stores point (6, 7) and we show it at the top. All the points with X coordinates smaller than or equal to that at the root are stored in the left branches of the root and those with X coordinates greater than that of the root point are stored in the right branches of the root. Going down the tree to the second level, we have two points there, (4, 6) and (9, 4). For each of these points, we make sure that the rest of the points will be stored on the left if they have a smaller or equal Y coordinate value, and on the right if greater. We alternate the use of X and Y coordinates going down the tree, until we find the point we are looking for or reach the end of a branch (a leaf node).

To use the tree structure to search for a point, we start from the root (we always start from the root for a tree structure) and go down the tree by determining which branch to proceed along using the appropriate coordinate at each level of the tree. For example, to

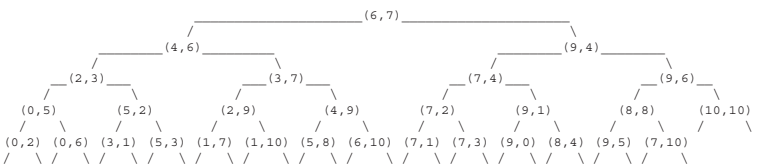


Figure 1.1 A tree structure that stores 29 random points. Each node of the tree is labeled using a point with X and Y coordinates that range from 0 to 10

search for a target point of (1, 7), we first go to the left branch of the root because the X coordinate of our target point is 1, smaller than that in the root. Then we go the the right branch of the second level node of (4, 6) because the Y coordinate (7) is greater than that in the node. Now we reach the node of (3, 7) at the third level of the tree and we will go to the left branch there because the target X coordinate is smaller than that in the node. Finally, we reach the node (2, 9) and we will move to its left branch because the Y coordinate in the target is smaller than that in the node. In sum, given a tree, we can write the algorithm in Listing 1.3 to fulfill such a search strategy using a tree structure.

Listing 1.3: Binary search to find point `p0` in a tree.

```
1  let t be the root of the tree
2  while t is not empty:
3      let p be the point at node t
4      if p is the same as point p0:
5          return p and stop
6      if t is on an even level of the tree:
7          coordp, coordp0 = X coordinates of p and p0
8      else:
9          coordp, coordp0 = Y coordinates of p and p0
10     if coordp0 <= coordp:
11         t = the left branch of t
12     else:
13         t = the right branch of t
```

This is called a binary search, using the tree structure. Based on our discussion about running time, it is straightforward to see that the running time of this search algorithm is determined by the number of times we have to run the while loop (line 2), which is determined by the height of the tree, as defined by the number of edges from the root to the farthest leaf node. The above tree has a height of 4 and can hold up to 31 points (we only have 29 here). In general, for a binary tree with a height of H, we can store up to $2^0 + 2^1 + 2^2 + \ldots + 2^H = 2^{H+1} - 1$ items. In other words, if we have n points in total that fill all the nodes in a perfectly balanced binary tree where all the leaf nodes are at exactly the same level, we have $2^{H+1} - 1 = n$ and hence $H = \log_2(n + 1) - 1$. In this case, when

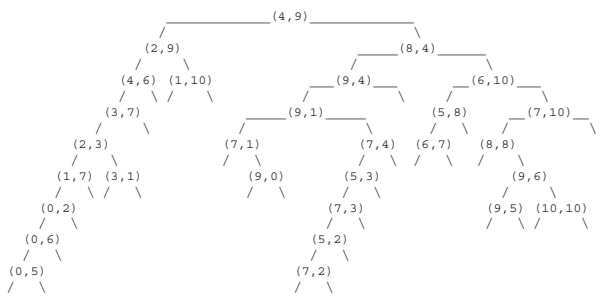


Figure 1.2 An unbalanced tree structure to store 29 random points

such a tree is given, we have a running time of the order of $\log_2(n+1)$, which is at the same order as $\log_2 n$, and we say the running time is $O(\log_2 n)$. For a balanced but not perfect tree, meaning the difference between the heights of all leaf nodes is at most 1, we can still achieve a running time of $O(\log_2 n)$ since the farthest leaf node has a height of H. When a balanced tree cannot be guaranteed, however, things can get worse. An example of an unbalanced tree is given in Figure 1.2 where we have exactly the same points but the tree has a height of 8. Therefore, we know the binary search algorithm is more efficient than linear search because $O(\log_2 n) < O(n)$, but the actual running time is dependent on how the tree is constructed and can be longer than $O(\log_2 n)$ if the tree is unbalanced.

Since using a balanced binary tree to store the points can greatly improve the efficiency of search, will that also help to reduce the running time of calculating the shortest pairwise distance? The brute-force approach we discussed here has a running time of $\Theta(n^2)$, which grows quickly as n increases. It would be reasonable to look for a more efficient way to get the shortest pairwise distance in a list of points. The answer to our question is yes, and the key lies in the use of tree structures. We will continue to explore this topic in the second part of the book where we discuss different tree structures in more depth. Throughout the book, we do not focus much on the theoretical analysis of running time for each algorithm. Instead, we will conduct more empirical analysis by actually running the algorithms on different data sets.

The discussion about search and especially binary search on a tree logically leads to the topic of data structure: how we store and organize data to facilitate the procedures in an algorithm. A tree structure is a good example of how the original data stored in a list can be reorganized to achieve better search performance. Many data structures are problem-specific. Some data structures can be complicated, but the increase in storage is often compensated by a decrease in running time.

1.2 Coding

Algorithms can be described in different ways. We use verbal statements in this chapter to describe the linear and binary search algorithms. For theoretical work, a formal description that details the steps but is not necessarily executable will suffice. We call this type of description pseudo-code because it is not real computer code, though very close. In this book, we take a more practical and explicit route by describing algorithms in an actual computer programming language. Specifically, we use Python to describe the algorithms covered in this book.

Writing computer programs (i.e., coding) to describe algorithms has a substantial benefit: all the algorithms will immediately be executable. In this way, we present everything related to how the algorithms work in the plain text of the book. The code becomes part of the text and consequently becomes an open source experiment where each line of the process can be examined, modified, and improved. However, this code as text approach may present too much information, especially when the programming language may need ancillary code to help the main task. For example, many programming languages require end of line symbols and brackets as part of the code to ensure syntax correctness. These symbols, when added as part of the text, may

hamper the reading process and therefore make it difficult for us to concentrate on the main contents of the text. We choose Python in this book largely because of its simple syntax, along with many of its popular, powerful, and well maintained modules. All the programs listed in this book were tested in Python 2.7, which is a stable and widely adopted version at the time of writing. The majority of the programs in this book only use the basic Python features, so these programs in principle are likely to be compatible with newer versions of Python.

Python has become a popular programming language in recent years, including the use of Python for plugins in GIS packages such as QGIS and ArcGIS. It is important to point out that programming in Python is a skill that can definitely be acquired through learning and practice. To help readers make a quick start on the language, a short introduction to Python is included as Appendix A. This is not a comprehensive tutorial as many of the online tutorials have more detailed and in-depth discussion about the language. However, many of Python's useful features, especially those related to the main text, are included in the tutorial.

1.3 How to use this book

The main text of this book is divided into three major parts.The general logic here is to start the book with a discussion on the most fundamental aspects of the data – the geometry – before we move on to more advanced topics in spatial indexing and spatial analysis and modeling. At the end of each chapter we review the major literature related to the topics covered in a section called Notes. At the end of the book, we also include three appendices to help readers understand the Python programming language and the structure of the programs included in the book.

In Part I, we focus on locations, or more specifically on coordinates that can be used to help us understand geospatial information. In Chapter 2, we examine a few algorithms to compute different kinds of distance, such as distances between points and distance from a point to a line. We also look at the calculation of polygon centroids and a widely used algorithm called point-in-polygon that efficiently helps us determine whether a point is located within a polygon. The final topic in Chapter 2 is about the transformation between coordinate systems involving map projections. Chapter 3 covers a traditional GIS operation, known as overlay. As "old" as this topic is, the actual computation of overlaying two polygons can be tedious, though not necessarily complicated. Many of the topics in this part of the book are related to the field of computational geometry. But we focus on those that are most relevant to the GIS world.

Part II is centered around the idea of spatial indexing. Spatial information is special. Though the general concept in indexing, divide and conquer, is the same for spatial information, because of the two dimensions (or more in some cases) in spatial information, more dedicated algorithms must be designed. We first introduce the basic concepts of indexing in Chapter 4, where we focus on the development of a tree structure. Chapter 5 is devoted to *k*-D trees that are commonly used to index point data. Chapter 6 covers a popular indexing technique called quadtrees, for both point and raster data. Chapter 7 extends the discussion to indexing lines and polygons in spatial data.

Part III of the book focuses on the heart of GIS applications: spatial analysis and modeling. We first explore the interpolation methods on point data in Chapter 8 where we compare and contrast two commonly used interpolation methods: inverse distance weighting and kriging. We also include a data simulation algorithm called midpoint displacement from the fractal geometry literature. Chapter 9 is devoted to spatial pattern analysis where the calculation of indices such as Moran's I is included. Algorithms for network analysis, especially those calculating the shortest paths, are included in Chapter 10. We devote two chapters to topics in spatial optimization: in Chapter 11 we focus on the exact methods and in Chapter 12 we explore some of the heuristic methods.

In addition to the three parts in the main text, we have also included three appendices to cover some of the technical details about coding. It should be obvious that, while we talk about algorithms most of the time, this book is about coding as well. For this reason, a short introduction to Python is first included. Then we compile a short introduction on the Python binding of a powerful library called GDAL/OGR and a Python library for spatial analysis called PySAL. The purpose here is to help readers quickly get started with these libraries so that they can have a sense how "real-world" data sets can be closely related to the topics (and code of course) presented in this book.

Most of the programs listed in this book are also given a file name that can be used in other programs. In that case the name of the program is listed in the caption of each code listing. We use directories to organize the programs. In the last appendix, we provide an overview of the code by listing all the Python programs and example data sets and discuss how they can be used.

Each of the chapters in this book could easily be extended into another book to cover the depth of each topic. This book, then, is a survey of the general topics in GIS algorithms. The best way to grasp the breadth of the topics presented in this book is coding. A collective Github page[2] is under development where readers can contribute their thoughts on implementing the algorithms included in this book and new algorithms that are beyond the scope of the book. While theoretical derivation is not the focus of this book, empirical analysis definitely is. We have included many experiments in the text and have suggested more in the exercises. This, however, should not limit further experiments, especially those that are innovative and overarch various topics. The book will only be successful if it achieves two goals: first, based on the skills built on understanding the algorithms and code, that readers are able to develop their own tool sets that fit different data sets and application requirements; and second, that coding becomes a habit when it comes to dealing with geospatial data.

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