Algorithms Illuminated

Part 2: Graph Algorithms and Data Structures

Tim Roughgarden

Second Printing, 2021

First Edition

Cover image: *Untitled*, by Nick Terry

ISBN: 978-0-9992829-2-2 (Paperback)
ISBN: 978-0-9992829-3-9 (ebook)

Library of Congress Control Number: 2017914282

Soundlikeyourself Publishing, LLC
New York, NY
`soundlikeyourselfpublishing@gmail.com`
`www.algorithmsilluminated.org`

Contents

Preface

This book is the second in a series based on my online algorithms courses that have been running regularly since 2012, which in turn are based on an undergraduate course that I taught many times at Stanford University. The first part of the series is not a prerequisite for this one, and this book should be accessible to any reader who has the background described in the "Who Are You?" section below and is familiar with asymptotic notation (which is reviewed in Appendix C).

What We'll Cover in This Book

Algorithms Illuminated, Part 2 provides an introduction to and basic literacy in the following three topics.

Graph search and applications. Graphs model many different types of networks, including road networks, communication networks, social networks, and networks of dependencies between tasks. Graphs can get complex, but there are several blazingly fast primitives for reasoning about graph structure. We begin with linear-time algorithms for searching a graph, with applications ranging from network analysis to task sequencing.

Shortest paths. In the shortest-path problem, the goal is to compute the best route in a network from point A to point B. The problem has obvious applications, like computing driving directions, and also shows up in disguise in many more general planning problems. We'll generalize one of our graph search algorithms and arrive at Dijkstra's famous shortest-path algorithm.

Data structures. This book will make you an educated client of several different data structures for maintaining an evolving set of objects with keys. The primary goal is to develop your intuition about which data structure is the right one for your application. The optional advanced sections provide guidance in how to implement these data structures from scratch.

We first discuss heaps, which can quickly identify the stored object with the smallest key and are useful for sorting, implementing a priority queue, and implementing Dijkstra's algorithm in near-linear time. Search trees maintain a total ordering over the keys of the stored objects and support an even richer array of operations. Hash tables are optimized for super-fast lookups and are ubiquitous in modern programs. We'll also cover the bloom filter, a close cousin of the hash table that uses less space at the expense of occasional errors.

For a more detailed look into the book's contents, check out the "Upshot" sections that conclude each chapter and highlight the most important points. The starred sections are the most advanced ones, and a time-constrained reader can skip them on a first reading without any loss of continuity.

Topics covered in the other parts. *Algorithms Illuminated, Part 1* covers asymptotic notation (big-O notation and its close cousins), divide-and-conquer algorithms and the master method, randomized QuickSort and its analysis, and linear-time selection algorithms. *Part 3* focuses on greedy algorithms (scheduling, minimum spanning trees, clustering, Huffman codes) and dynamic programming (knapsack, sequence alignment, shortest paths, optimal search trees). *Part 4* is all about NP-hard problems: how to recognize them in the wild (using reductions); ways to compromise on your algorithmic ambitions for them (through approximation or an exponential worst-case running time); and algorithmic tools to realize those revised ambitions (greedy heuristic algorithms, local search, advanced dynamic programming, MIP and SAT solvers).

Skills You'll Learn From This Book Series

Mastering algorithms takes time and effort. Why bother?

Become a better programmer. You'll learn several blazingly fast subroutines for processing data as well as several useful data structures for organizing data that you can deploy directly in your own programs. Implementing and using these algorithms will stretch and improve your programming skills. You'll also learn general algorithm design paradigms that are relevant to many different problems across different domains, as well as tools for predicting the performance of such algorithms. These "algorithmic design patterns" can help you come up with new algorithms for problems that arise in your own work.

Sharpen your analytical skills. You'll get lots of practice describing and reasoning about algorithms. Through mathematical analysis, you'll gain a deep understanding of the specific algorithms and data structures that these books cover. You'll acquire facility with several mathematical techniques that are broadly useful for analyzing algorithms.

Think algorithmically. After you learn about algorithms, you'll start seeing them everywhere, whether you're riding an elevator, watching a flock of birds, managing your investment portfolio, or even watching an infant learn. Algorithmic thinking is increasingly useful and prevalent in disciplines outside of computer science, including biology, statistics, and economics.

Literacy with computer science's greatest hits. Studying algorithms can feel like watching a highlight reel of many of the greatest hits from the last sixty years of computer science. No longer will you feel excluded at that computer science cocktail party when someone cracks a joke about Dijkstra's algorithm. After reading these books, you'll know exactly what they mean.

Ace your technical interviews. Over the years, countless students have regaled me with stories about how mastering the concepts in these books enabled them to ace every technical interview question they were ever asked.

How These Books Are Different

This series of books has only one goal: *to teach the basics of algorithms in the most accessible way possible.* Think of them as a transcript of what an expert algorithms tutor would say to you over a series of one-on-one lessons.

There are a number of excellent more traditional textbooks about algorithms, any of which usefully complement this book series with additional problems and topics. I encourage you to explore and find your own favorites. There are also several books that, unlike these books, cater to programmers looking for ready-made algorithm implementations in a specific programming language. Many such implementations are freely available on the Web as well.

Who Are You?

The whole point of these books and the online courses upon which they are based is to be as widely and easily accessible as possible. People of all ages, backgrounds, and walks of life are well represented in my online courses, and there are large numbers of students (high-school, college, etc.), software engineers (both current and aspiring), scientists, and professionals hailing from all corners of the world.

This book is not an introduction to programming. Ideally, you've already acquired basic programming skills, such as the use of arrays and recursion, in some standard programming language (be it Java, Python, C, Scala, Haskell, etc.). For a litmus test, check out Section 8.2—if it makes sense, you'll be fine for the rest of the book. If you need to beef up your programming skills, there are several outstanding free online courses that teach basic programming.

We also use mathematical analysis as needed to understand how and why algorithms really work. The freely available book *Mathematics for Computer Science*, by Eric Lehman, F. Thomson Leighton, and Albert R. Meyer is an excellent and entertaining refresher on mathematical notation (like $\sum$ and $\forall$), the basics of proofs (induction, contradiction, etc.), discrete probability, and much more.

Additional Resources

These books are based on online courses that are currently running on the Coursera and edX platforms. I've made several resources available to help you replicate as much of the online course experience as you like.

Videos. If you're more in the mood to watch and listen than to read, check out the YouTube video playlists available at `www.algorithmsilluminated.org`. These videos cover all the topics in this book series, as well as additional advanced topics. I hope they exude a contagious enthusiasm for algorithms that, alas, is impossible to replicate fully on the printed page.

Quizzes. How can you know if you're truly absorbing the concepts in this book? Quizzes with solutions and explanations are scattered throughout the text; when you encounter one, I encourage you to pause and think about the answer before reading on.

End-of-chapter problems. At the end of each chapter you'll find several relatively straightforward questions for testing your understanding, followed by harder and more open-ended challenge problems. Hints or solutions to most of these problems (as indicated by an "*(H)*" or "*(S)*," respectively) are included at the end of the book. Readers can interact with me and each other about the end-of-chapter problems through the book's discussion forum (see below).

Programming problems. Most of the chapters conclude with a suggested programming project whose goal is to help you develop a detailed understanding of an algorithm by creating your own working implementation of it. Data sets, along with test cases and their solutions, can be found at `www.algorithmsilluminated.org`.

Discussion forums. A big reason for the success of online courses is the opportunities they provide for participants to help each other understand the course material and debug programs through discussion forums. Readers of these books have the same opportunity, via the forums available at `www.algorithmsilluminated.org`.

Changes Since the First Printing

The second printing of this book (July 2021) includes numerous minor improvements throughout the text, several new end-of-chapter problems, hints or solutions to almost all end-of-chapter problems, and additional applications of the single-source shortest path problem (Section 9.1.2).

Acknowledgments

These books would not exist without the passion and hunger supplied by the hundreds of thousands of participants in my algorithms courses over the years. I am particularly grateful to those who supplied detailed feedback on an earlier draft of this book: Tonya Blust, Yuan Cao, Jim Humelsine, Vladimir Kokshenev, Bayram Kuliyev, Patrick Monkelban, and Daniel Zingaro.

I always appreciate suggestions and corrections from readers. These are best communicated through the discussion forums mentioned above.

Tim Roughgarden
London, United Kingdom
July 2018

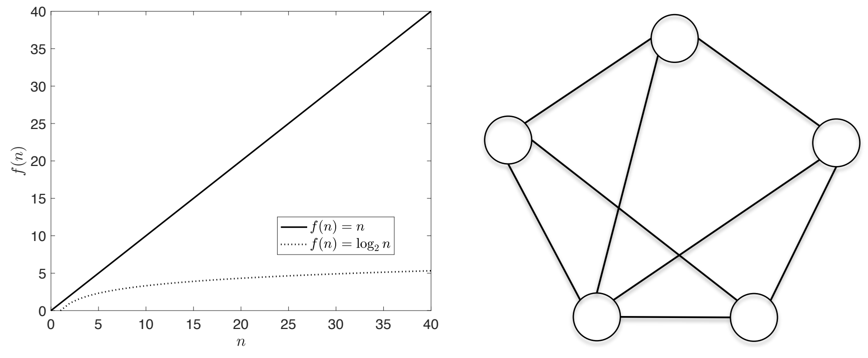
Chapter 7

Graphs: The Basics

This short chapter explains what graphs are, what they are good for, and the most common ways to represent them in a computer program. The next two chapters cover a number of famous and useful algorithms for reasoning about graphs.

7.1 Some Vocabulary

When you hear the word "graph," you probably think about an x-axis, a y-axis, and so on (Figure 7.1(a)). To an algorithms person, a *graph* can also mean a representation of the relationships between pairs of objects (Figure 7.1(b)).

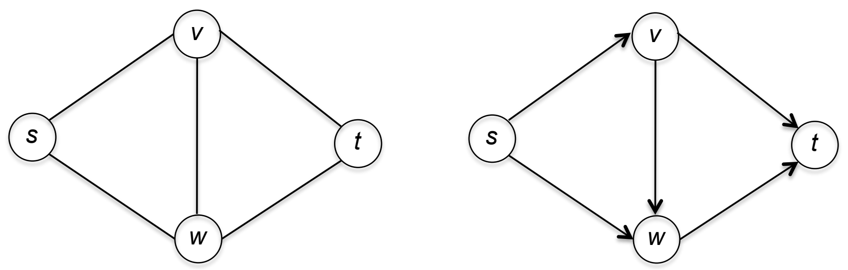


(a) A graph (to most of the world)   (b) A graph (in algorithms)

Figure 7.1: In algorithms, a graph is a representation of a set of objects (such as people) and the pairwise relationships between them (such as friendships).

The second type of graph has two ingredients—the objects being represented, and their pairwise relationships. The former are called the *vertices* (singular: vertex) or the *nodes* of the graph.[1] The pairwise relationships translate to the *edges* of the graph. We usually denote the vertex and edge sets of a graph by V and E, respectively, and sometimes write $G = (V, E)$ to mean the graph G with vertices V and edges E.

There are two flavors of graphs, directed and undirected. Both types are important and ubiquitous in applications, so you should know about both of them. In an *undirected* graph, each edge corresponds to an unordered pair $\{v, w\}$ of vertices, which are called the *endpoints* of the edge (Figure 7.2(a)). In an undirected graph, an edge with endpoints v and w can be denoted by (v, w) or by (w, v)—there is no difference between the two. In a *directed* graph, each edge (v, w) is an ordered pair, with the edge traveling from the first vertex v (called the *tail*) to the second w (the *head*); see Figure 7.2(b).[2]



(a) An undirected graph (b) A directed graph

Figure 7.2: Graphs with four vertices and five edges. The edges of undirected and directed graphs are unordered and ordered vertex pairs, respectively.

7.2 A Few Applications

Graphs are a fundamental concept, and they show up all the time in computer science, biology, sociology, economics, and so on. Here are a few of the countless examples.

[1]Having two names for the same thing can be annoying, but both terms are in widespread use and you should be familiar with them. For the most part, we'll stick with "vertices" throughout this book series.

[2]Directed edges are sometimes called *arcs*, but we won't use this terminology in this book series.

Road networks. When an app on your smartphone computes driving directions, it searches through a graph that represents the road network, with vertices corresponding to intersections and edges corresponding to individual road segments.

The World Wide Web. The Web can be modeled as a directed graph, with the vertices corresponding to individual Web pages, and the edges corresponding to hyperlinks, directed from the page containing the hyperlink to the destination page.

Social networks. A social network can be represented as a graph whose vertices correspond to individuals and edges to some type of relationship. For example, an edge could indicate a friendship between its endpoints, or that one of its endpoints is a follower of the other. Among the currently popular social networks, which ones are most naturally modeled as an undirected graph, and which ones as a directed graph? (There are interesting examples of both.)

Precedence constraints. Graphs are also useful in problems that lack an obvious network structure. For example, imagine that you have to complete a bunch of tasks, subject to precedence constraints—perhaps you're a first-year university student, planning which courses to take and in which order. One way to tackle this problem is to apply the topological sorting algorithm described in Section 8.5 to the following directed graph: there is one vertex for each course that your major requires, with an edge directed from course A to course B whenever A is a prerequisite for B.

7.3 Measuring the Size of a Graph

In this book, like in *Part 1*, we'll analyze the running time of different algorithms as a function of the input size. When the input is a single array, as for a sorting algorithm, there is an obvious way to define the "input size," as the array's length. When the input involves a graph, we must specify exactly how the graph is represented and what we mean by its "size."

7.3.1 The Number of Edges in a Graph

Two parameters control a graph's size—the number of vertices and the number of edges. Here is the most common notation for these quantities.

Notation for Graphs

For a graph $G = (V, E)$ with vertex set V and edge set E:

- $n = |V|$ denotes the number of vertices; and
- $m = |E|$ denotes the number of edges.[3]

The next quiz asks you to think about how the number m of edges in an undirected graph can depend on the number n of vertices. For this question, we'll assume that there's at most one undirected edge between each pair of vertices—no "parallel edges" are allowed. We'll also assume that the graph is "connected." We'll define this concept formally in Section 8.3; intuitively, it means that the graph is "in one piece," with no way to break it into two parts without any edges crossing between the parts. The graphs in Figures 7.1(b) and 7.2(a) are connected, while the graph in Figure 7.3 is not.

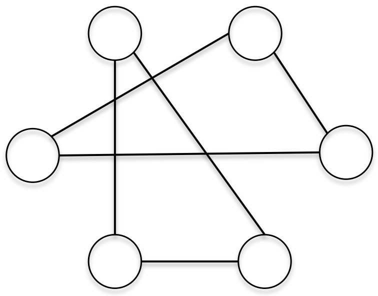


Figure 7.3: An undirected graph that is not connected.

Quiz 7.1

Consider an undirected graph with n vertices and no parallel edges. Assume that the graph is connected, meaning "in one piece." What are the minimum and maximum numbers of edges, respectively, that the graph could have?

[3]For a finite set S, $|S|$ denotes the number of elements in S.

a) $n-1$ and $\frac{n(n-1)}{2}$

b) $n-1$ and n^2

c) n and 2^n

d) n and n^n

(See Section 7.3.3 for the solution and discussion.)

7.3.2 Sparse vs. Dense Graphs

Now that Quiz 7.1 has you thinking about how the number of edges of a graph can vary with the number of vertices, we can discuss the distinction between *sparse* and *dense* graphs. The difference is important because some data structures and algorithms are better suited for sparse graphs, and others for dense graphs.

Let's translate the solution to Quiz 7.1 into asymptotic notation.[4] First, if an undirected graph with n vertices is connected, the number of edges m is at least linear in n (that is, $m = \Omega(n)$).[5] Second, if the graph has no parallel edges, then $m = O(n^2)$.[6] We conclude that the number of edges in a connected undirected graph with no parallel edges is somewhere between linear and quadratic in the number of vertices.

Informally, a graph is *sparse* if the number of edges is relatively close to linear in the number of vertices, and *dense* if this number is closer to quadratic in the number of vertices. For example, graphs with n vertices and $O(n \log n)$ edges are usually considered sparse, while those with $\Omega(n^2/\log n)$ edges are considered dense. "Partially dense" graphs, like those with $\approx n^{3/2}$ edges, may be considered either sparse or dense, depending on the specific application.

7.3.3 Solution to Quiz 7.1

Correct answer: (a). In a connected undirected graph with n vertices and no parallel edges, the number m of edges is at least $n-1$

[4]See Appendix C for a review of big-O, big-Omega, and big-Theta notation.

[5]If the graph need not be connected, there could be as few as zero edges.

[6]If parallel edges are allowed, a graph with at least two vertices can have an arbitrarily large number of edges.

and at most $n(n-1)/2$. To see why the lower bound is correct, consider a graph $G = (V, E)$. As a thought experiment, imagine building up G one edge at a time, starting from the graph with vertices V and no edges. Initially, before any edges are added, each of the n vertices is completely isolated, so the graph trivially has n distinct "pieces." Adding an edge (v, w) has the effect of fusing the piece containing v with the piece containing w (Figure 7.4). Thus, each edge addition decreases the number of pieces by at most 1.[7] To get down to a single piece from n pieces, you need to add at least $n-1$ edges. There are plenty of connected graphs that have n vertices and only $n-1$ edges—these are called *trees* (Figure 7.5).

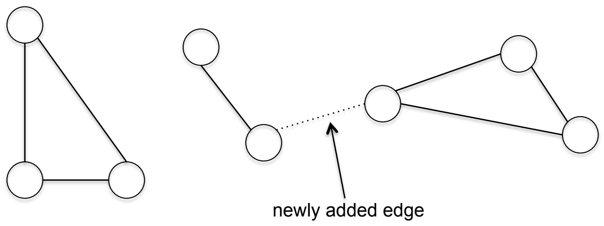
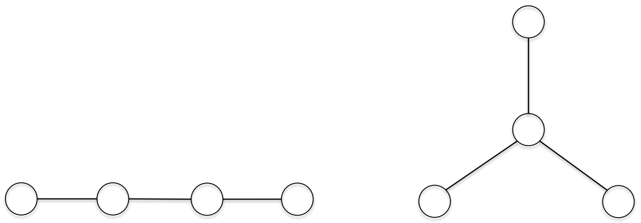


Figure 7.4: Adding a new edge fuses the pieces containing its endpoints into a single piece. In this example, the number of different pieces drops from three to two.



(a) A path on four vertices (b) A star on four vertices

Figure 7.5: Two connected undirected graphs with four vertices and three edges.

The maximum number of edges in a graph with no parallel edges is achieved by the *complete graph*, with every possible edge present.

[7]If both endpoints of the edge are already in the same piece, the number of pieces doesn't decrease at all.

Because there are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of vertices in an n-vertex graph, this is also the maximum number of edges. For example, when $n = 4$, the maximum number of edges is $\binom{4}{2} = 6$ (Figure 7.6).[8]

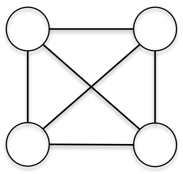


Figure 7.6: The complete graph on four vertices has $\binom{4}{2} = 6$ edges.

7.4 Representing a Graph

There is more than one way to encode a graph for use in an algorithm. In this book series, we'll work primarily with the "adjacency list" representation of a graph (Section 7.4.1), but you should also be aware of the "adjacency matrix" representation (Section 7.4.2).

7.4.1 Adjacency Lists

The *adjacency list* representation of graphs is the dominant one that we'll use in this book series.

Ingredients for Adjacency Lists

1. An array containing the graph's vertices.
2. An array containing the graph's edges.
3. For each edge, a pointer to each of its two endpoints.
4. For each vertex, a pointer to each of the incident edges.

[8] $\binom{n}{2}$ is pronounced "n choose 2," and is also sometimes referred to as a "binomial coefficient." To see why the number of ways to choose an unordered pair of distinct objects from a set of n objects is $n(n-1)/2$, think about choosing the first object (from the n options) and then a second, distinct object (from the $n-1$ remaining options). The $n(n-1)$ resulting outcomes produce each pair (x, y) of objects twice (once with x first and y second, once with y first and x second), so there must be $n(n-1)/2$ pairs in all.

The adjacency list representation boils down to two arrays (or linked lists, if you prefer): one for keeping track of the vertices, and one for the edges. These two arrays cross-reference each other in the natural way, with each edge associated with pointers to its endpoints and each vertex with pointers to the edges for which it is an endpoint.[9]

For a directed graph, each edge keeps track of which endpoint is the tail and which endpoint is the head. Each vertex v maintains two arrays of pointers, one for the outgoing edges (for which v is the tail) and one for the incoming edges (for which v is the head).

What are the memory requirements of the adjacency list representation?

Quiz 7.2

How much space does the adjacency list representation of a graph require, as a function of the number n of vertices and the number m of edges?

a) $\Theta(n)$

b) $\Theta(m)$

c) $\Theta(m+n)$

d) $\Theta(n^2)$

(See Section 7.4.4 for the solution and discussion.)

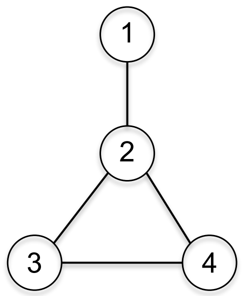
7.4.2 The Adjacency Matrix

Consider an undirected graph $G = (V, E)$ with n vertices and no parallel edges, and label its vertices $1, 2, 3, \ldots, n$. The *adjacency matrix* representation of G is a square $n \times n$ matrix A—equivalently, a two-dimensional array—with only zeroes and ones as entries. Each entry A_{ij} is defined as

$$A_{ij} = \begin{cases} 1 & \text{if edge } (i,j) \text{ belongs to } E \\ 0 & \text{otherwise.} \end{cases}$$

[9]By a *pointer*, we mean a reference to an object (as opposed to the object itself). For example, if you pass an object to a subroutine in Python or Java, it works only with a pointer to that object (as opposed to with its own local copy).

Thus, an adjacency matrix maintains one bit for each pair of vertices, which keeps track of whether or not the edge is present (Figure 7.7).



(a) A graph...

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 0 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 \\ 4 & 0 & 1 & 1 & 0 \end{array}$$

(b) ...and its adjacency matrix

Figure 7.7: The adjacency matrix of a graph maintains one bit for each vertex pair, indicating whether or not there is an edge connecting the two vertices.

It's easy to add bells and whistles to the adjacency matrix representation of a graph:

- *Parallel edges.* If a graph can have multiple edges with the same pair of endpoints, then A_{ij} can be defined as the number of edges with endpoints i and j.

- *Weighted graphs.* Similarly, if each edge (i, j) has a weight w_{ij}—perhaps representing a cost or a distance—then each entry A_{ij} stores w_{ij}.

- *Directed graphs.* For a directed graph G, each entry A_{ij} of the adjacency matrix is defined as

$$A_{ij} = \begin{cases} 1 & \text{if edge } (i, j) \text{ belongs to } E \\ 0 & \text{otherwise,} \end{cases}$$

  where "edge (i, j)" now refers to the edge directed from i to j. Every undirected graph has a symmetric adjacency matrix, while a directed graph usually has an asymmetric adjacency matrix.

What are the memory requirements of an adjacency matrix?

Quiz 7.3

How much space does the adjacency matrix of a graph require, as a function of the number n of vertices and the number m of edges?

a) $\Theta(n)$

b) $\Theta(m)$

c) $\Theta(m+n)$

d) $\Theta(n^2)$

(See Section 7.4.4 for the solution and discussion.)

7.4.3 Comparing the Representations

Confronted with two different ways to represent a graph, you're probably wondering: Which is better? The answer, as it so often is with such questions, is "it depends." First, it depends on the density of your graph—on how the number m of edges compares to the number n of vertices. The moral of Quizzes 7.2 and 7.3 is that the adjacency matrix is an efficient way to encode a dense graph but is wasteful for a sparse graph. Second, it depends on which operations you want to support. On both counts, adjacency lists make more sense for the algorithms and applications described in this book series.

Most of our graph algorithms will involve exploring a graph. Adjacency lists are perfect for graph exploration—you arrive at a vertex, and the adjacency list immediately indicates your options for the next step.[10] Adjacency matrices do have their applications, but we won't see them in this book series.[11]

Much of the modern-day interest in fast graph primitives is motivated by massive sparse networks. Consider, for example, the Web graph (Section 7.2), in which vertices correspond to Web pages and directed edges to hyperlinks. It's hard to get an exact measurement of

[10]If you had access to only the adjacency matrix of a graph, how long would it take you to figure out which edges are incident to a given vertex?

[11]For example, you can count the number of common neighbors of each pair of vertices in one fell swoop by squaring the graph's adjacency matrix.

the size of this graph, but a conservative lower bound on the number of vertices is 10 billion, or 10^{10}. Storing and reading through an array of this length already requires significant computational resources, but it is well within the limits of what modern computers can do. The size of the adjacency matrix of this graph, however, is proportional to 100 quintillion (10^{20}). This is way too big to store or process with today's technology. But the Web graph is sparse—the average number of outgoing edges from a vertex is well under 100. The memory requirements of the adjacency list representation of the Web graph are therefore proportional to 10^{12} (a trillion). This may be too big for your laptop, but it's within the capabilities of state-of-the-art data-processing systems.[12]

7.4.4 Solutions to Quizzes 7.2–7.3

Solution to Quiz 7.2

Correct answer: (c). The adjacency list representation requires space linear in the size of the graph (meaning the number of vertices plus the number of edges), which is ideal.[13] Seeing this is a little tricky. Let's step through the four ingredients one by one. The vertex and edge arrays have lengths n and m, respectively, and so require $\Theta(n)$ and $\Theta(m)$ space. The third ingredient associates two pointers with each edge (one for each endpoint). These $2m$ pointers contribute an additional $\Theta(m)$ to the space requirement.

The fourth ingredient might make you nervous. After all, each of the n vertices can participate in as many as $n-1$ edges—one per other vertex—seemingly leading to a bound of $\Theta(n^2)$. This quadratic bound would be accurate in a very dense graph, but is overkill in sparser graphs. The key insight is: *For every vertex$\rightarrow$edge pointer in the fourth ingredient, there is a corresponding edge$\rightarrow$vertex pointer in the third ingredient.* If the edge e is incident to the vertex v, then e has a pointer to its endpoint v, and, conversely, v has a pointer to the incident edge e. We conclude that the pointers in the third and fourth ingredients are in one-to-one correspondence, and so they require

[12]For example, the essence of Google's original `PageRank` algorithm for measuring Web page importance relied on efficient search in the Web graph.

[13]Caveat: The leading constant factor here is larger than that for the adjacency matrix by an order of magnitude.

exactly the same amount of space, namely $\Theta(m)$. The final scorecard is:

	vertex array	$\Theta(n)$
	edge array	$\Theta(m)$
	pointers from edges to endpoints	$\Theta(m)$
$+$	pointers from vertices to incident edges	$\Theta(m)$
	total	$\Theta(m+n)$.

The bound of $\Theta(m+n)$ applies whether or not the graph is connected, and whether or not it has parallel edges.[14]

Solution to Quiz 7.3

Correct answer: (d). The straightforward way to store an adjacency matrix is as an $n \times n$ two-dimensional array of bits. This uses $\Theta(n^2)$ space, albeit with a small hidden constant. For a dense graph, in which the number of edges is itself close to quadratic in n, the adjacency matrix requires space close to linear in the size of the graph. For sparse graphs, however, in which the number of edges is closer to linear in n, the adjacency matrix representation is highly wasteful.[15]

The Upshot

- ✰ A graph is a representation of the pairwise relationships between objects, such as friendships in a social network, hyperlinks between Web pages, or dependencies between tasks.
- ✰ A graph comprises a set of vertices and a set of edges. Edges are unordered in undirected graphs and ordered in directed graphs.
- ✰ A graph is sparse if the number of edges m is close to linear in the number of vertices n, and dense if m is close to quadratic in n.

[14]If the graph is connected, then $m \geq n-1$ (by Quiz 7.1), and we could write $\Theta(m)$ in place of $\Theta(m+n)$.

[15]This waste can be reduced by using tricks for storing and manipulating sparse matrices, meaning matrices with lots of zeroes. For instance, both Matlab and Python's SciPy package support sparse matrix representations.

- ✰ The adjacency list representation of a graph maintains vertex and edge arrays, cross-referencing each other in the natural way, and requires space linear in the total number of vertices and edges.
- ✰ The adjacency matrix representation of a graph maintains one bit per pair of vertices to keep track of which edges are present, and requires space quadratic in the number of vertices.
- ✰ The adjacency list representation is the preferred one for sparse graphs, and for applications that involve graph exploration.

Test Your Understanding

Problem 7.1 *(S)* Consider a directed graph with n vertices and no parallel edges. What is the maximum numbers of edges that the graph could have?

a) $n(n-1)/2$

b) $n^2/2$

c) $n(n-1)$

d) n^2

Problem 7.2 *(S)* Let $G = (V, E)$ be an undirected graph. By the *degree* of a vertex $v \in V$, we mean the number of edges in E that are incident to v (i.e., that have v as an endpoint).[16] What is the sum of the degrees of all of G's vertices, as a function of n and m? (As usual, in this and subsequent problems, n and m denote the number of vertices and edges, respectively.)

a) m

b) $m + n$

[16]The abbreviation "i.e." stands for *id est*, and means "that is."

c) $2m$

d) n^2

Problem 7.3 *(S)* For each of the following conditions on a graph $G = (V, E)$, is the condition satisfied only by dense graphs, only by sparse graphs, or by both some sparse and some dense graphs? Assume that the number n of vertices is large (say, at least 10,000).

a) At least one vertex of G has degree at most 10.

b) Every vertex of G has degree at most 10.

c) At least one vertex of G has degree $n - 1$.

d) Every vertex of G has degree $n - 1$.

Problem 7.4 *(S)* Consider an undirected graph $G = (V, E)$ that is represented as an adjacency matrix. Given a vertex $v \in V$, how many operations are required to identify the edges incident to v? (Let k denote the number of such edges.)

a) $\Theta(1)$

b) $\Theta(k)$

c) $\Theta(n)$

d) $\Theta(m)$

Problem 7.5 *(S)* Consider a directed graph $G = (V, E)$ that is represented with adjacency lists, except with each vertex storing only an array of its outgoing edges (and *not* its incoming edges). Given a vertex $v \in V$, how many operations are required to identify the incoming edges of v? (Let k denote the number of such edges.)

a) $\Theta(1)$

b) $\Theta(k)$

c) $\Theta(n)$

d) $\Theta(m)$

Chapter 8

Graph Search and Its Applications

This chapter is all about fundamental primitives for graph search and their applications. One very cool aspect of this material is that all the algorithms that we'll cover are blazingly fast (linear time with small constants), and it can be quite tricky to understand why they work! The culmination of this chapter—computing the strongly connected components of a directed graph with only two passes of depth-first search (Section 8.6)—vividly illustrates how fast algorithms often require deep insight into the problem structure.

We begin with an overview section (Section 8.1), which covers some reasons why you should care about graph search, a general strategy for searching a graph without doing any redundant work, and a high-level introduction to the two most important search strategies, breadth-first search (BFS) and depth-first search (DFS). Sections 8.2 and 8.3 describe BFS in more detail, including applications to computing shortest paths and the connected components of an undirected graph. Sections 8.4 and 8.5 drill down on DFS and how to use it to compute a topological ordering of a directed acyclic graph (equivalently, to sequence tasks while respecting precedence constraints). Section 8.6 uses DFS to compute the strongly connected components of a directed graph in linear time. Section 8.7 explains how this fast graph primitive can be used to explore the structure of the Web.

8.1 Overview

This section provides a bird's-eye view of algorithms for graph search and their applications.

8.1.1 Some Applications

Why would we want to search a graph, or to figure out if there's a path from point A to point B? Here are a few of the many reasons.

Checking connectivity. In a physical network, such as a road network or a network of computers, an important sanity check is that you can get anywhere from anywhere else. That is, for every choice of a point A and a point B, there should be a path in the network from the former to the latter.

Connectivity can also be important in abstract (non-physical) graphs that represent pairwise relationships between objects. One network that's fun to play with is the movie network, in which vertices correspond to movie actors, and two actors are connected by an undirected edge whenever they appeared in the same movie.[1] For example, how many "degrees of separation" are there between different actors? The most famous statistic of this type is the *Bacon number*, which is the minimum number of hops through the movie network needed to reach the fairly ubiquitous actor Kevin Bacon.[2] So, Kevin Bacon himself has a Bacon number of 0, every actor who has appeared in a movie with Kevin Bacon has a Bacon number of 1, every actor who has appeared with an actor whose Bacon number is 1 (and does not have a Bacon number of 0 or 1) has a Bacon number of 2, and so on. For example, Jon Hamm—perhaps best known as Don Draper from the cable television series *Mad Men*—has a Bacon number of 2. Hamm never appeared in a movie with Bacon, but he did have a bit part in the Colin Firth vehicle *A Single Man*, and Firth and Bacon co-starred in Atom Egoyan's *Where the Truth Lies* (Figure 8.1).[3]

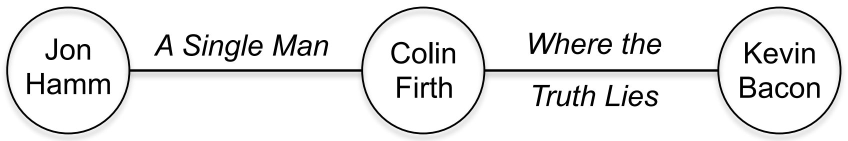


Figure 8.1: A snippet of the movie network, showing that Jon Hamm's Bacon number is at most 2.

[1]`https://oracleofbacon.org/`

[2]The Bacon number is a riff on the older concept of the *Erdös number*, named after the famous mathematician Paul Erdös, which measures the number of degrees of separation from Erdös in the co-authorship graph (in which vertices are researchers, and there is an edge between each pair of researchers who have co-authored a paper).

[3]There are also lots of other two-hop paths between Bacon and Hamm in the movie network.

Shortest paths. The Bacon number concerns the *shortest* path between two vertices of the movie network, meaning the path using the fewest number of edges. We'll see in Section 8.2 that a graph search strategy known as breadth-first search naturally computes shortest paths.

Plenty of other problems boil down to a shortest-path computation, where the definition of "short" depends on the application (minimizing time for driving directions, or money for airline tickets, and so on). Dijkstra's shortest-path algorithm, the subject of Chapter 9, builds on breadth-first search to solve more general shortest-path problems.

Planning. A path in a graph need not represent a physical path through a physical network. More abstractly, a path is a sequence of decisions taking you from one state to another. Graph search algorithms can be applied to such abstract graphs to compute a plan for reaching a goal state from an initial state.

For example, imagine you want to use an algorithm to solve a Sudoku puzzle. Think of the graph in which vertices correspond to partially completed Sudoku puzzles (with some of the 81 squares blank, but no rules of Sudoku violated), and directed edges correspond to filling in one new entry of the puzzle (subject to the rules of Sudoku). The problem of computing a solution to the puzzle is exactly the problem of computing a directed path from the vertex corresponding to the initial state of the puzzle to the vertex corresponding to the completed puzzle.[4]

For another example, using a robotic hand to grasp a coffee mug is essentially a planning problem. In the associated graph, vertices correspond to the possible configurations of the hand, and edges correspond to small and realizable changes in the configuration.

Connected components. We'll also see algorithms that build on graph search to compute the connected components (the "pieces") of a graph. Defining and computing the connected components of an undirected graph is relatively easy (Section 8.3). For directed graphs, even defining what a "connected component" should mean is a little subtle. Section 8.6 defines them and shows how to use depth-first search (Section 8.4) to compute them efficiently. We'll also

[4]Because this graph is too big to write down explicitly, practical Sudoku solvers incorporate some additional ideas.

see applications of depth-first search to sequencing tasks (Section 8.5) and to understanding the structure of the Web graph (Section 8.7).

8.1.2 For-Free Graph Primitives

The examples in Section 8.1.1 demonstrate that graph search is a fundamental and widely applicable primitive. I'm happy to report that, in this chapter, all our algorithms will be blazingly fast, running in just $O(m + n)$ time with small constant factors, where m and n denote the number of edges and vertices of the graph. That's only a constant factor larger than the amount of time required to read the input![5] These algorithms are "for-free primitives"—whenever you have graph data, you should immediately consider applying one or more of these primitives to glean information about what it looks like.[6]

For-Free Primitives

You can think of an algorithm with a linear or near-linear running time as a primitive that can be used essentially "for free"—the amount of time required barely exceeds what you need to read the input. When you have a primitive relevant to your problem that is so blazingly fast, why not use it? For example, you can always compute the connected components of your graph data in a preprocessing step, even if you're not quite sure how it will help later. One of the goals of this book series is to stock your algorithmic toolbox with as many for-free primitives as possible, ready to be applied at will.

8.1.3 Generic Graph Search

The point of a graph search algorithm is to solve the following problem.

[5]In graph search and connectivity problems, there is no reason to expect that the input graph is connected. In the disconnected case, in which m might be much smaller than n, the size of a graph is $\Theta(m + n)$ but not necessarily $\Theta(m)$.

[6]Can we do better? No, up to the hidden constant factor: Every correct algorithm must at least read the entire input in some cases.