11 Graphs and Trees

In which our heroes explore the many twisting paths through the gnarled forest, emerging in the happy and peaceful land in which their computational adventures will continue.

11.1 Why You Might Care

> Oh what a tangled web we weave,
> When first we practise to deceive!
>
> Sir Walter Scott (1771–1832)
> *Marmion* (1808)

In computer science, a *graph* means a network: a collection of things (people, web pages, subway stations, animal species, . . .) where some pairs of those things are joined by some kind of pairwise relationship (spent more than 15 minutes inside an enclosed space with, has a [hyper]link to, is the stop before/after on some subway line, is a predator of, . . .). It's possible to make graphs sound hopelessly abstract and utterly uninteresting—*a* graph *is a pair* $\langle V, E \rangle$*, where* V *is a nonempty collection of entities called* nodes *and* E *is a collection of* edges *that join pairs of nodes*—but graphs are fascinating whenever the entities and the relationship represented by the edges are themselves interesting! Here are just a few examples.

In a *social network,* the nodes are people, and an edge between two people represents a friendship (or whatever friendship-like relationship is represented in an online social networking sites). Figure 11.1 shows an example. Or we could define such a graph based on communication—an edge between two people might represent that they have texted each other within the last year—or as a *contact* network, where an edge between two people represents close physical interaction (for example, the kind of proximity and duration that would count as a "close contact" for the purpose of possible disease transmission).

In a *road network,* edges represent roads, and nodes represent intersections of roads. A graph like this one is what your GPS operates on when it finds driving directions: the shortest sequence of edges that gets you from where you are to where you're going. (Sometimes "shortest" can be a little subtle; for example, some delivery companies save fuel and money by using route-finding algorithms that avoid turns that cross traffic [82].) Similarly, the route map for an airline forms another kind of transportation network.

We can also think about technological networks, like the internet—nodes are computers (laptops, phones, servers, routers, etc.), and edges represent wires or radio signals connecting two machines together—or *the web,* in which nodes are web pages and an edge represents a link from one page to another. And there are many other kinds of networks, too: dating networks (and sexual contact networks), food webs

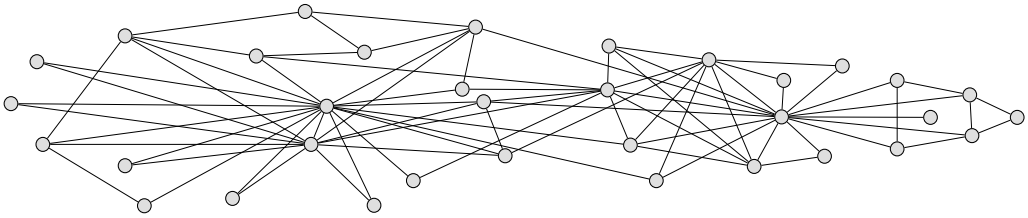


Figure 11.1 A sample social network (in this case, among the 34 members of a karate club in the 1970s [132]).

(what species eats what other species?), purchase networks (nodes correspond to people and products, and an edge joins a person to a product they bought), among many more.

In this chapter, we'll begin with a bunch of underlying definitions that are related to graphs (Section 11.2), and then move on to *paths,* sequences of hops to get from one node in a graph to another (Section 11.3). We'll then look at *trees* (Section 11.4), which are a special type of graph, and *weighted* graphs, in which different edges have different costs or lengths (Section 11.5). And we'll spend a little time with a few of the ubiquitous applications of graphs as we go.

11.2 Formal Introduction

> The Bible tells us to love our neighbors, and also to love our enemies; probably because they are generally the same people.
>
> G. K. Chesterton (1874–1936)
> *Illustrated London News* (1910)

We begin by defining the terminology for the two different basic types of graphs. In both, we have a set of entities called *nodes,* some pairs of which are joined by a relationship called an *edge.* (A node can also be called a *vertex,* and we'll use the terms *node/nodes* and *vertex/vertices* interchangeably; they're both used commonly in CS. A graph is also sometimes called a *network;* edges are also sometimes called *links*.)

The two types of graph differ in whether the relationship represented by an edge is "between two nodes" or "from one node to another." In an *undirected graph*, the relationship denoted by the edges is symmetric (for example, "u and v are genetically related"):

Definition 11.1: Undirected graph.
A *undirected graph* is a pair $G = \langle V, E \rangle$ where V is a nonempty set of *vertices* or *nodes*, and $E \subseteq \big\{ \{u, v\} : u, v \in V \big\}$ is a set of *edges* joining pairs of vertices.

The second basic kind of graph is a *directed graph*, in which the relationship denoted by the edges need not be reciprocated (for example, "u has texted v"):

Definition 11.2: Directed graph.
A *directed graph* is a pair $G = \langle V, E \rangle$ where V is a nonempty set of nodes, and $E \subseteq V \times V$ is a set of edges joining (ordered) pairs of vertices.

In other words, in a directed graph an edge is an *ordered* pair of vertices ("an edge from u to v") and in an undirected graph an edge is an *unordered* pair of vertices ("an edge between u and v"). Think about the difference between Twitter followers (directed) and Facebook friendships (undirected): Alice can follow Bob without Bob following Alice, but they're either friends or they're not friends.

Graphs are generally drawn with nodes represented as circles, and edges represented by lines. Each edge in directed graphs is drawn with an arrow indicating its *orientation* ("which way it goes"). Here is an example of each:

Example 11.1: A sample undirected graph.
A small undirected graph is shown in Figure 11.2a. This graph contains:

- 12 nodes: `A`, `B`, `C`, `D`, `E`, `F`, `G`, `H`, `I`, `J`, `K`, and `L`.
- 10 edges: {`A`, `B`}, {`B`, `C`}, {`C`, `D`}, {`E`, `F`}, {`E`, `H`}, {`F`, `G`}, {`G`, `H`}, {`I`, `J`}, {`J`, `K`}, and {`K`, `L`}.

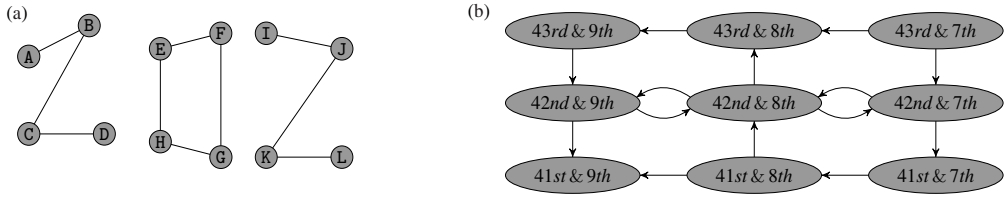


Figure 11.2 Two small graphs: (a) one undirected graph and (b) one directed graph (a small portion of Manhattan).

Figure 11.3 Parallel edges and self-loops, in (a) undirected and (b) directed graphs.

Example 11.2: Streets of Manhattan: a sample directed graph.
The directed graph shown in Figure 11.2b contains 9 nodes, each corresponding to an intersection of a "street" running east–west and an "avenue" running north–south in Manhattan. There are 14 edges in this graph. There's something potentially tricky in counting to 14: edges in a directed graph are *ordered* pairs, which means that there are *two* edges between $42nd \,\&\, 9th$ and $42nd \,\&\, 8th$, one in each direction—$\langle 42nd \,\&\, 9th, 42nd \,\&\, 8th\rangle$ and $\langle 42nd \,\&\, 8th, 42nd \,\&\, 9th\rangle$. The pair of nodes $42nd \,\&\, 8th$ and $42nd \,\&\, 7th$ is similar.

For many of the concepts that we'll explore in this chapter, it will turn out that there are no substantive differences between the ideas for directed and undirected graphs. To avoid being tedious and unhelpfully repetitive, whenever it's possible we'll state definitions and results about both undirected and directed graphs simultaneously. But doing so will require a little abuse of notation: we'll allow ourselves to write an edge as an ordered pair $\langle u, v\rangle$ *even for an undirected graph.* In an undirected graph, we will agree to understand both $\langle u, v\rangle$ and $\langle v, u\rangle$ as meaning $\{u, v\}$.

Simple graphs

For many of the real-world phenomena that we will be interested in modeling, it will make sense to make a simplifying assumption about the edges in our graphs. Specifically, we will typically restrict our attention to so-called *simple* graphs, which forbid two different kinds of edges: edges that connect nodes to themselves, and edges that are precise duplicates of other existing edges.

Definition 11.3: Self-loops and parallel edges.
A *self-loop* is an edge from a node u to itself. Two edges are called *parallel* if they both go from same node u and both go to the same node v.

(See Figure 11.3.) Note that the edges $\langle u, v\rangle$ and $\langle v, u\rangle$ are not parallel in a directed graph: directed edges are parallel only if they both go *from* the same node and *to* the same node, in the same orientation. (Edges $\langle u, v\rangle$ and $\langle v, u\rangle$ are sometimes called *antiparallel.*)

Definition 11.4: Simple graph.
A graph is *simple* if it contains no parallel edges and no self-loops.

Throughout, we'll assume that graphs are undirected and simple unless otherwise noted.

In general, the particular real-world phenomenon that we seek to model will dictate whether self-loops, parallel edges, or both will make sense. Here are a few examples:

Example 11.3: Self-loops and parallel edges.
Suppose that we construct a graph to model each of the following phenomena. In which settings do self-loops or parallel edges make sense?

1. A social network: nodes correspond to people; (undirected) edges represent friendships.
2. The web: nodes correspond to web pages; (directed) edges represent links.
3. The flight network for a commercial airline: nodes correspond to airports; (directed) edges denote flights scheduled by the airline in the next month.
4. The email network at a college: nodes correspond to students; there is a (directed) edge $\langle u, v \rangle$ if u has sent at least one email to v within the last year.

Solution. *A social network:* Neither self-loops nor parallel edges make sense. A self-loop would correspond to a person being a friend of himself, and parallel edges between two people would correspond to them being friends "twice." (But two people are either friends or not friends.)

The web: Both self-loops and parallel edges are reasonable. It is easy to imagine a web page p that contains a hyperlink to p itself. It is also easy to imagine a web page p that contains two separate links to another web page q. (For example, as of this writing, the "CNN" logo on `www.cnn.com` links to `www.cnn.com`. And, as of the end of this sentence, this page has three distinct references to `www.cnn.com`.)

A commercial flight network: In a flight network, many parallel edges will exist: there are generally many scheduled commercial flights from one airport to another—for example, there are dozens of flights every week from BOS (Boston, MA) to SFO (San Francisco, CA) on most major airlines. However, there are no self-loops: a commercial flight from an airport back to the same airport doesn't go anywhere!

A who-emailed-whom network: Self-loops are reasonable but parallel edges are not. A student u has either sent email to v in the last year or she has not, so parallel edges don't make sense in this network. However, self-loops exist if any student has sent an email to herself (as many people do to remind themselves to do something later).

Taking it further: Technically speaking, the way that we phrased our definitions of graphs in Definitions 11.1 and 11.2 doesn't even *allow* us to consider parallel edges. (Our definitions do allow self-loops, though.) That's because we defined the edges as a subset E of $V \times V$ or $\{\{u, v\} : u, v \in V\}$, and sets don't allow duplication—which means that we can't have $\langle u, v \rangle$ in E "twice." There are alternate ways to formalize graphs that do permit parallel edges, but they're needlessly complicated for the applications that we'll focus on in this chapter.

11.2.1 Neighborhoods and Degree

Imagine a social network in which two people, Ursula and Victor, are friends—or, more generally, imagine an undirected graph in which nodes u and v are joined by an edge. Here's the vocabulary for referring to these nodes and the edge between them:

Definition 11.5: Adjacency, neighbors, endpoints, incidence.
For an edge $e = \{u, v\}$ in an undirected graph (see Figure 11.4a), we say that:

- the nodes u and v are *adjacent*;
- the node v is a *neighbor* of the node u (and vice versa);
- the nodes u and v are the *endpoints* of the edge e; and
- the nodes u and v are both *incident* to the edge e.

It's important to distinguish between two distinct concepts: first, the *direct* connection between two nodes u and v that are adjacent—that is, a single edge that joins u and v directly; and, second, an *indirect* connection between two nodes that follows a sequence of edges. At the moment, we're talking *only* about the first kind, a direct connection via a single edge. (A multihop connection is called a *path*; we'll talk about paths in Section 11.3.)

Here's an example of the vocabulary from Definition 11.5:

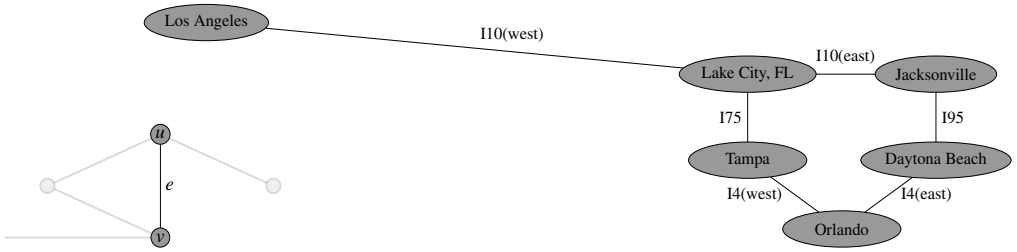
Example 11.4: Disney World to Disney Land.
A small portion of the U.S. Interstate system is shown in Figure 11.4b. In this graph:

- Orlando is adjacent to Tampa and Daytona Beach.
- None of the other nodes (Lake City, Jacksonville, Los Angeles) is a neighbor of Orlando. Orlando is also not a neighbor of itself.
- The endpoints of edge I75 are Tampa and Lake City.
- Jacksonville is incident to I95, as is Daytona Beach.

The *neighborhood* of a node is the set of all nodes adjacent to it:

Definition 11.6: Neighborhood.
Let $G = \langle V, E \rangle$ be an undirected graph, and let $u \in V$ be a node. The *neighborhood* of u is the set $\{v \in V : \{u, v\} \in E\}$—that is, the set of all neighbors of u.

For example, in the graph in Figure 11.4b, the neighborhood of Lake City is {Los Angeles, Tampa, Jacksonville}. Or, for a graph G that represents a social network, the neighborhood of a node u is the set of people who are u's friends.



(a) Two nodes in an undirected graph joined by an edge.

(b) A small portion of the U.S. Interstate system between Orlando, FL and Los Angeles, CA. Each of the roads is labeled by its name.

Figure 11.4 A schematic view of edges joining nodes in an undirected graph, and an example.

Degree

It's also common to refer the *number* of neighbors that a node has (without reference to which particular nodes happen to be that node's neighbors):

Definition 11.7: Degree.
The *degree* of a node u in an undirected graph G is the size of the neighborhood of u in G—that is, the number of nodes adjacent to u.

For example, in the graph in Figure 11.4b, Lake City has degree 3 and Los Angeles has degree 1. (See Figure 11.5 for a version of the road network with each node labeled by its degree.) Or, in a social network, the degree of a node u is the popularity of u—the number of friends that u has. Here are a few practice questions:

Example 11.5: Neighborhood and degree.
Here are a few questions about the graph in Figure 11.6:

1. What are the neighbors of node `C`?
2. What nodes, if any, have degree equal to one?
3. What node has the highest degree in this graph?
4. What nodes, if any, are in the neighborhoods of both nodes `B` and `E`?

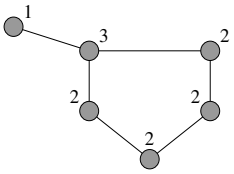


Figure 11.5 The road network from Figure 11.4b, with nodes labeled by their degree.

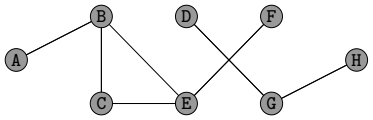


Figure 11.6 An undirected graph, for some practice questions.

Solution. By inspecting Figure 11.6, we see:

Node `C` has two neighbors, namely the nodes `B` and `E`.

The nodes with degree one are those with precisely one neighbor. These nodes are: `A`, `D`, `F`, and `H`. (Their solitary neighbors are, respectively: `B`, `G`, `E`, and `G`.)

We simply count neighbors for each node, and we find that nodes `B` and `E` both have degree three, and are tied as the nodes with the highest degree.

The neighborhood of node `B` is $\{\texttt{A}, \texttt{C}, \texttt{E}\}$, and the neighborhood of node `E` is $\{\texttt{B}, \texttt{C}, \texttt{F}\}$. Taking the intersection of those sets yields the one node in the neighborhood of both `B` and `E`, namely node `C`.

Taking it further: Consider a population of people—say, the current residents of Canada—represented as a social network, in an undirected graph whose edges represent friendship. For a node in the social network (also known as a person), we can calculate many numbers that may be interesting: height, age, income, number of cigarettes smoked per day, self-reported happiness, etc. Then, for any one of these numerical properties, we can consider the *distribution* over the population: for example, the distribution of heights, or the distribution of ages. (The height distribution will follow a roughly bell-shaped curve; the age distribution is more complicated, both because of death and because of variation in the birth rate over time.) Another interesting numerical property of a person u is the *degree* of u: that is, the number of friends that u has. The *degree distribution* of a graph describes how popularity varies across the nodes of the network. The degree distribution has some interesting properties—very different from the distribution of heights or ages. See p. 11-26.

The Handshaking Lemma

Before we move on from degree, we'll prove a basic but valuable fact, colloquially called the "handshaking lemma." (We can represent a group of people, some pairs of whom shake hands, using an undirected graph: an edge joins u and v if and only if u and v shook hands; the theorem describes the number of shakes.) The handshaking lemma relates the sum of nodes' degrees to the number of edges in the graph:

Theorem 11.8: "Handshaking Lemma".
Let $G = \langle V, E \rangle$ be an undirected graph. Then

$$\sum_{u \in V} degree(u) = 2|E|.$$

For example, Figure 11.5 shows our road network from Figure 11.4b, with all nodes labeled by their degree. This graph has $|E| = 6$ edges, and the sum of the nodes' degrees is $1+3+2+2+2+2 = 12$, and indeed $12 = 2 \cdot 6$. Here is a proof:

Proof of Theorem 11.8. Every edge has two endpoints! (Or, as The Player says to Rosencrantz in Tom Stoppard's brilliant play *Rosencrantz and Guildenstern are Dead*: "look on every exit being an entrance somewhere else" [122].)

Or, more formally, imagine looping over each edge to compute all nodes' degrees:

```
1 initialize d_u to 0 for each node u
2 for each edge {u, v} ∈ E:
3     d_u := d_u + 1
4     d_v := d_v + 1
```

In each iteration of the **for** loop, we increment two different $d_\bullet$ values; thus, after i iterations, we have that $\sum_u d_u = 2i$. (We could give a fully rigorous proof of this fact by induction.) We complete $|E|$ iterations of the **for** loop, one for each edge, and thus at the end of the algorithm we have that $\sum_{u \in V} d_u = 2|E|$. Furthermore, after the loop, it's clear that $d_u = degree(u)$ for every node u. Thus

$$\sum_{u \in V} d_u = \sum_{u \in V} degree(u) = 2|E|. \qquad \square$$

Here's a useful corollary of Theorem 11.8 (the proof is left to you as Exercise 11.17):

Corollary 11.9. Let n_{odd} denote the number of nodes whose degree is odd. Then n_{odd} is even.

(For example, for the road-network graph in Figure 11.5, we have $n_{\text{odd}} = 2$: the two nodes with odd degree are those with degree 1 and 3. And 2 is an even number.)

Neighborhoods and degree: directed graphs

The definitions of adjacency, neighbors, and degree from Definitions 11.5–11.7 were all for *undirected* graphs. Here we'll introduce the analogous notions for directed graphs, all of which are slightly more complicated because they must account for the orientation of each edge. We start with the directed version of "neighbors":

Definition 11.10: Neighbors in directed graphs.
For an edge $\langle u, v \rangle$ from node u to node v in a directed graph, we say that:

- the node v is an *out-neighbor* of the node u; and
- the node u is an *in-neighbor* of the node v.

For example, if G represents a flight network (with nodes as airports and directed edges corresponding to flights), then the out-neighbors of node u are those airports that have direct flights from u, and the in-neighbors of u are those airports that have direct flights to u. (See Figure 11.7.) Now, using these definitions, we can define the analogues of neighborhoods and degree in directed graphs:

Definition 11.11: Neighborhoods and degrees in directed graphs.
For a node u in a directed graph, we say that:

- the *in-neighborhood* of u is $\{v : \langle v, u \rangle \in E\}$, the set of in-neighbors of v;
- the *in-degree* of u is its number of in-neighbors (its in-neighborhood's cardinality);

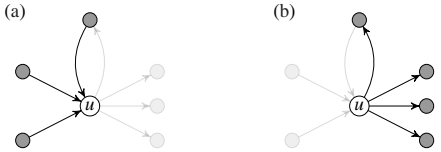


Figure 11.7 The (a) in-neighbors and (b) out-neighbors of a node u.

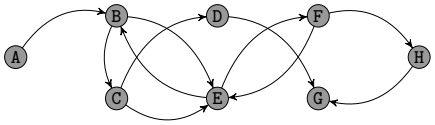


Figure 11.8 A directed graph for some practice questions.

- the *out-neighborhood* of u is $\{v : \langle u, v\rangle \in E\}$, the set of out-neighbors of u; and
- the *out-degree* of u is its number of out-neighbors (its out-neighborhood's cardinality).

Here are a few practice questions about in- and out-neighborhoods:

Example 11.6: Neighborhood and degree in a directed graph.
For the directed graph in Figure 11.8:

1 What are the in-neighbors of node `C`? The out-neighbors of `C`?

2 What nodes, if any, are in both the in-neighborhood and out-neighborhood of node `E`?

3 What nodes, if any, have in-degree zero? Out-degree zero?

Solution. Node `C` has one in-neighbor, namely `B`, and two out-neighbors, namely `D` and `E`.

Node `E` has three in-neighbors (`B`, `C`, and `F`) and two out-neighbors (`B` and `F`). So nodes `B` and `F` are in both `E`'s in-neighborhood and `E`'s out-neighborhood.

Node `A` has no in-neighbors, so `A`'s in-degree is zero. Node `G` has no out-neighbors, so `G`'s out-degree is zero.

11.2.2 Representing Graphs: Data Structures

The graphs that we've considered so far have been presented visually: as a picture, with nodes drawn as circles and edges drawn as lines or arrows. But, of course, when we represent a graph on a computer, we'll need to use some data structure to store a network, not just some image file. Here we will give a brief summary of the two major data structures used to represent graphs. If you've had a course on data structures, then this material may be a review; if not, it will be a preview.

Taking it further: A visual representation is great for some smaller networks, and a well-designed layout can sometimes make even large networks easy to understand at a glance. *Graph drawing* is the problem of algorithmically laying out the nodes of a graph well—in an aesthetic and informative manner. There's a physics analogy that's often used in laying out graphs, in which we imagine nodes "attracting" and "repelling" each other depending on the presence or absence of edges. See p. 11-28 for more, including an application of this graph-drawing idea to the 9/11 Memorial in New York City. Some other gorgeous visualizations of network (and other!) data can be found online at sites like Flowing Data (`http://flowingdata.com/`), Information Is Beautiful (`http://informationisbeautiful.net`), or some of the beautiful books on data visualization (such as [18]).

The most straightforward data structure for a graph is just a list of nodes and a list of edges. But this straightforward representation suffers for some standard, natural questions that are typically asked about graphs. Many of the natural questions that we will find ourselves asking are things like: *What are all of the neighbors of* `A`*?* or *Are* `B` *and* `C` *joined by an edge?* There are two standard data structures for graphs, each of which is tailored to make it possible to answer one of these two questions quickly.

Adjacency lists

The first standard data structure for graphs is an *adjacency list,* which—as the name implies—stores, for each node u, a list of the nodes adjacent to u:

Definition 11.12: Adjacency list.
In an *adjacency list* of a graph $G = \langle V, E \rangle$, for each node $u \in V$, we store an unsorted list of all of u's neighbors in the graph.

The schematic for an adjacency list is illustrated in Figure 11.9: each node in the graph corresponds to a row of the table, which points to an unsorted list of that node's neighbors. (These lists are unsorted so that it's faster to add a new edge to the data structure.)

There's no significant difference between adjacency lists for undirected graphs and for directed graphs: for an undirected graph, we list the *neighbors* for each node u; for a directed graph, we list the *out-neighbors* of each node. (Every edge $\langle u, v \rangle$ in a directed graph appears only once in the data structure, in u's list. Every edge $\{u, v\}$ in an undirected graph is represented twice: v appears in u's list, and u appears in v's list. This observation is another way of thinking of the proof of Theorem 11.8.)

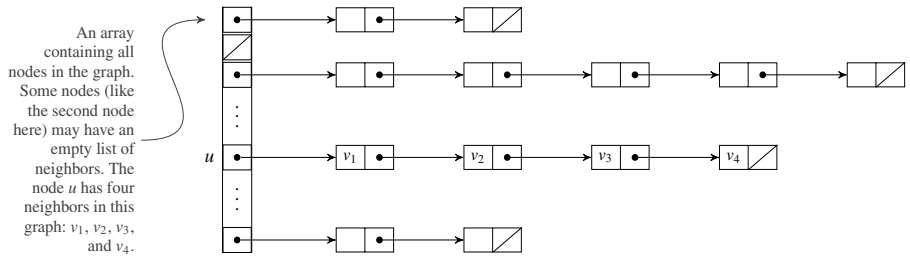


Figure 11.9 A schematic of an adjacency list.

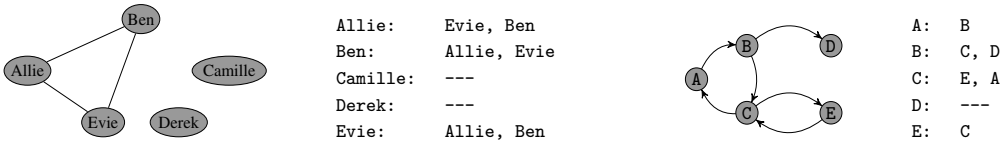


Figure 11.10 Two small graphs, one undirected and one directed, represented by adjacency lists.

Example 11.7: Two sample adjacency lists.
Two small graphs, one undirected and one directed, and their corresponding adjacency lists are shown in Figure 11.10. Note that the order of the (out-)neighbors of any particular node isn't specified: for example, we could just as well said that `Evie`'s neighbors were `[Ben, Allie]` as `[Allie, Ben]`.

Adjacency matrices

The second standard data structure for representing graphs is an *adjacency matrix:*

Definition 11.13: Adjacency matrix.
In an *adjacency matrix* of a graph $G = \langle V, E \rangle$, we store the graph using an $|V|$-by-$|V|$ table. The ith row of the table corresponds to the neighbors of node i. A True (or 1) in column j indicates that the edge $\langle i, j \rangle$ is in E; a False (or 0) indicates that $\langle i, j \rangle \notin E$.

In a directed graph, the ith row corresponds to the *out-neighbors* of node i, so that the $\langle i, j \rangle$th entry of the matrix corresponds to the presence/absence of an edge *from i to j*. The ith column corresponds to the in-neighbors of i. See Figure 11.11.

Example 11.8: Two sample adjacency matrices.
The adjacency matrices for the graphs from Example 11.7 are shown in Figure 11.12.

The adjacency matrix has two properties that are worth a note. (Again, see Figure 11.11.) First, *the main diagonal contains all zeros:* a 1 in the $\langle i, i \rangle$th position of the matrix would correspond to an edge between node i and node i—that is, a self-loop, which is forbidden in a simple graph.

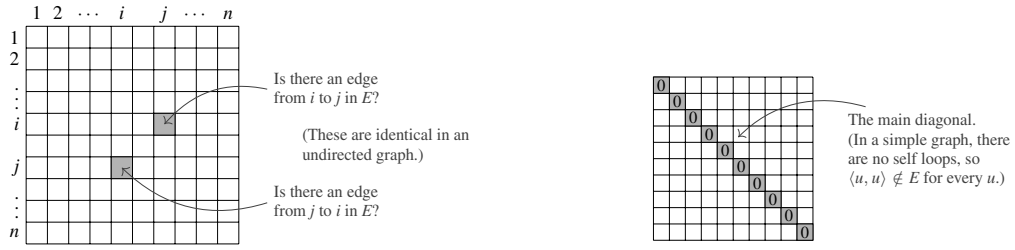


Figure 11.11 A schematic of an adjacency matrix.

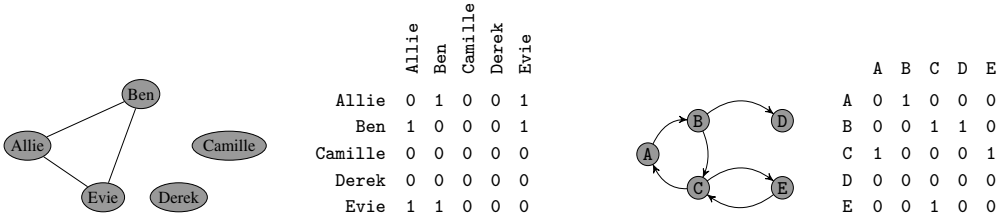


Figure 11.12 The two graphs from Figure 11.10, represented by adjacency matrices.

Second, *for an undirected graph, the matrix is symmetric:* the $\langle i, j \rangle$th position of the matrix records the presence or absence of an edge from i to j, which is identical to the presence or absence of an edge from j to i in an undirected graph. Adjacency matrices are not necessarily symmetric in directed graphs: there may be an edge from u to v without an edge from v to u.

Choosing between adjacency lists and matrices

Which of the two data structures that we've seen for graphs should we choose? Are adjacency lists better than adjacency matrices, or the other way around?

> *Meta–problem-solving tip:* The answer to "which is better?" in a class or textbook is almost always *It depends!* After all, why would we waste time/pages on a solution that's always worse!? (The only plausible answer is that it warms us up conceptually for a better but more complex solution.) The real question here *what does it depend on?*

Recall that there are two basic questions about graphs that we wish to answer quickly: (A) is v a neighbor of u?, and (B) what are all of u's neighbors? Figuring the details of how efficiently we can answer these questions with an adjacency list or an adjacency matrix is better suited to a data-structures textbook than this one, but here's a brief summary of the reasoning.

Adjacency lists: An adjacency list is perfectly tailored to answering Question (B): we've stored precisely the list of u's neighbors for each node u, so we simply iterate through that list to output u's neighborhood. To answer Question (A), we need to search through that same unsorted list to see if v is present. In both cases, we have to spend constant time finding u's list in the table, and then we examine a list of length $degree(u)$ to answer the question.

Adjacency matrices: An adjacency matrix is perfect for answering Question (A): we just look at the appropriate spot in the table. If the $\langle u, v \rangle$th entry is True, then the edge $\langle u, v \rangle$ exists. This lookup takes constant time. Answering Question (B) requires looking at one entire row of the table, entry by entry. There are $|V|$ entries in the row, so this loop requires $|V|$ operations.

Thus adjacency matrices solve Question (A) faster, while adjacency lists are faster at solving Question (B).

In addition to worrying about the time required to answer these questions, we'd also want the *space*—the amount of memory—consumed by the data structure to be as small as possible. (You can think of "the amount of memory" as the total number of boxes that appear in the diagrams in Figures 11.9 and 11.11.)

	adjacency list	adjacency matrix
running time for Question (A): is v a neighbor of u?	$1 + \Theta(degree(u))$	$\Theta(1)$
running time for Question (B): what are all of u's neighbors?	$1 + \Theta(degree(u))$	$\Theta(\|V\|)$
total space consumed by the data structure	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\|^2)$

Figure 11.13 A summary of the efficiency differences between adjacency lists and adjacency matrices. The better data structure in each row is highlighted.

Example 11.9: Space consumption for adjacency lists and matrices.
Consider a graph $G = \langle V, E \rangle$ stored using an adjacency list or an adjacency matrix. In terms of the number of nodes and the number of edges in G—that is, in terms of $|V|$ and $|E|$—how much memory is used by these data structures?

Solution. An adjacency matrix is a $|V|$-by-$|V|$ table, and thus contains exactly $|V|^2$ cells. (Of them, the $|V|$ cells on the diagonal are always 0, but they're still there!)

An adjacency list is a $|V|$-element table pointing to $|V|$ lists; the length of the list for node u is exactly $degree(u)$. Thus the total number of cells in the data structure is

$$|V| + \sum_{u \in V} degree(u).$$

In an undirected graph we have $\sum_u degree(u) = 2|E|$, by Theorem 11.8; in a directed graph we have $\sum_u out\text{-}degree(u) = |E|$ by Exercise 11.18. Thus the total amount of memory used is $|V| + 2|E|$ for an undirected graph, and $|V| + |E|$ for a directed one.

Figure 11.13 summarizes the efficiency of these data structures, using asymptotic notation from Chapter 6. (Note that, in a simple graph, we have that $degree(u) \leq |V|$ and $|E| \leq |V|^2$.) So, is an adjacency list or an adjacency matrix better? *It depends!*

First, it depends on what kind of questions we want to answer: if we will ask few "is v a neighbor of u?" questions, then adjacency lists will be faster. If we will ask many of those questions, then we probably prefer adjacency matrices. Similarly, it might depend on how much, if at all, the graph changes over time: adjacency lists are harder to update than adjacency matrices.

Second, it depends on how many edges are present in the graph. If the total number of edges in the graph is relatively small—and thus most nodes have only a few neighbors—then $degree(u)$ will generally be small, and the adjacency list will win. If the total number of edges in the graph is relatively large, then $degree(u)$ will generally be larger, and the adjacency matrix will perform better. (Many of the most interesting real-world graphs are sparse: for example, the typical degree of a person in a social network like Facebook is perhaps a few hundred or at most a few thousand—very small in relation to the billions of Facebook users.)

11.2.3 Relationships between Graphs: Isomorphism and Subgraphs

Now that we have the general definitions, we'll turn to a few more specific properties that certain graphs have. We'll start in this section with two different relationships between pairs of graphs—when two graphs are "the same" and when one is "part" of another; in Section 11.2.4, we'll look at single graphs with a particular structure.

Graph isomorphism

When two graphs G and H are identical except for how we happen to have arranged the nodes when we drew them on the page (and except for the names that we happen to have assigned to the nodes), then we call the graphs *isomorphic* (Greek: *iso* "same"; *morph* "form"). Informally, G and H are isomorphic if there's a way to relabel (and rearrange) the nodes of G so that G and H are exactly identical. More formally:

Definition 11.14: Graph isomorphism.
Consider two graphs $G = \langle V, E \rangle$ and $H = \langle U, F \rangle$. We say that G and H are *isomorphic* if there exists a bijection $f : V \to U$ such that

$$\text{for all } a \in V \text{ and } b \in V, \qquad \langle a, b \rangle \in E \Leftrightarrow \langle f(a), f(b) \rangle \in F.$$

(By abusing notation as we described earlier, this definition works for either undirected or directed graphs G and H.) Here are some small examples:

Example 11.10: Two isomorphic graphs.
Let's show that the directed graphs in Figure 11.14a and Figure 11.14b are isomorphic. To do so, define the following bijection $f : \{1, 2, \ldots, 6\} \to \{\mathtt{A}, \mathtt{B}, \ldots, \mathtt{F}\}$:

$$f(1) = \mathtt{A} \qquad f(2) = \mathtt{D} \qquad f(3) = \mathtt{C} \qquad f(4) = \mathtt{F} \qquad f(5) = \mathtt{B} \qquad f(6) = \mathtt{E}.$$

Then the edges in the two graphs are

$\langle 1,2 \rangle$	$\langle 1,3 \rangle$	$\langle 1,4 \rangle$	$\langle 1,5 \rangle$	$\langle 1,6 \rangle$	$\langle 2,4 \rangle$	$\langle 2,6 \rangle$	$\langle 3,6 \rangle$
$\langle f(1), f(2) \rangle$	$\langle f(1), f(3) \rangle$	$\langle f(1), f(4) \rangle$	$\langle f(1), f(5) \rangle$	$\langle f(1), f(6) \rangle$	$\langle f(2), f(4) \rangle$	$\langle f(2), f(6) \rangle$	$\langle f(3), f(6) \rangle$
$\langle \mathtt{A}, \mathtt{D} \rangle$	$\langle \mathtt{A}, \mathtt{C} \rangle$	$\langle \mathtt{A}, \mathtt{F} \rangle$	$\langle \mathtt{A}, \mathtt{B} \rangle$	$\langle \mathtt{A}, \mathtt{E} \rangle$	$\langle \mathtt{D}, \mathtt{F} \rangle$	$\langle \mathtt{D}, \mathtt{E} \rangle$	$\langle \mathtt{C}, \mathtt{E} \rangle$

Because these lists of edges match exactly, the two graphs are isomorphic.

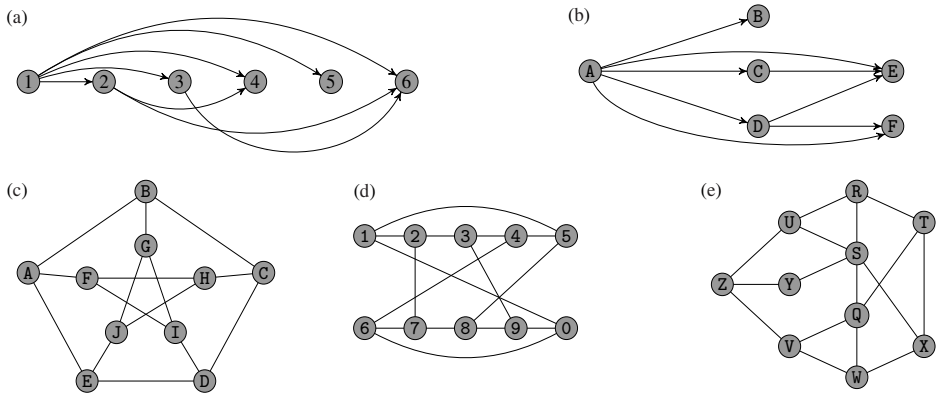


Figure 11.14 Several graphs. (Incidentally, the edges in (a) can be written as $\{\langle a, b \rangle : a < b \text{ and } a \text{ evenly divides } b\}$.)

Example 11.11: Isomorphic graphs.
Which pairs, if any, of the graphs in Figure 11.14c, Figure 11.14d, and Figure 11.14e are isomorphic?

Solution. The graphs in Figure 11.14c and Figure 11.14d are isomorphic. The easiest way to see this fact is to show the mapping between the nodes of the two graphs:

A	B	C	D	E	F	G	H	I	J
1	2	3	4	5	0	7	9	6	8

It's easy to verify that all 15 edges now match up between the first two graphs. But the third graph is not isomorphic to either of the others. The easiest justification is that node S in the third graph has degree 5, and no node in either of the first two graphs has degree 5. No matter how we reshuffle the nodes of graph #3, there will still be a node of degree 5—so the third graph can never match the others.

Problem-solving tip: When you're trying to prove or disprove a claim about graphs, you may find it useful to test out the claim against the following four "trivial" graphs: a graph with a single node (•); a graph with two nodes and no edges (• •); a graph with two nodes joined by an edge (•–•); and graph with two pairs of nodes with each pair joined by an edge (•–• •–•). A lot of bogus claims about graphs turn out to be false on one of these four examples—or, unexpectedly, the so-called *Petersen graph,* the graph in Figure 11.14c. (The Petersen graph is named after Julius Petersen (1839–1910), a Danish mathematician.) It's a good idea to try out any conjecture on all five of these graphs before you let yourself start to believe it!

Taking it further: In general, it's easy to test whether two graphs are isomorphic by brute force (try all permutations!), but no substantially better algorithms are known. The computational complexity of the graph isomorphism problem has been studied extensively over the last few decades, and there has been substantial progress—but no complete resolution. It's easy to convince someone that two graphs G and H are isomorphic: we can simply describe the relabeling of the nodes of G so that the resulting graphs are identical. (The "convincee" then just needs to verify that the edges really do match up.) When G and H are not isomorphic, it *might* be easy to demonstrate their nonisomorphism: for example, if they have a different number of nodes or edges, or if the degrees in G aren't identical to the degrees in H. But the graphs may have identical degree distributions and yet *not* be isomorphic; see Exercise 11.49.

Subgraphs

When a graph H is isomorphic to a graph G, we can think of having created H by moving around some of the nodes and edges of G. When H is a *subgraph* of G, we can think of having created H by deleting some of the nodes and edges of G. (Of course, it doesn't make sense to delete either endpoint of an edge e without also deleting the edge e.) Here's the definition, for either undirected or directed graphs:

Definition 11.15: Subgraph.
Let $G = \langle V, E \rangle$ be a graph. A *subgraph* of G is a graph $G' = \langle V', E' \rangle$ where $V' \subseteq V$ and $E' \subseteq E$ such that every edge $\langle u, v \rangle \in E'$ satisfies $u \in V'$ and $v \in V'$.

(Definition 11.15 uses the abuse of notation that we mentioned earlier: we "ought" to have written $\{u, v\} \in E'$ for the case that G is undirected.)

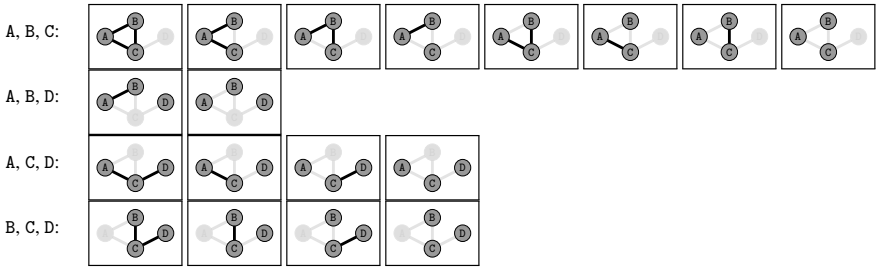


Figure 11.15 All 3-node subgraphs of the graph with four nodes (A, B, C, and D) and four edges (three edges joining C to every other node, plus an edge between A and B).

Example 11.12: All 3-node subgraphs of G.
Consider the graph G with nodes $\{\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{D}\}$ and edges $\{\{\mathtt{A},\mathtt{B}\},\{\mathtt{A},\mathtt{C}\},\{\mathtt{B},\mathtt{C}\},\{\mathtt{C},\mathtt{D}\}\}$. Then the graph G' with nodes $\{\mathtt{B},\mathtt{C},\mathtt{D}\}$ and edges $\{\{\mathtt{B},\mathtt{C}\},\{\mathtt{C},\mathtt{D}\}\}$ is a subgraph of G. In fact, G has *many* different subgraphs. Figure 11.15 shows all of the 3-node subgraphs of G. (There are many other subgraphs—about 50 total—when we consider subgraphs with 1, 2, 3, or 4 nodes.)

Taking it further: One of the earliest applications of a formal, mathematical perspective to networks—a collaboration between a psychologist and mathematician, in the 1950s—was based on subgraphs. Consider a *signed social network,* an undirected graph where each edge is labeled with '+' to indicate friends, or '−' to indicate enemies. (See Figure 11.16a.) The adages "the enemy of my enemy of my friend" and "the friend of my friend is my friend" correspond to the claim that the subgraphs in Figure 11.16b would not appear. Dorwin Cartwright (1915–2008, the psychologist) and Frank Harary (1921–2005, the mathematician) proved some very interesting structural properties of any signed social network G that does not have either triangle in Figure 11.16b as a subgraph—a property that they called "structural balance"—and in the process helped launch much of the mathematical and computational work on graphs that's followed. For more, see [27].

We sometimes refer to a special kind of subgraph: the subgraph of $G = \langle V, E\rangle$ *induced* by a set $V' \subseteq V$ of nodes is the subgraph of G where every edge between nodes in V' is retained. The first subgraph in each row of Example 11.12 is the induced subgraph for its nodes.

Here's a brief description of one application of (induced) subgraphs:

Example 11.13: Motifs in biological networks.
At any particular moment in any particular cell, some of the genes in the organism's DNA are being *expressed*—that is, some genes are "turned on" and the proteins that they code for are being produced by the cell. Furthermore, one gene g can *regulate* another gene g': when g is being expressed, gene g

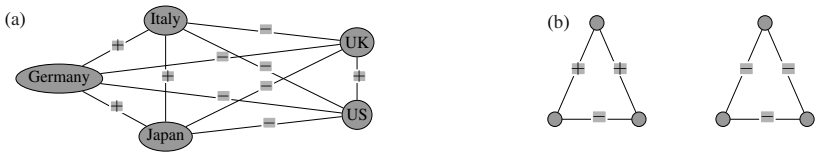


Figure 11.16 Signed social networks: (a) a signed network from 1941, and (b) two forbidden triangles.

can cause the expression of gene g' to increase or decrease over the baseline level. A great deal of recent biological research has allowed us to construct *gene-regulation networks* for different such settings: that is, a directed graph G whose nodes are genes, and whose edges represent the regulation of one gene by another.

Consider the induced subgraph of a particular set of genes in such a graph G—that is, the interactions among the particular genes in that set. Certain patterns of these subgraphs, called *motifs,* occur significantly more frequently in gene-regulation networks than would be expected by chance. Biologists generally believe that these repeated patterns indicate something important in the way that our genes work, so computational biologists have been working hard to build efficient algorithms to identify induced subgraphs that are overrepresented in a network.

11.2.4 Special Types of Graphs: Complete, Bipartite, Regular, and Planar Graphs

In Section 11.2.3, we looked at two ways in which a pair of graphs might be related. Here, we'll consider special characteristics that a single graph might have—that is, subcategories of graphs with some particular structural properties. These special types of graphs arise frequently in various applications.

Complete graphs

Our first special type of graph is a *complete graph* (also called a *clique*), which is an undirected graph in which every possible edge exists:

Definition 11.16: Complete graph/clique.
A *complete graph* or *clique* is an undirected graph $G = \langle V, E \rangle$ such that $\{u, v\} \in E$ for any two distinct nodes $u \in V$ and $v \in V$.

See Figure 11.17 for examples of complete graphs of varying sizes. (In everyday usage, a *clique* is a small, tight-knit, and exclusionary group of friends that doesn't mingle with outsiders—like the cool kids who never would even say hello to me when I was in middle school. If you think about a graph as a social network, the common-language meaning is similar to Definition 11.16. In computer science, though, the word *clique* usually rhymes with *bleak* or *sleek*. In common-language usage, the word usually rhymes with *slick* or *flick*.)

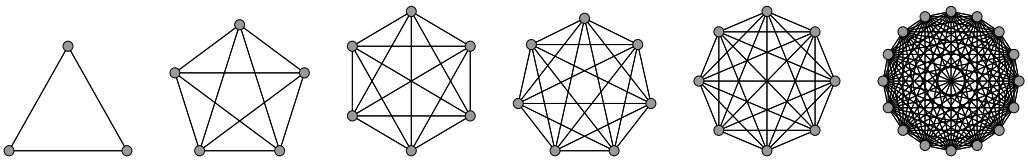


Figure 11.17 Complete graphs with 3, 5, 6, 7, 8, and 16 nodes.

Observe that an undirected graph with n nodes has $\binom{n}{2}$ unordered pairs of nodes, and therefore an n-node complete graph has $\binom{n}{2} = n(n-1)/2$ edges.

A complete graph with n nodes is sometimes denoted by $\mathcal{K}_n$. (Why? There are two different prevailing explanations for this notation, so there's some debate. One is that the K is short for *complete*—or, rather, short for *komplett*; the notation was invented by a German speaker. The second is that the K is in honor of Kazimierz Kuratowski (1896–1980), a Polish mathematician who made major contributions to the study of graphs, among other mathematical topics.)

The word *clique* can also refer to a *subgraph* that's complete—that is, a subgraph in which every possible edge actually exists. So we'd say that the graph with nodes $\{\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{D}\}$ and edges $\{\{\mathtt{A},\mathtt{B}\},\{\mathtt{A},\mathtt{C}\},\{\mathtt{B},\mathtt{C}\},\{\mathtt{C},\mathtt{D}\}\}$ contains a 3-node clique $\{\mathtt{A},\mathtt{B},\mathtt{C}\}$.

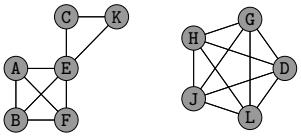
Here's one small example of an interesting application in which cliques arise:

Example 11.14: Collaboration networks and cliques.
Imagine a setting in which different groups of people can work together in different teams, with each person allowed to participate in multiple teams. For example:

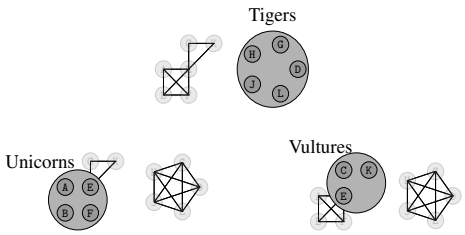
- actors in movies. (A "team" is the cast of a single movie.)
- scientific researchers. (A "team" is the set of coauthors of a published paper.)
- employees of a company. (A "team" is a group that worked on a specific project.)

A *collaboration network* is a graph G that represents a setting like these: the nodes of G are the people involved; there is an edge between any two people who have worked together on at least one team. (You may have heard of a challenge in the collaboration network: in the *Kevin Bacon Game,* you're given the name of some actor A; your job is to find a sequence of edges that connects A to the "Kevin Bacon" node in the movie collaboration network. There's a similar game that computer scientists play in the scientific collaboration network, trying to connect themselves to the Hungarian polymath Paul Erdős. See p. 4-46.)

Team	*Members*
Tigers	Deborah, George, Hicham, Josh, Lauren
Unicorns	Anita, Bev, Eva, Fernan
Vultures	Cathy, Eva, Kelly



(a) Three teams, and the resulting collaboration graph.



(b) The clique resulting from each team in the collaboration graph. (Here, $\mathcal{K}_5$, $\mathcal{K}_4$, and $\mathcal{K}_3$.)

Figure 11.18 A collaboration network.

For a concrete example, see Figure 11.18. Notice that each team results in a clique in the collaboration graph—every pair of members of that team is joined by an edge—as shown in Figure 11.18b.

Bipartite graphs

Our second special kind of graph is a *bipartite* graph (Latin: *bi* "two"; *part* "part"). In a bipartite graph, the nodes can be divided into two groups such that no edges join two nodes that are in the same group: that is, there are two "kinds" of nodes, and all edges join a node of Type A to a node of Type B. Formally:

Definition 11.17: Bipartite graph.
A *bipartite graph* is an undirected graph $G = \langle V, E \rangle$ such that V can be partitioned into two disjoint sets L and R where, for every edge $e \in E$, one endpoint of e is in L and the other endpoint of e is in R.

For example, consider a graph with nodes `A`, `B`, `C`, `D`, `E`, and `F`, and with edges $\{\texttt{A}, \texttt{B}\}$, $\{\texttt{A}, \texttt{C}\}$, $\{\texttt{C}, \texttt{E}\}$, and $\{\texttt{D}, \texttt{E}\}$. This graph is bipartite: for example, we can split the nodes into two groups—the vowels $\{\texttt{A}, \texttt{E}\}$ and the consonants $\{\texttt{B}, \texttt{C}, \texttt{D}, \texttt{F}\}$—such that every edge joins a vowel and a consonant. (There's another split that would also have worked: $\{\texttt{A}, \texttt{E}, \texttt{F}\}$ and $\{\texttt{B}, \texttt{C}, \texttt{D}\}$.) See Figure 11.19 for a visualization of the vowel–consonant split.

Bipartite graphs are traditionally drawn with the nodes arranged in two columns, one for each part: *left* ("*L*") and *right* ("*R*"). But notice that the definition only requires that it be *possible* to divide the nodes into two groups, with no within-group edges; it doesn't require that there be two columns in whatever drawing of the graph you're looking at.

Example 11.15: Bipartite and nonbipartite graphs.
Which of the graphs in Figure 11.20 are bipartite?

Solution. All of them except (iii)! Although (iv) and (v) are the only graphs drawn in the "two-column" format, both (i) and (ii) can be rearranged into two columns. In fact, aside from node positioning, graphs (i) and (iv) are identical. And, similarly, graphs (ii) and (v) are isomorphic! Only (iii) is not bipartite: if we attempt to put the topmost node in one group, then both of the next higher two nodes must both be in the other group—but they're joined by an edge themselves, and so we're stuck.

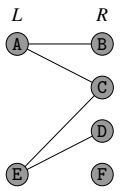


Figure 11.19 A bipartite graph.

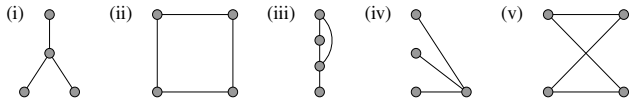


Figure 11.20 Five possibly bipartite graphs.

Many interesting real-world phenomena can be modeled using bipartite graphs:

Example 11.16: Bipartite graphs as models.
Here are just a few of the scenarios that are naturally modeled using bipartite graphs:

- dating relationships in a strictly binary, heterosexual community: the nodes are the boys B and the girls G; every edge connects some boy to some girl.
- nodes are courses and students; an edge joins a student to each class they've taken.
- *affiliation networks*: people and organizations are the nodes; an edge connects person p and organization o if p is a member of o.

There's one further refinement of bipartite graphs that we'll mention: a *complete bipartite graph* is a bipartite graph in which every possible edge exists. In other words, a complete bipartite graph has the form $G = \langle L \cup R, E \rangle$ where, for every node $\ell \in L$ and $r \in R$, we have $\{\ell, r\} \in E$. A complete bipartite graph with ℓ nodes in the left group and r nodes in the right group is sometimes denoted by $\mathcal{K}_{\ell,r}$. See Figure 11.21 for a few examples. (Note again that, as with the $\mathcal{K}_{2,4}$ in Figure 11.21, we don't have to draw a bipartite graph in two-column format—if it's bipartite, then it's still bipartite no matter how we draw it!)

Regular graphs

Our next type of graph is defined in terms of the degree of its nodes: a *regular graph* is one in which all of the nodes have an identical number of neighbors.

Definition 11.18: Regular graph.
Let $d \geq 0$ be an integer. A *d-regular graph* is a graph G such that every node has degree precisely equal to d. If G is d-regular for any d, then we say that G is a *regular graph.*

(Most of the time one talks about regular graphs that are undirected, but we can speak of regular directed graphs, too; we'd generally require that all in-degrees match each other *and* all out-degrees match each other.) Several examples of regular graphs are shown in Figure 11.22.

There are many real-world examples in which regular graphs are useful: for example, imagine constructing a physical network of computers in which each machine only has the capacity for a fixed number of connections. Here are two other useful applications of regular graphs:

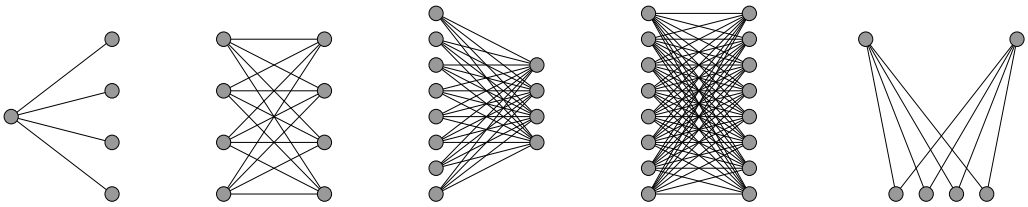
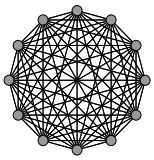


Figure 11.21 Complete bipartite graphs of varying sizes: $\mathcal{K}_{1,4}$, $\mathcal{K}_{4,4}$, $\mathcal{K}_{8,4}$, $\mathcal{K}_{8,8}$, and $\mathcal{K}_{2,4}$.

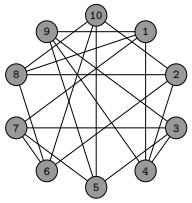
$V = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}\}$

$E = \{\ \{\mathtt{A}, \mathtt{B}\}, \{\mathtt{A}, \mathtt{E}\}, \{\mathtt{B}, \mathtt{C}\},$
$\{\mathtt{C}, \mathtt{F}\}, \{\mathtt{D}, \mathtt{E}\}, \{\mathtt{D}, \mathtt{F}\}\ \}$

(a) A 2-regular graph. (You can check: each node has degree two.)



(b) For any n, the complete graph $\mathcal{K}_n$ is $(n-1)$-regular.



(c) A 4-regular 10-node graph.

Figure 11.22 Several examples of regular graphs.

Example 11.17: Scheduling sports with a regular graph.
You are the League Commissioner for an intramural ultimate frisbee league. There are 10 teams in the league, each of whom should play four games. No two teams should play each other twice. Suppose that you construct an undirected graph $G = \langle V, E \rangle$, where $V = \{1, 2, \ldots, 10\}$ is the set of teams, and E is the set of games to be played. If G is an 4-regular graph, then all of the listed requirements are met. Figure 11.22c is a randomly generated example of such a graph; you could use that graph to set the league schedule.

A 1-regular graph is called a *perfect matching,* because each node is "matched" with one—and only one—neighbor. (If every node has degree *at most* 1, then the graph is just called a *matching.*) Matchings have a variety of applications—for example, see p. 9-75 for their role in the Enigma machine—but here's another specific use of matchings, in assigning partnerships:

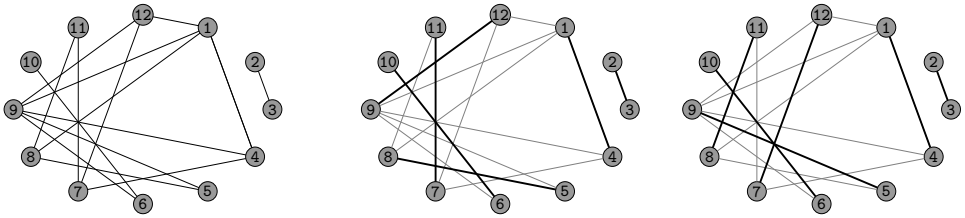
Example 11.18: Matchings for CS partnerships.
Each of n students in an Intro CS class submits a list of people whom they'd like to have as a partner for the final project. Define the following undirected graph G:

- the set V of nodes is $\{1, 2, \ldots, n\}$, one per student.
- the set E of edges includes $\{u, v\}$ if *both* of the following are true: student u wants to work with student v, *and* student v wants to work with student u.

The instructor can assign partnerships by finding a 1-regular graph $G' = \langle V, E' \rangle$ with $E' \subseteq E$—that is, a subgraph of G that includes all of the nodes of G. See Figure 11.23 for an example. (Incidentally, Example 9.34 asked: how many perfect matchings are there in $\mathcal{K}_n$?)

Planar graphs

Our last special type of graph is a *planar graph*, which is one that can be drawn on a sheet of paper without any lines crossing:



(a) A graph G of compatabilities. (b) For G, these graphs (among others) are valid partner assignments.

Figure 11.23 Choosing partnerships in a class. (See Example 11.18.)

Definition 11.19: Planar graph.
A *planar graph* is a graph G such that it is possible to draw G on a plane (that is, on a piece of paper) such that no edges cross.

It's important to note that a graph is planar if it is *possible* to draw it with no crossing edges; just because a graph is drawn with edges crossing does not mean that it isn't planar. Here is an example of a planar graph:

Example 11.19: New England, in a plane.
Figure 11.24 shows two copies of the same graph—one drawn with edge crossings, and another with the nodes rearranged to avoid any edges crossing—representing the U.S. states in the New England region.

Figure 11.24 corresponds to one of the most famous types of planar graph, one derived from a map: we can think of the countries on a map as nodes, and we draw an edge between two country–nodes if those two countries share a border. (See p. 4-48 for a discussion of the *four-color theorem* for maps, which we could have phrased as a result about planar graphs instead.)

There are other applications of planar graphs in computer science, too. For example, we can view a *circuit* (see Section 3.3.3) as a graph, where the logic gates correspond to nodes and the wires correspond to edges. Most modern circuits are now *printed* on a board (where the "ink" is the conducting material that serves as the wire), and the question of whether a particular circuit can be printed on a single layer is precisely the question of whether its corresponding graph is planar. (If it's not planar, we'd like to minimize the number of edges that cross, or more specifically the number of layers we'd need in the circuit.)

Here's one more set of planarity challenges for you to try:

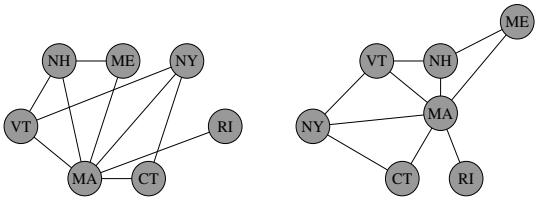


Figure 11.24 New England, drawn in two different ways.

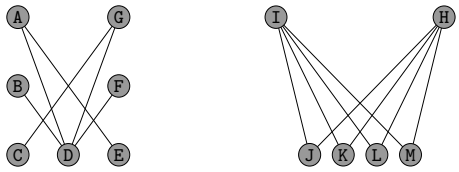


Figure 11.25 Two (possibly planar) graphs.

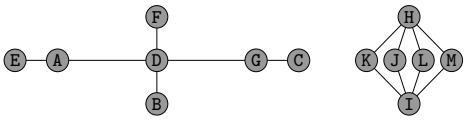


Figure 11.26 Rearrangements of the graphs in Figure 11.25 to show that they are planar.

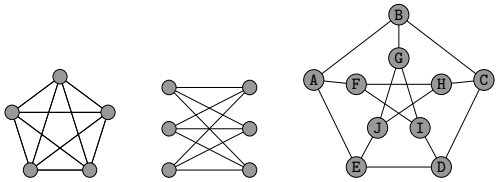


Figure 11.27 Three non-planar graphs: $\mathcal{K}_5$, $\mathcal{K}_{3,3}$, and the Petersen graph.

Example 11.20: Two planar challenges.

Consider the two graphs in Figure 11.25. Are these graphs planar?

Solution. Yes, both: we can rearrange the nodes so that there are no edges that cross, as shown in Figure 11.26.

Taking it further: Determining how to lay out a planar graph without edge crossings can be an interesting amusement—see `www.planarity.net` for a surprisingly fun game based on planar graphs. So far we haven't seen any examples of graphs that *can't* be rearranged so that no edges cross. But, if you play around long enough, you should be able to convince yourself that neither $\mathcal{K}_5$ and $\mathcal{K}_{3,3}$ are planar; see Figure 11.27. And, while this shouldn't be at all obvious, it turns out that $\mathcal{K}_5$ and $\mathcal{K}_{3,3}$ are in a sense the only "reasons" that a graph can be nonplanar. A theorem known as *Kuratowski's Theorem*—after the Polish mathematician who may have lent his initial to the notation for complete graphs—says that every graph is planar unless it "contains" $\mathcal{K}_5$ or $\mathcal{K}_{3,3}$ for a subgraph-like notion of "containment." (It's not exactly the subgraph relation, because there are graphs that do not contain $\mathcal{K}_5$ or $\mathcal{K}_{3,3}$ as subgraphs but nonetheless are nonplanar in some sense "because" of one of them. For example, the Petersen Graph from Example 11.11—see Figure 11.27—is nonplanar, but it doesn't have $\mathcal{K}_5$ as a subgraph. But if we "collapse" together the nodes `A/F`, `B/G`, `C/H`, `D/I`, and `E/J` into "supernodes" then the resulting graph *is* $\mathcal{K}_5$.)

Computer Science Connections

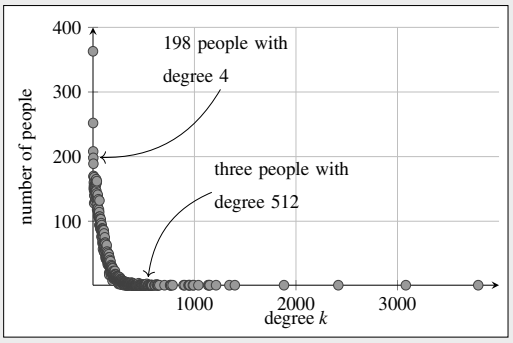
Degree Distributions and the Heavy Tail

When we think about massive graphs like the web (with nodes representing web pages and edges representing links from one page to another) or an online social network (with nodes representing people and edges representing whatever it is that being "friends" in an online social network means), it is interesting to look at how properties of individual nodes are distributed across the population. We can look at the distribution of any node-by-node property—the physical height of Twitter users, or the number of words of text per web page, for example. But in addition to demographic properties like height and length, we can also look at the distribution of network-type properties.

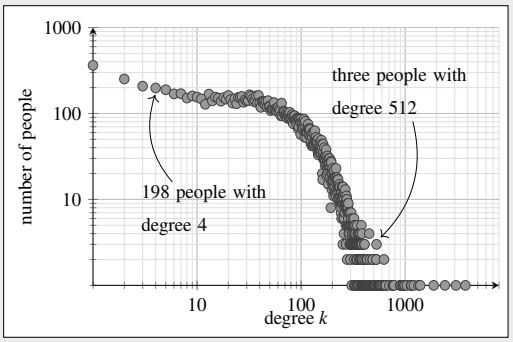
The *degree distribution* of a graph G shows, for each possible degree d, the number of nodes in G whose degree is d. While one might initially expect the degree distribution of a real social network to look similar to the distribution of heights, it turns out that the degree distribution has very different properties.

Figure 11.28 shows the degree distribution (in linear, log–log, and cumulative form) for members of the University of North Carolina: that is, for each value of k, the number of people in the network who have precisely k Facebook friends in the network. About 350 people have only 1 friend, which is the most common number of friends to have. There are about 750,000 friendships represented in this dataset; the *average degree* is ≈ 84.

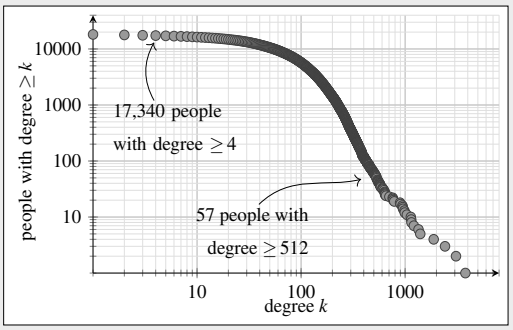
But, looking at the far-right end of Figure 11.28a and 11.28b, we see a handful of people with very high degrees: 2000, 2500, 3000, and even ≈ 3800. One of the interesting facts about degree distributions in real social networks (or the web) is that there are people whose popularity is massively larger than average: the highest-degree person in this dataset is about $3800/84 \approx 45$ times more popular than average. (Imagine the tallest person at the University of North Carolina being 45 times taller than average!)



(a) The degree distribution.



(b) Another plot of the degree distribution: the data as in (a), but with the axes scaled logarithmically instead of linearly.



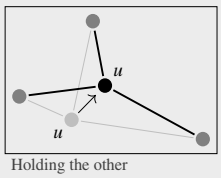
(c) The cumulative degree distribution: the number of people with degree $\geq k$ (whereas (a) and (b) showed the number of people with degree $= k$).

Figure 11.28 The degree distribution of $\approx 18{,}000$ Facebook users at the University of North Carolina. (The data are from the Facebook5 dataset, from Mason Porter via the International Network for Social Network Analysis [125].)

Significant research by computer scientists (among researchers from many other fields) interested in the structure of social networks and the world-wide web has focused on this so-called *heavy-tailed degree distribution*. Some of the literature debates the particular form of this distribution; for example, whether the distribution has the particular form of a *power law,* where the number of people with degree k is roughly k^{α} for some small constant α, usually around 2. (If you're interested, you can read a lot more about power laws and heavy-tailed degree distributions in [43].)

Computer Science Connections

Graph Drawing, Graph Layouts, and the 9/11 Memorial



Holding the other three nodes' positions constant, node u would move to the new position that's shown.

Figure 11.29 Part of the New York City subway system, where each node's position corresponds to the station's spatial location. Inset: a node "seeks" a position that perfectly balances the attractive forces to each of its neighbors (and repelling forces from all other nodes).

Visual representations of most large graphs are too cluttered for a human viewer to process: there are just too many nodes and edges crammed into a small space to see much of anything. Visually presenting a graph like Facebook (billions of nodes, tens of billions of edges) without it looking like a grade-school scribble is daunting. But there is an entire subfield of computer science called *graph drawing*, which is devoted to taking networks and producing good—clear, aesthetic, informative—images of networks.

In some graphs, each node has a "natural location" and it is clear where on the page it should be placed. For example, graphs may represent nodes that have a precise location situated in the physical world. With that kind of information for each node, presenting the graph well is easier. (See Figure 11.29.) But many large graphs do not have obvious coordinates associated with each node: while you and your classmates may have geographic locations (residence halls), it's not clear that your room really best describes "where" you fit in the social scene of your institution.

For graphs whose nodes don't have obvious coordinates, we have to do something else. One common approach is to arrange the nodes based on a physics analogy, as follows. Imagine each node as a charged particle: any two nodes that are joined by an edge are pulled together by an attractive force, and any two nodes that are not joined by an edge are pushed apart by a repulsive force. Then figuring out how to place nodes on the page can be done by starting them in a random configuration and letting the attractive/repulsive forces move the nodes around until they're "happy" in their current positions. (See the inset of Figure 11.29.)

An idea like this one was actually used in designing the 9/11 memorial at the site of the World Trade Center. The memorial was designed with bronze panels inscribed with the 2982 names of victims. A team of computer scientists, architects, and visual artists collaborated to organize the names in a meaningful way. Families were invited to submit "meaningful adjacencies" between victims—which would cause two names to be as close together in the bronze panels as possible. (One of the other algorithmic issues regarding the layout of this memorial was that the designers wanted the names to be placed at evenly spaced intervals on the bronze panels; this constraint added to the computational complexity of the process.) The team used an algorithm to organize the names in an arrangement that respected these requests, which was then used in the final design of the memorial. (In addition to the broader news reports on the wrenching emotional and historical aspects of 9/11 Memorial, the algorithmic aspects of the memorial were also covered in the popular press; for more, see [98].)

EXERCISES

Draw a graph with the following nodes and edges. Does it make sense to use a directed or undirected graph? Is the graph you've drawn simple?

11.1 The nodes are $V = \{1, 2, \ldots, 10\}$; an edge connects x and y if $gcd(x, y) = 1$.

11.2 The nodes are $V = \{1, 2, \ldots, 10\}$; an edge connects x and y if x divides y.

11.3 The nodes are $V = \{1, 2, \ldots, 10\}$; an edge connects x and y if $x < y$.

11.4 List the edges of the graph in Figure 11.30a, and identify the node(s) with the highest degree.

11.5 Repeat for the graph in Figure 11.30b.

11.6 Repeat for the graph in Figure 11.30c, identifying both the highest in-degree node(s) and the highest out-degree node(s).

11.7 Repeat for the graph in Figure 11.30d, again for both in- and out-degree.

11.8 If $G = \langle V, E \rangle$ is an *undirected,* simple graph with n nodes, what's the largest that $|E|$ can be (in terms of n)? The smallest? Explain.

11.9 If G is a *directed,* simple graph with n nodes, what's the largest that $|E|$ can be? The smallest?

11.10 How do your answers to Exercise 11.9 change if self-loops are allowed?

11.11 How do your answers to Exercise 11.9 change if self-loops and parallel edges are allowed?

The anthropologist Robin Dunbar has argued that humans have a mental capacity for only approximately 150 *friends [40]. Dunbar's argument is based in part on the physical size of the human brain, and cross-species comparisons;* 150 *is now occasionally known as* Dunbar's Number. *(Thanks to Michael Kearns, from whom I learned a somewhat related version of these exercises.)*

11.12 Suppose that Alice has exactly 150 friends, and each of her friends has exactly 150 friends—that is, a friend of Alice knows Alice and 149 other people. (Note that Alice's friends' sets of friends can overlap.) Let S denote the set of people that Alice knows directly or with whom Alice has a mutual friend. What's the largest possible value of $|S|$?

11.13 For the set S defined as in Exercise 11.12, what's the *smallest* possible value of $|S|$?

11.14 Continue to assume that everyone has precisely 150 friends. Let S_k denote the set of all people that Bob knows via a chain of k or fewer intermediate friends:

$$\begin{aligned} S_0 &= \text{Bob's friends;} \\ S_1 &= \text{the people in } S_0 \text{ and the friends of people in } S_0; \\ S_2 &= \text{the people in } S_1 \text{ and the friends of people in } S_1\text{; and so forth.} \end{aligned}$$

In terms of k, what's the largest possible value of $|S_k|$?

11.15 Let $k \geq 0$ be arbitrary. For the set S_k as defined in Exercise 11.14, what's the *smallest* possible value of $|S_k|$?

11.16 Let u be a node in an undirected graph G. Prove that u's degree is at most the sum of the degrees of u's neighbors.

11.17 Prove Corollary 11.9: in an undirected graph $G = \langle V, E \rangle$, let n_{odd} denote the number of nodes whose degree is odd. Prove that n_{odd} is an even number. That is: prove that

$$|\{u \in V : degree(u) \bmod 2 = 1\}| \bmod 2 = 0.$$

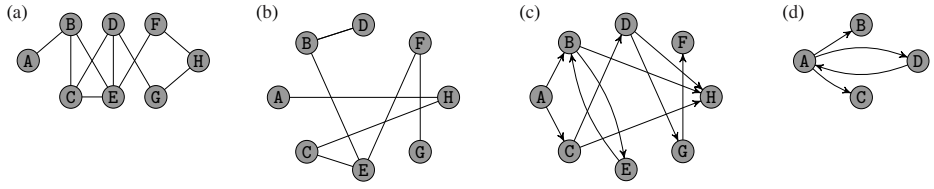


Figure 11.30 A few graphs.

11.18 Prove the analogy of Theorem 11.8 for directed graphs: for a directed graph $G = \langle V, E \rangle$,

$$\sum_{u \in V} \text{in-degree}(v) = \sum_{u \in V} \text{out-degree}(v) = |E|.$$

11.19 A *linked list* is a data structure consisting of a collection of *nodes*, as illustrated in Figure 11.31a. Define a directed graph $G = \langle V, E \rangle$, where V is the set of all nodes reachable by following any number of `next` pointers starting at the head node, and $\langle u, v \rangle \in E$ if u's `next` field points to u. (G may not be simple.) Observe that each node u in G has out-degree $d \in \{0, 1\}$. Describe a 5-node linked list in which every node of G has in-degree $d = 1$.

11.20 Describe a 5-node linked list for which the graph G (as defined in Exercise 11.19) contains a node with in-degree $d = 2$.

11.21 Describe a 5-node linked list for which the graph G (as defined in Exercise 11.19) is not simple.

11.22 *(This exercise is a tougher algorithmic challenge.)* You are given access to the head node h of an n-node linked list. The value of n is unknown to you. The only operations permitted are (a) to save a node; (b) test whether two saved nodes are the same or different; and (c) given a node u, fetch the node pointed to by u's `next` field. Give an algorithm to determine whether the given list is circular *using only a constant amount of memory*—that is, remembering only a constant number of nodes at a time.

11.23 A *doubly linked list* has n nodes with data and *two* pointers, `previous` and `next`. (See Figure 11.31b.) Consider a doubly linked list with nodes $\{1, 2, \ldots, n\}$, where each node u's `next` field points to node $u + 1$ (and node n points back to node 1). That is, u's `next` node is $v = (u \bmod n) + 1$ (and v's `previous` node is u). (Figure 11.31b shows an example with $n = 4$.) Define the directed graph $G_n = \langle V, E \rangle$, where $V = \{1, 2, \ldots, n\}$ is the set of nodes, and every node has two edges leaving it: one edge $\langle u, u.\texttt{next} \rangle$, and one edge $\langle u, u.\texttt{previous} \rangle$. Draw the graph G_5.

11.24 Give an example of a value of n for which G_n (as defined in Exercise 11.23) contains a self-loop.

11.25 Give an example of a value of n for which G_n (as defined in Exercise 11.23) contains parallel edges.

11.26 Write down an adjacency list representing the graph in Figure 11.30a.

11.27 Write down an adjacency list representing the graph in Figure 11.30b.

11.28 Write down an adjacency list representing the graph in Figure 11.30c.

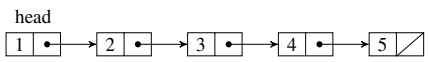
11.29 Write down an adjacency list representing the graph in Figure 11.30d.

11.30 Write down an adjacency matrix for the graph in Figure 11.30a.

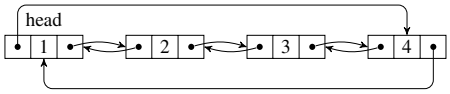
11.31 Write down an adjacency matrix for the graph in Figure 11.30b.

11.32 Write down an adjacency matrix for the graph in Figure 11.30c.

11.33 Write down an adjacency matrix for the graph in Figure 11.30d.



(a) A linked list. Each node has two fields: `data` on the left and `next` on the right.



(b) A doubly linked list. The three fields in each node, from left to right, are: `previous`, `data`, and `next`. (This doubly linked list is circular.)

Figure 11.31 Singly and doubly linked lists. Each rectangle is a node, and contains a `data` field (whatever the node stores) and a `next` field that is either `null` or points to a node in the linked list. A particular node is designated as the *head node*. Note that a *circular linked list* in which a node's `next` field points back to a previously encountered node meets this definition. In a doubly linked list, there is also a `previous` field that is either `null` or points to a node in the linked list.

11.34 Suppose that a (possibly directed or undirected) simple graph G is represented by an adjacency list. Suppose further that, for every node u in G, the list of (out-)neighbors of u has a unique length. (That is, no two nodes have the same number of (out-)neighbors.) *True or False: G must be a directed graph.* Justify your answer.

11.35 Describe a directed graph G meeting the specifications of Exercise 11.34.

11.36 The *density* of a graph $G = \langle V, E \rangle$ is the fraction of all possible edges that actually exist: that is,

$$density = \frac{|E|}{\text{[your answer to the first part of Exercise 11.8/Exercise 11.9]}}.$$

As a function of n, what is the density of an n-node path? (Like this one, for $n = 12$: .)

11.37 As a function of n, what is the density of an n-node cycle? (Like this one, for $n = 12$: .)

11.38 As a function of n, what is the density of a graph that consists of $\frac{n}{3}$ disconnected triangles? Assume that $n \bmod 3 = 3$. (Like this one, for $n = 12$: .)

11.39 As a function of n, what is the density of a graph that consists of 3 separate cliques, each of which contains exactly $\frac{n}{3}$ nodes? Assume that $n \bmod 3 = 3$. (Like this one, for $n = 12$: .)

11.40 A *hypercube* H_n is a graph in which the 2^n different nodes are all elements of $\{0, 1\}^n$. There is an edge between x and y if they differ in only one bit position. (Using the language of Section 4.2, there's an edge between any two nodes whose Hamming distance is 1.) Draw H_3.

11.41 Write down an adjacency list for the hypercube H_4 (see Exercise 11.40).

11.42 Write down an adjacency matrix for the hypercube H_4 (see Exercise 11.40).

11.43 For the hypercube H_n (see Exercise 11.40): in terms of n, how many edges does H_n have? What is its density?

11.44 Are the pair of graphs in Figure 11.32a isomorphic? Prove your answer.

11.45 Are the pair of graphs in Figure 11.32b isomorphic? Prove your answer.

11.46 Are the following graphs isomorphic? Prove your answer.

$G_1 = \langle V_1, E_1 \rangle$, where $V_1 = \{10, 11, 12, 13, 14, 15\}$ and $\langle x, y \rangle \in E_1$ if and only if x and y are not relatively prime
$G_2 = \langle V_2, E_2 \rangle$, where $V_2 = \{20, 21, 22, 23, 24, 25\}$ and $\langle x, y \rangle \in E_2$ if and only if x and y are not relatively prime.

11.47 Prove or disprove: all 5-node graphs with degrees 1, 1, 1, 1, and 0 are isomorphic.

11.48 Prove or disprove: all 5-node graphs with degrees 4, 4, 4, 3, and 3 are isomorphic.

11.49 Prove or disprove: all 5-node graphs with degrees 3, 3, 2, 2, and 2 are isomorphic.

11.50 Prove or disprove: all n-node, 3-regular graphs are isomorphic.

11.51 The computational problem of finding the largest clique (complete graph) that's a subgraph of a given graph G is believed to be very difficult. But for small graphs it's possible to do, even by brute force. Identify the size of the largest clique that's a subgraph of the graph in Figure 11.33a.

11.52 What's the size of the largest clique that's a subgraph of the graph in Figure 11.33b?

11.53 What's the size of the largest clique that's a subgraph of the graph in Figure 11.33c?

Figure 11.32 Two pairs of graphs.

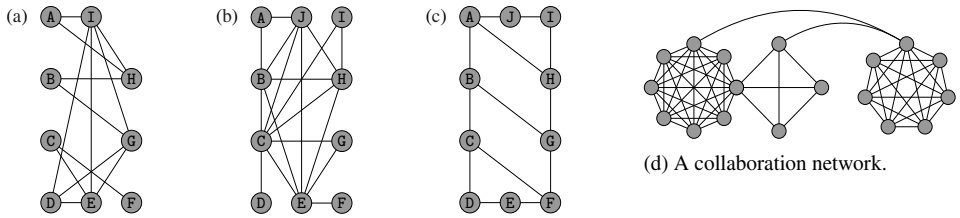


Figure 11.33 Three graphs in which to find cliques, and a collaboration network.

11.54 Consider the collaboration network in Figure 11.33d. (See Example 11.14.) Assuming that the nodes correspond to actors in movies, what is the *smallest number* of movies that could possibly have generated this collaboration network?

11.55 Are you certain that there weren't more movies than *[your answer to Exercise 11.54]* that generated this graph? Explain.

11.56 Consider the graph $V = \{1, 2, \ldots, n\}$ and $E = \{\langle i, i-1\rangle : i \geq 2\}$. For which n is this graph bipartite? Give proof.

11.57 For which n is the graph $V = \{0, 1, \ldots, n-1\}$ and $E = \{\langle i, i+1 \bmod n\rangle : i \geq 1\}$ bipartite? Give proof.

11.58 For which n is $\mathcal{K}_n$ (that is, the complete graph on n nodes) bipartite? Prove your answer.

11.59 Identify all values of n for which the following graph $G = \langle V, E\rangle$ is bipartite, and prove your answer. The set of nodes is $V = \{0, 1, \ldots, 2n-1\}$; the set of edges is $E = \{\langle i, (i+n) \bmod 2n\rangle : i \in V\}$.

11.60 Is the graph in Figure 11.34a bipartite? Explain.

11.61 Is the graph in Figure 11.34b bipartite? Explain.

Consider a bipartite graph with a set L of nodes in the left column and a set of nodes R on the right column, where $|L| = |R|$. Prove or disprove the following claims:

11.62 The sum of the degrees of the nodes in L must equal the sum of the degrees of the nodes in R.

11.63 The sum of the degrees of the nodes in L must be even.

11.64 The sum of the degrees of all nodes (that is, all nodes in $L \cup R$) must be an even number.

11.65 Suppose that G is a complete bipartite graph with n nodes—that is, $G = \mathcal{K}_{|L|,|R|}$ for $|L| + |R| = n$. What's the largest number of edges that can appear in G?

11.66 What's the smallest number of edges that can appear in a complete bipartite graph with n total nodes? *(Careful!)*

11.67 Suppose that G is graph that does not contain a triangle (that is, there is no set of three nodes a, b, and c with the edges $\{a, b\}$ and $\{b, c\}$ and $\{c, a\}$ all appearing in the graph). Prove or disprove: G is bipartite.

11.68 Definition 11.18 describes a regular undirected graph. In a *directed* regular graph, we require that there be two integers d_{in} and d_{out} such that every node's in-degree is d_{in} and every node's out-degree is d_{out}. Prove that $d_{\text{in}} = d_{\text{out}}$.

11.69 Show that the graph in Figure 11.34c is planar.

11.70 Show that the graph in Figure 11.34d is planar.

11.71 Prove that any 2-regular graph is planar.

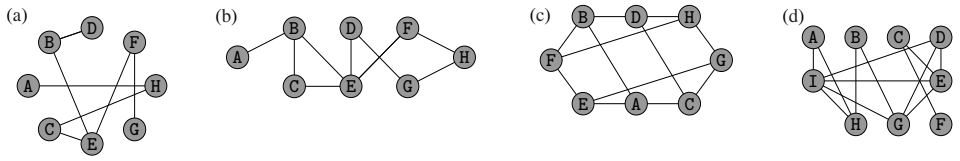


Figure 11.34 A few graphs.

11.3 Paths, Connectivity, and Distances

> I love North Dakota cause you have never been there
> and the days go on forever and the towns all look the same
> and I can ride the back roads and I can walk the main streets
> and show someone your picture but they would not know your name.
>
> Kris Delmhorst (b. 1970)
> excerpted lyrics from "North Dakota" (1998)

One of the most basic questions that one can ask about a graph is whether it is possible to get from some given node s to some given node t by following a sequence of edges. Is there some chain of handshakes that connects Bea Arthur to Arthur Conan Doyle? Can you get from Missoula to Madison by car? (And, if there is a way to get from s to t, what is the *shortest* way to get there?) These basic questions concern the existence of *paths* in the graph:

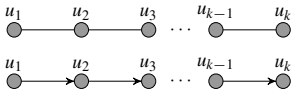
Definition 11.20: Path.
Consider a (directed or undirected) graph $G = \langle V, E \rangle$. A *path* in G is a sequence $\langle u_1, u_2, \ldots, u_k \rangle$ of $k \geq 1$ nodes such that:

- $u_i \in V$ for every $i \in \{1, \ldots, k\}$, and
- $\langle u_i, u_{i+1} \rangle \in E$ for every $i \in \{1, \ldots, k-1\}$.

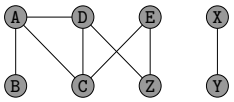
We say that such a sequence of nodes is *a path from u_1 to u_k*, and that this path has *length $k-1$*. We also say that this path *traverses* the edges $\langle u_i, u_{i+1} \rangle$.

See Figure 11.35a. (Note that this definition includes both directed and undirected graphs: if the edges are directed, we have to follow them "in the right direction.") For example, in the graphs shown in Figure 11.35b and 11.35c, there is no path from `A` to `X`. But, in both, the sequence $\langle \texttt{A}, \texttt{C}, \texttt{E}, \texttt{Z} \rangle$ is a path of length 3 from `A` to `Z`. In both cases, the edges traversed by the path are $\{\langle \texttt{A}, \texttt{C} \rangle, \langle \texttt{C}, \texttt{E} \rangle, \langle \texttt{E}, \texttt{Z} \rangle\}$. Notice that the length of a path is the number of *edges* that it traverses, which is one fewer than the number of nodes in the path.

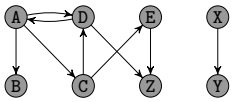
> **Taking it further:** A common mistake made by novice (and not-so-novice) programmers is an *off-by-one error* in specifying the bounds on a loop, by iterating either one time too many or one time too few. These errors are also sometimes called *fencepost errors*: if you build a 10-yard fence with posts placed every yard, then there are *eleven* fenceposts (at yard 0, yard 1, . . ., yard



(a) Paths in undirected and directed graphs.



(b) An undirected graph with a path from `A` to `Z`.



(c) A directed graph with a path from `A` to `Z`.

Figure 11.35 Paths in graphs.

10). Be careful! A path $\langle \mathtt{A}, \mathtt{C}, \mathtt{E}, \mathtt{Z} \rangle$ contains four nodes, but it traverses only three edges ($\mathtt{A} \to \mathtt{C}$, $\mathtt{C} \to \mathtt{E}$, and $\mathtt{E} \to \mathtt{Z}$) and has length 3.

Here's an example of finding paths in a small graph:

Example 11.21: Finding paths.
Consider the undirected graph in Figure 11.36.

1 Is there a path from node H to node E?

2 Name three different paths from node D to node F. What is the length of each?

Solution. For (1), yes, there is a path from node H to node E—for example, $\langle \mathtt{H}, \mathtt{A}, \mathtt{F}, \mathtt{G}, \mathtt{E} \rangle$.

For (2), two paths from D to F are shown in Figure 11.36b: the path $\langle \mathtt{D}, \mathtt{B}, \mathtt{E}, \mathtt{G}, \mathtt{F} \rangle$, which has length 4; and the path $\langle \mathtt{D}, \mathtt{B}, \mathtt{C}, \mathtt{E}, \mathtt{G}, \mathtt{F} \rangle$, which has length 5. Finding a third path might seem harder, but Definition 11.20 did not require that the nodes in a path be distinct from each other. (In other words, nothing forbade the repetition of nodes in a path.) So $\langle \mathtt{D}, \mathtt{B}, \mathtt{C}, \mathtt{E}, \mathtt{B}, \mathtt{C}, \mathtt{E}, \mathtt{G}, \mathtt{F} \rangle$ is a path from D to F, with length 8.

We will often restrict our attention to paths that never go back to a vertex that they've already visited, which are called *simple paths*:

Definition 11.21: Simple path.
A path $\langle u_1, u_2, \ldots, u_k \rangle$ is *simple* if all of the nodes $u_1, \ldots, u_k$ are distinct.

Of the three paths identified in Example 11.21, the first two are simple paths, but the third path is not simple because it repeated nodes $\{\mathtt{B}, \mathtt{C}, \mathtt{E}\}$.

11.3.1 Connectivity in Undirected Graphs

The most basic question about two nodes in a graph is whether it's possible to get from one to another—that is, are these two nodes *connected?* We start with a formal definition of connectivity for undirected graphs, because the relevant notions are a bit easier in the undirected setting.

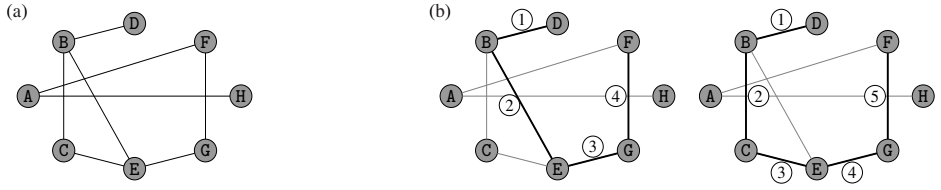


Figure 11.36 (a) An undirected graph, and (b) two paths from D to F: $\langle \mathtt{D}, \mathtt{B}, \mathtt{E}, \mathtt{G}, \mathtt{F} \rangle$ and $\langle \mathtt{D}, \mathtt{B}, \mathtt{C}, \mathtt{E}, \mathtt{G}, \mathtt{F} \rangle$.

Definition 11.22: Connected nodes and connected graphs.
Let $G = \langle V, E \rangle$ be an undirected graph.

- Two nodes $u \in V$ and $v \in V$ are *connected* if there exists a path from u to v.
- The graph G is *connected* if u and v are connected for *any* two nodes $u \in V$ and $v \in V$.
- The graph G is called *disconnected* if it is not connected.

For example, Figure 11.37a shows a disconnected graph—there's no path from `A` to `H`, for example—and Figure 11.37b shows a connected one. You can check that the second graph is connected by testing all pairs of nodes. (Exercise 11.87 asks you to show that connectivity is symmetric in an undirected graph: if there exists a path from u to v, then there exists a path from v to u.)

Example 11.22: Connectivity of an undirected graph.
Is the graph in Figure 11.38 connected?

Solution. No: odd-numbered nodes have edges only to other odd-numbered nodes, and even-numbered nodes have edges only to other even-numbered nodes. So there is no path from, for example, node `1` to node 2; this graph is disconnected.

Problem-solving tip: Sometimes it's very helpful to redraw a graph that you're given, with nodes placed more meaningfully. For example, the graph in Figure 11.38 can be redrawn as just by sliding the even-numbered nodes to the right. This visualization makes it clear that the graph is disconnected.

Connected components

More generally, we will talk about the *connected components* of an undirected graph $G = \langle V, E \rangle$—"subsections" of the graph in which all pairs of nodes are connected.

Definition 11.23: Connected component.
In an undirected graph $G = \langle V, E \rangle$, a *connected component* is a set $C \subseteq V$ such that
(i) any two nodes $s \in C$ and $t \in C$ are connected; and
(ii) for any node $x \in V - C$, adding x to C would make (i) false.

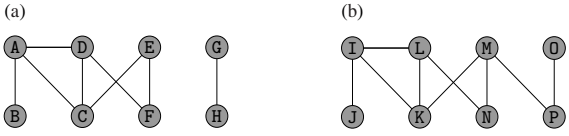


Figure 11.37 (a) A disconnected graph and (b) a connected graph.

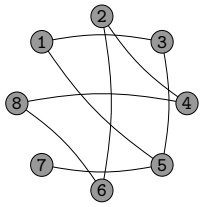
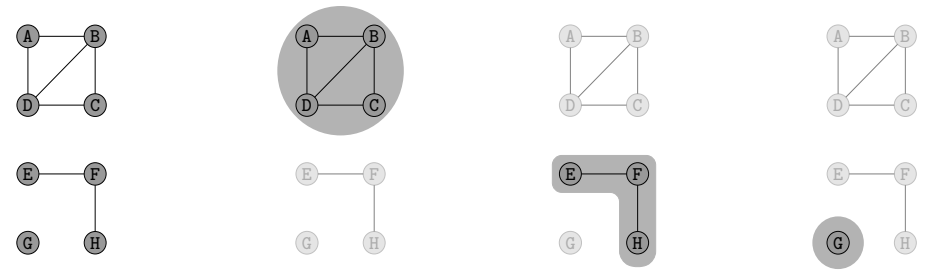


Figure 11.38 A possibly connected graph.



(a) The original graph. (b) Component #1. (c) Component #2. (d) Component #3.

Figure 11.39 A graph, and its three connected components.

A subset $C \subseteq V$ of nodes is a connected component of an undirected graph $G = \langle V, E \rangle$ if, intuitively, it forms its own "section" of the graph: any two nodes in C are connected, and no node in C is connected to any node not in C. For example, Figure 11.39 shows a graph with three connected components—one with 4 nodes, one with 3 nodes, and one with just a single node.

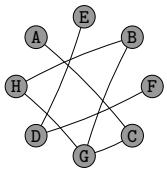
Note that we could have defined a "connected graph" in terms of the definition of connected components (instead of Definition 11.22): an undirected graph $G = \langle V, E \rangle$ is *connected* if it contains only one connected component, namely the entire node set V.

Example 11.23: Connected components of an undirected graph.
What are the connected components of the graph in Figure 11.40a?

Solution. The set $S = \{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{G}, \texttt{H}\}$ is a connected component; there are paths from every node $u \in S$ to every node $v \in S$, and furthermore no node in S is connected to any node not in S. To be thorough, paths connecting each pair of nodes from S are shown in Figure 11.40b.

Note that we haven't bothered to write down a path from u to v when we'd already recorded a path from v to u, because the graph is undirected and paths are symmetric. We also had many choices of paths for many of these entries: for example, other paths from B to H included $\langle \texttt{B}, \texttt{G}, \texttt{H} \rangle$ or $\langle \texttt{B}, \texttt{G}, \texttt{H}, \texttt{B}, \texttt{G}, \texttt{H} \rangle$.

There's a second connected component in the graph: the nodes $\{\texttt{D}, \texttt{E}, \texttt{F}\}$. It's not too hard to check that both clauses of Definition 11.23 are also satisfied for this set.



(a) The graph.

	A	B	C	G	H
A	⟨A⟩	⟨A, C, G, B⟩	⟨A, C⟩	⟨A, C, G⟩	⟨A, C, G, H⟩
B		⟨B⟩	⟨B, G, C⟩	⟨B, G⟩	⟨B, H⟩
C			⟨C⟩	⟨C, G⟩	⟨C, G, H⟩
G				⟨G⟩	⟨G, H⟩
H					⟨H⟩

(b) Paths connecting every pair of nodes in the largest connected component.

Figure 11.40 An undirected graph, and some paths connecting several pairs of nodes.

Observe that, in *any* undirected graph $G = \langle V, E \rangle$, there is a path from each node $u \in V$ to itself. Namely, the path is $\langle u \rangle$, and it has length 0. Check Definition 11.20!

Taking it further: There are many computational settings in which undirected paths are relevant; here's one example, in brief. In *computer vision,* we try to build algorithms to process—"understand," even—images. For example, before it can decide how to react to them, a self-driving car must partition the image of the world from a front-facing camera into separate objects: painted lines on the road, trees, other cars, pedestrians, etc. Here's a crude way to get started (real systems use far more sophisticated techniques): define a graph whose nodes are the image's pixels; there is an edge between pixels p and p' if (i) the two pixels are adjacent in the image, and (ii) the colors of p and p' are within a threshold of acceptable difference. The connected components of this graph are a (very rough!) approximation to the "objects" in the image. This description misses all sorts of crucial features of good algorithms for the image-segmentation problem, but even as stated it may be familiar from a different context: the "region fill" tool in image-manipulation software uses something very much like what we've just described.

11.3.2 Connectivity in Directed Graphs

Recall that we have to follow edges "in the right direction" in a directed graph G: as in Definition 11.20, a path from u_1 to u_k in G is a sequence $\langle u_1, u_2, \ldots, u_k \rangle$ where every pair $\langle u_i, u_{i+1} \rangle$ is an edge in G. Thus notions of connectivity in directed graphs are more complicated: the existence of a path from u to v does not imply the existence of a path from v to u. We will speak of a node t as being *reachable* from a node s if it's possible to go from s to t, and of pairs of nodes as being *strongly connected* when it's possible to "go in both directions" between them:

Definition 11.24: Reachability and strongly connected nodes/graphs.
Let $G = \langle V, E \rangle$ be a directed graph.

- A node $u \in V$ is *reachable from* a node $v \in V$ if there is a directed path from u to v.
- Two nodes $u \in V$ and $v \in V$ are *strongly connected* if u is reachable from v, *and* v is reachable from u.
- The graph G is *strongly connected* if every pair of nodes in V is strongly connected.

For example, you can check that the graph in Figure 11.41a is strongly connected by testing for directed paths between all pairs of nodes, in both directions. But the graph in Figure 11.41b is not strongly connected: there's no path from any node in the right-hand side (nodes $\{\mathtt{M}, \mathtt{N}, \mathtt{O}, \mathtt{P}\}$) to any node in the left-hand side (nodes $\{\mathtt{I}, \mathtt{J}, \mathtt{K}, \mathtt{L}\}$).

Figure 11.41 Two directed graphs: (a) one that's strongly connected, and (b) one that's not.

Strongly connected components

As with undirected graphs, for a directed graph we will divide the graph into "sections"—subsets of the nodes—each of which is strongly connected. These sections are called *strongly connected components* of the graph:

Definition 11.25: Strongly connected component.
In a directed graph $G = \langle V, E \rangle$, a *strongly connected component (SCC)* is a set $C \subseteq V$ such that:
(i) any two nodes $s \in C$ and $t \in C$ are strongly connected; and
(ii) for any node $x \in V - C$, adding x to C would make (i) false.

Figure 11.42 shows an example of a directed graph G and the three strongly connected components in G. The easiest strongly connected component to identify is $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$: we can go counterclockwise around the loop $\texttt{A} \to \texttt{B} \to \texttt{C} \to \texttt{D} \to \texttt{A}$, so we can go from any one of these four nodes to any other, and we can't get from any of these four nodes to any of the other nodes. The other two strongly connected components are $\{\texttt{E}, \texttt{F}, \texttt{H}\}$ and, separately, $\{\texttt{G}\}$ on its own. The reason is that G is not strongly connected to any other node: we can't get *from* G *to* any other node. (We can go around the $\texttt{E} \to \texttt{F} \to \texttt{H} \to \texttt{E}$ loop, so these three nodes are together in the other strongly connected component.)

Example 11.24: Finding strongly connected components.
What are the strongly connected components of the graph in Figure 11.43a?

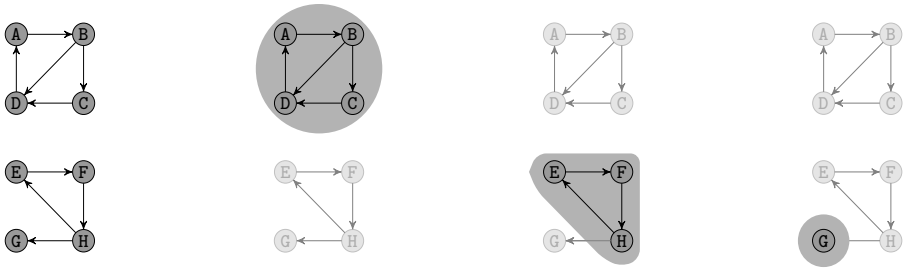
Solution. The three nodes $\{\texttt{C}, \texttt{D}, \texttt{E}\}$ form a strongly connected component: there is a path from any one of them to any other of them ($\texttt{C} \to \texttt{D} \to \texttt{E} \to \texttt{C} \to \texttt{D} \to \texttt{E} \cdots$), and furthermore there is no path from any $\{\texttt{C}, \texttt{D}, \texttt{E}\}$ to any other node in the graph.

In fact, every other node in the graph is alone in a strongly connected component by itself. For example, while there is a path from A to every node in the graph, there is no path *from* any other node to A. (There is a path from A to A, so the set $\{\texttt{A}\}$ is a strongly connected component.) Thus the four strongly connected components of the graph are $\{\texttt{A}\}$, $\{\texttt{B}\}$, $\{\texttt{F}\}$, and $\{\texttt{C}, \texttt{D}, \texttt{E}\}$.

Here's an example that shows why the second clause of Definition 11.25 is crucial:

Example 11.25: A non-SCC.
In the graph in Figure 11.43b, the set $S = \{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{E}, \texttt{F}\}$ is *not* a strongly connected component. Why not?

Solution. It is indeed the case that there is a path in both directions between any two nodes in S: we can just keep "going around" clockwise in S and we eventually reach every other node in S. So S satisfies clause (i) of Definition 11.25. But it fails to satisfy clause (ii) of Definition 11.25: if we considered the



(a) The original graph. (b) SCC #1. (c) SCC #2. (d) SCC #3.

Figure 11.42 A graph and its strongly connected components.

Figure 11.43 Two directed graphs.

set $S^+ = S \cup \{\texttt{D}\}$, it is still the case that there is a path in both directions between any nodes in S^+. Thus S is not a strongly connected component!

On the other hand, $S^+ = \{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}, \texttt{F}\}$ *is* a strongly connected component: we can't add any other node (specifically G; it's the only other node) to S^+ without falsifying this property—because there's no path from G to A, for example. Thus the two strongly connected components are $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{E}, \texttt{F}\}$ and $\{\texttt{G}\}$.

Taking it further: There are many computational settings in which directed paths, reachability, and strongly connected components are relevant. For example, for a spreadsheet, consider a directed graph whose nodes are the spreadsheet's cells, and an edge $\langle u, v \rangle$ indicates that u's contents affect the contents of cell v; when a user changes the content of cell c, we must update all cells that are reachable from node c. For a chess-playing program, consider a directed graph whose nodes are board configurations, and there's an edge $\langle u, v \rangle$ if a legal move in u can result in v; any configuration u that's unreachable from the starting board configuration can never occur in chess, and thus your program doesn't have to bother evaluating what move to make in position u. See p. 11-49 for a discussion of another application of reachability and strongly connected components: the structure of the world-wide web, understood with respect to the directed paths in the graph defined by the pages and the hyperlinks of the web.

11.3.3 Shortest Paths and Distance

So far we have concentrated on the basic question of connectivity: for a given pair of nodes, does any path exist from one node to the other? Here we address a more refined question: what is the *shortest* path that goes from one node to the next?

Definition 11.26: Shortest paths.
Let $G = \langle V, E \rangle$ be a graph (undirected or directed), and let $s \in V$ and $t \in V$ be two nodes. A path from s to t is a *shortest path* if its length is the smallest out of all s-to-t paths.

(Recall that the *length* of a path $\langle u_1, u_2, \ldots, u_k \rangle$ is $k - 1$, the number of edges that it traverses.) There may be more than one shortest path from a node s to a node t, if there are multiple paths that are tied in length.

Definition 11.27: Distance.
The *distance* from s to t is the length of a shortest path from s to t. If there is no path from s to t, then we say that the distance from s to t is infinite (written as "∞").

Example 11.26: Some shortest paths.
Consider the undirected graph in Figure 11.44a. The distance from `A` to `A` is 0 because $\langle$`A`$\rangle$ is a path from `A` to `A`. This graph also has an example of a pair of nodes connected by two different shortest paths, going from `A` to `C` (via either `B` or `E`).

For the directed graph in Figure 11.44b, again, there's a path from `G` to `G` of length zero, so the distance from `G` to `G` is 0. Note that there's no `G`-to-`J` path of length two (because the edge from `J` to `K` goes in the wrong direction), so the distance from `G` to `J` is 3 (via `K` and `I`, or via `H` and `I`). Similarly, there is no directed path from `G` to `L`, so the distance is infinite.

Example 11.27: Shortest paths in directed graphs.
Find the shortest path from `A` to `L` in the graph with the adjacency list shown in Figure 11.45a.

Solution. The nodes at distance 1 from `A` are `B`, `D`, `E`, `F`, and `G`. There's no edge from any of those nodes to `L`—or indeed to `K`, which is `L`'s only in-neighbor. Thus the distance from `A` to `L` cannot be any smaller than 4. But there is an edge from `I` to `K`, and one from `B` to `I`. We can assemble these edges into the path $\langle$`A`, `B`, `I`, `K`, `L`$\rangle$. This path has length 4. So the distance from `A` to `L` is 4. (Drawing the graph, as in Figure 11.45b, with nodes arranged by their distance from `A`, can make these facts easier to see.)

Problem-solving tip: In solving any graph problem with a small graph, a good first move is to draw the graph.

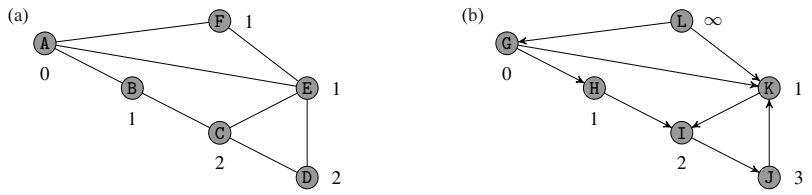
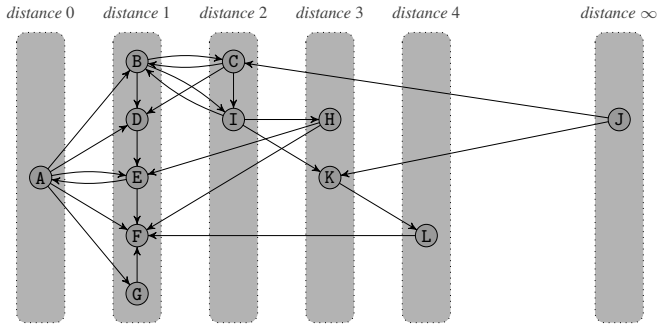


Figure 11.44 Some distances, in undirected and directed graphs. The nodes in (a) are marked with their distances from `A`; the nodes in (b) are marked with their distances from `G`.

```
A:  B, D, E, F, G
B:  C, D, I
C:  B, D, I
D:  E
E:  A, F
F:
G:  F
H:  E, F
I:  B, H, K
J:  C, K
K:  L
L:  F
```

(a) An adjacency list.



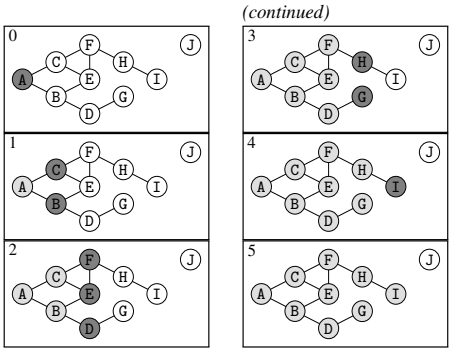
(b) A drawing of the graph, with one layer for each distance from A.

Figure 11.45 A directed graph.

11.3.4 Finding Paths: Breadth-First Search (BFS)

There are many aspects of graphs that are valuable for interesting computational applications, but perhaps the single most important graph algorithm is *breadth-first search (BFS)*. BFS is a path-finding algorithm: it explores outward from a given source node s in a given graph G until it finds every node reachable from s in G. BFS can be used to solve all sorts of graph-related problems, as we'll see.

Here's the intuition of the algorithm. (See Figure 11.46a.) We maintain a set of nodes that are reachable from the given node s (the shaded nodes in Figure 11.46a). To start, the reachable set is just $\{s\}$. Now we find all as-yet-undiscovered neighbors of nodes in our set, and add those nodes (the darker-shaded nodes in Figure 11.46a) to the set of reachable nodes: if $\langle u, v\rangle \in E$ and you can reach the node u from s, then you can also reach v from s, via u. But now we've found some more nodes that can be reached from s, which means that we can also reach any nodes that are directly connected to *them* from s. So we'll repeat that



(a) The steps of BFS on a small graph, starting at A.

Breadth-First Search (BFS):
Input: a graph $G = \langle V, E\rangle$ and a source node $s \in V$
Output: the set of nodes reachable from s in G

```
1  Frontier := ⟨s⟩
   // Frontier will be a list of nodes to process, in order.
2  Known := ∅
   // Known will be the set of already-processed nodes.
3  while Frontier is nonempty:
4      u := the first node in Frontier
5      remove u from Frontier
6      for every neighbor v of u:
7          if v is in neither Frontier nor Known then
8              add v to the end of Frontier
9      add u to Known
10 return Known
```

(b) The pseudocode for breadth-first search.

Figure 11.46 The intuition of a small example, and the pseudocode of breadth-first search.

process with the updated set of reachable nodes. And we'll do it again, and again, and again, until we stop finding new nodes.

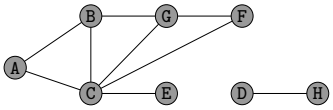
Observe that BFS discovers nodes in order of their *distance* from the source node. Every expansion of the set of reachable nodes takes the full breadth of the frontier and expands it out by one more "layer" in the graph. (That's why the algorithm is called breadth-first search.) You can think of BFS as throwing a pebble onto the graph at the node s, and then watching the ripples expanding out from s.

Breadth-first search is presented more formally in Figure 11.46b. (While we've described BFS in terms of undirected graphs for simplicity, it works equally well for directed graphs. The only change is that Line 6 should say "for every *out*-neighbor" for a directed graph.)

Here's another example of breadth-first search in action, running the algorithm in full detail (precisely as specified in Figure 11.46b):

Example 11.28: Sample run of BFS, in detail.
Let's trace BFS starting at node `A` in the graph shown in Figure 11.47.

Every step of the algorithm's execution is shown in Figure 11.48. Because *Frontier* is empty after the last step shown in Figure 11.48, the **while** loop in BFS terminates. The algorithm returns the final set *Known*, which is $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{G}, \texttt{E}, \texttt{F}\}$.




```
A: B, C
B: A, C, G
C: A, B, E, F, G
D: H
E: C
F: C, G
G: B, C, F
H: D
```

Figure 11.47 A sample graph for BFS, shown both as a picture and as an adjacency list.

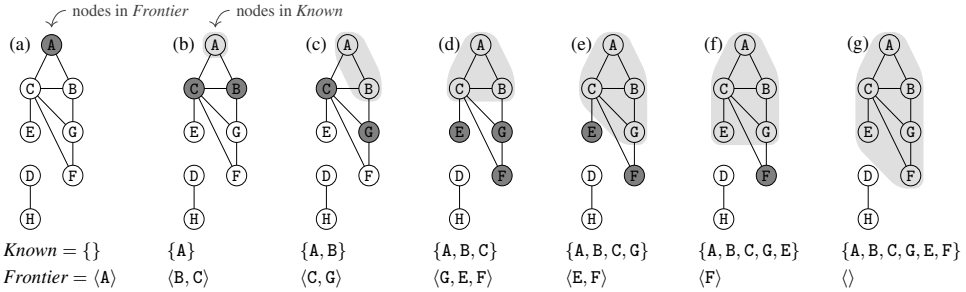


Figure 11.48 Every step of BFS on the graph in Figure 11.47: (a) the initialization (Lines 1–2 of the BFS pseudocode from Figure 11.46b); and (b–g) the processing of the next node (Lines 4–9) [in order, the processed nodes are: `A`, `B`, `C`, `G`, `E`, and `F`].

We'll prove two important properties of BFS. The first is *correctness:* the set that BFS returns is precisely those nodes that are reachable from the starting node. The second is *efficiency:* BFS finds this set quickly. The first claim might seem obvious—and thus proving it may feel annoyingly pedantic—but there's a bit of subtlety to the argument, and it's good practice at using induction in proofs besides.

Correctness of BFS

Theorem 11.28: Correctness of BFS.
Let $G = \langle V, E \rangle$ be any graph, and let $s \in V$ be an arbitrary node. Then the set of nodes discovered by **BFS**(G, s) is exactly $\{t \in V : t \text{ is reachable from } s \text{ in } G\}$.

We'll prove the result by showing two set inclusions: the discovered nodes form a subset of the reachable nodes, and the reachable nodes form a subset of the discovered nodes. Both proofs will use induction, though on different quantities.

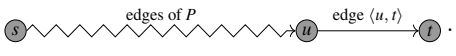
> *Problem-solving tip:* The hard part here is figuring *on what quantity* to do induction. One way to approach this question is to figure out a recursive way of stating the correctness claim. Question #1: why is there a path to every node added to *Frontier*? (Answer #1: there was a path to every previous node in *Frontier*, and there's an edge from some previously added node to this one!) Question #2: why is every node u reachable from s eventually added to *Frontier*? (Answer #2: because a neighbor of u that's closer to s is eventually added to *Frontier*, and every neighbor of a node in *Frontier* is eventually added to *Frontier*!)

Proof of Theorem 11.28, Part #1: every node returned by **BFS**(G, s) *is reachable from s in G.* By examining the algorithm, we see that (i) BFS returns the set of nodes that end up in the *Known* set, and (ii) the only way that a node ends up in *Known* is having previously been in *Frontier*. Thus it will suffice to prove the following property for all $k \geq 0$, by strong induction on k:

$$Q(k) = \text{if a node } t \in V \text{ is added to the list } \textit{Frontier} \text{ during the } k\text{th iteration of the } \textbf{while} \text{ loop of BFS, then there is a path from } s \text{ to } t.$$

For the base case ($k = 0$), we must show that there is a path from s to any node t added to *Frontier* during the 0th iteration of the **while** loop—that is, before the **while** loop begins. Then t was added in Line 1 of **BFS**, and therefore t is actually the node s itself. There is a path from s to s itself in any graph, and thus $Q(0)$ holds.

For the inductive case ($k \geq 0$), we assume the inductive hypotheses $Q(0), \ldots, Q(k-1)$, and we must prove $Q(k)$. Consider a node t that was added to *Frontier* during the kth iteration of the **while** loop—in other words, t was added in the **for** loop (Lines 6–8) because t is a neighbor of some node u that was already in *Frontier*. That is, we know that $\langle u, t \rangle \in E$ and that u was added to *Frontier* in the (k')th iteration, for some $k' < k$. By the inductive hypothesis $Q(k')$, there is a path P from s to u. Therefore there is a path from s to t, too:



□

Proof of Theorem 11.28, Part #2: every node reachable from s in G is returned by **BFS**(G, s). If a node t is reachable from s in G, then by definition the distance from s to t is some integer $d \geq 0$. Furthermore, by inspection of the algorithm, we see that any node that's added to *Frontier* is eventually moved to *Known*. Thus it will suffice to prove the following property for all $d \geq 0$, by (weak) induction on d:

$$R(d) = \text{if a node } t \in V \text{ at distance } d \text{ from } s, \text{ then } t \text{ is eventually added to } \textit{Frontier}.$$

For the base case ($d = 0$), we must prove $R(0)$: any node t at distance 0 is eventually added to *Frontier*. But the only node at distance 0 from s is s itself, and **BFS** adds s itself to *Frontier* in Line 1 of the algorithm.

For the inductive case ($d \geq 1$), we assume the inductive hypothesis $R(d-1)$, and we must prove $R(d)$. Let t be a node at distance d from s. Then by definition of distance there is a shortest path P of length d from s to t. Let u be the node immediately before t in P. Then the distance from s to u must be $d-1$, and therefore by the inductive hypothesis $R(d-1)$ the node u is added to *Frontier* in some iteration of the **while** loop. There are at most $|V|$ iterations of the loop, and thus eventually u is the first node in *Frontier*. In that iteration, the node t is added to *Frontier* (if it had not already been added). Thus $R(d)$ follows. □

(In the exercises, you'll show how to modify BFS so that it actually computes *distances* from s, using an idea very similar to the proof of Part #2 of Theorem 11.28.)

Running time of BFS

Theorem 11.29: Efficiency of BFS.
For a graph $G = \langle V, E \rangle$ represented using an adjacency list, BFS takes $\Theta(|V| + |E|)$ time.

Proof. See Figure 11.49a for a reminder of the algorithm. Lines 1, 2, and 10 take $\Theta(1)$ time, so the only question is how long the **while** loop takes. In the worst case, every node in the graph is reachable from the node from which BFS is run. In this case, there is one iteration of the **while** loop for every node $u \in V$.

The key question, then, is how long the body of the **while** loop (Lines 4–9) takes for a particular node u. Lines 4, 5, and 9 take $\Theta(1)$ time. But what about the **for** loop in Lines 6–8? The **for** loop has one iteration *for each neighbor of u.* (In an adjacency list, the loop simply steps through the list of neighbors, one by one.) Each **for**-loop iteration takes $\Theta(1)$ time, and there are $degree(u)$ iterations for node u.

Therefore, ignoring multiplicative constants, the worst-case running time of BFS is

$$1 + \sum_{u \in V} \Big[1 + degree(u) \Big] = 1 + \underbrace{\sum_{u \in V} 1}_{= |V|} + \underbrace{\sum_{u \in V} degree(u)}_{\substack{= 2|E| \text{ for an undirected graph (by Theorem 11.8),} \\ \text{or } |E| \text{ in a directed graph (by Exercise 11.18)}}} \qquad \textit{rearranging the summation}$$

$$= \Theta(|V| + |E|).$$

□

Taking it further: BFS arises in applications throughout computer science, from network routing to artificial intelligence. Another application of BFS occurs (hidden from your view) as you use programming languages like Python and Java, through a language feature called *garbage collection.* In garbage-collected languages, when you as a programmer are done using whatever data you've stored in some chunk of memory, you just "drop it on the floor"; the "garbage collector" comes along to reclaim

that memory for other use in the future of your program. The garbage collector runs BFS-like algorithms to determine whether a particular piece of memory is actually trash. See p. 11-51.

11.3.5 Finding Paths: Depth-First Search (DFS)

Another important algorithm for exploring graphs is called *depth-first search (DFS),* whose pseudocode is shown in Figure 11.49b. Informally, instead of exploring outward from the source node s in "layers" as in BFS, in DFS we will try to explore a new node at every stage of the search. We start at s, and at every stage we move to an unvisited neighbor of our current node. If at any stage we're stuck at a node u that has no unvisited neighbors, we go back from u to the node from which we first reached u and continue exploring from there. An example of DFS in a small graph is shown, informally, in Figure 11.50.

Intuitively, depth-first search is a close match for the way that you would explore a maze: you start at the entrance, follow a passageway to a location you've never visited before; using breadcrumbs or a pencil, you remember where you've been and backtrack if you get stuck. You may have heard of another algorithm for mazes: *Place your right hand on the wall as you go in the entrance. Continue to walk forward, always keeping your right hand on the wall. Eventually, you will get out of the maze.* In fact, this right-hand-on-the-wall algorithm is identical in spirit to DFS: whenever you encounter a choice, you always choose the first (right-most) unexplored passageway, and if you ever get stuck at a dead end you turn around and go back from whence you came.

We can implement DFS with only a small change to BFS, as shown in Figure 11.49b: instead of putting a newly discovered node u at the *end* of the list *Frontier* of nodes from which to explore (as in BFS), we put a newly discovered node u at the *beginning* of *Frontier*. (In other words, *BFS treats the list* Frontier *as a* queue*—first in, first out—while DFS treats the list* Frontier *as a* stack*—last in, first out.*) Another small change is necessary, to allow a node already in *Frontier* to be "moved" earlier in the list of nodes to explore.

Breadth-First Search (BFS):
Input: a graph $G = \langle V, E \rangle$ and a source node $s \in V$
Output: the set of nodes reachable from s in G

```
1  Frontier := ⟨s⟩
2  Known := ∅
3  while Frontier is nonempty:
4      u := the first node in Frontier
5      remove u from Frontier

6      for every neighbor v of u:
7          if v is in neither Frontier nor Known then
8              add v to the end of Frontier
9      add u to Known
10 return Known
```

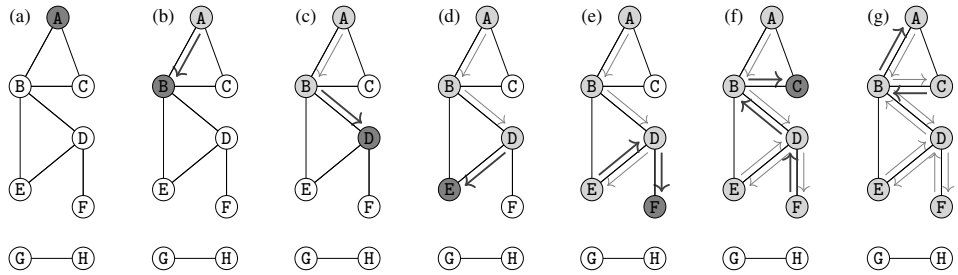
(a) A reminder of BFS.

Depth-First Search (DFS):
Input: a graph $G = \langle V, E \rangle$ and a source node $s \in V$
Output: the set of nodes reachable from s in G

```
1  Frontier := ⟨s⟩
2  Known := ∅
3  while Frontier is nonempty:
4      u := the first node in Frontier
5      remove u from Frontier
6      if u is not in Known then
7          for every neighbor v of u:
8              if v is not in Known then
9                  add v to the start of Frontier
10         add u to Known
11 return Known
```

(b) The pseudocode for depth-first search. The only changes from BFS are underlined.

Figure 11.49 A reminder of breadth-first search, and the pseudocode of depth-first search.



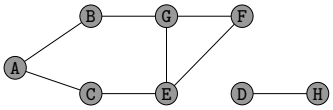
We start exploring node A; in each frame, the dark-shaded node is the current node. Previously discovered nodes are lightly shaded. Arrows indicate the steps of the exploration.

In each of (a–d), we move from the current node to a neighbor that is unexplored. (We pick the alphabetically first node if there's a choice.)

In (e), the current node E has no unvisited neighbors, so we backtrack from E to D to find D's unvisited neighbor F. Similarly, in (f) we backtrack from F to D to B to discover the new node C, and in (g) we backtrack from C to B to A. At this point there are no further unexplored nodes from any of these nodes, and thus the algorithm terminates.

Figure 11.50 A sample run of depth-first search, starting at node A.

Because this alteration of BFS changes only the order in which the nodes in *Frontier* are explored, DFS does precisely the same work as BFS, and is correct for the same reasons: DFS returns precisely the set of nodes reachable from the given source node s. (With a little more cleverness in moving nodes to the front of *Frontier*, DFS can also be implemented in $\Theta(|V| + |E|)$ time.) Here's a fully detailed example of DFS:



```
A:  B, C
B:  A, G
C:  A, E
D:  H
E:  C, F, G
F:  E, G
G:  B, E, F
H:  D
```

Figure 11.51 A sample graph for DFS, shown both as a picture and as an adjacency list.

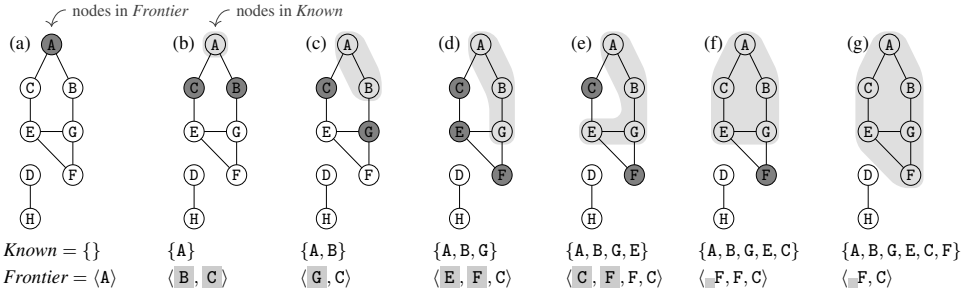


Figure 11.52 Every step of DFS on the graph in Figure 11.51: (a) initialization; (b–g) processing the node at the beginning of *Frontier* [in order, the processed nodes are: A, B, G, E, C, and F]. The names of the nodes that were just added to *Frontier* are highlighted. When nodes are in *Known*, they are not re-added to *Frontier* [as with A in (c), B in (d), G in (e), and both A and E in (f)].

Example 11.29: Sample run of DFS, in detail.
We'll trace DFS starting at node `A` in the graph shown in Figure 11.51. Every substantive step of the algorithm's execution is shown in Figure 11.52; after the last step shown in the figure, there are two more iterations that remove the last two entries in *Frontier* (making no changes to *Known* and adding nothing further to *Frontier*), because both `F` and `C` are already in *Known*. The **while** loop then terminates, and DFS returns $\{\texttt{A}, \texttt{B}, \texttt{G}, \texttt{E}, \texttt{C}, \texttt{F}\}$.

Computer Science Connections

The Bowtie Structure of the Web

As the web has grown more and more central in the daily lives of us all, it has garnered massive attention from researchers in computer science. A great deal of the early work performed attempted to characterize the web in terms of its degree distribution (see p. 11-26) or in terms of the "small-world phenomenon" (see p. 4-46). But one foundational and influential paper sought to characterize the web's structure in terms of its strongly connected components [22]. In the early days of the web, eight researchers from AltaVista, IBM, and Compaq downloaded around 200 million web pages, comprising about 1.5 billion links. They then analyzed the structure of the resulting graph, by categorizing the pages in terms of which pages could reach which other pages.

First, think about the web pages contained in the largest strongly connected component of the web graph. Like many other networks (for example, social networks and collaboration networks), the web graph has a so-called *giant component* that contains many more nodes than the second-largest strongly connected component. Denote by CORE those nodes in the largest SCC in the web graph.

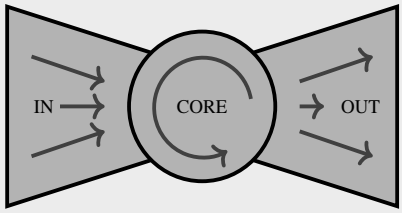


Figure 11.53 The "bowtie structure" of the web graph, in its basic form. Broder et al. [22] found that roughly 25% of web pages fell into each of these categories: 56M pages (of 200M) in CORE, 43M pages in IN, and 43M pages in OUT.

Second, imagine a web page from which you can reach the nodes in the largest SCC by following links, but that's not itself inside the giant component. Let IN denote those web pages p such that (i) $p \notin$ CORE, and (ii) there is a path from p to some node in CORE. (And so there is a path from p in IN to every page in CORE, but there's no path from any node in CORE to p.) And let OUT denote the downstream analogue of IN: pages reachable by following links from pages in CORE, but that are not themselves in CORE. That is, OUT contains those pages p such that (i) $p \notin$ CORE, and (ii) there is a path from some node in CORE to p. (And therefore there is a path from *every* page in CORE to page p.)

When displayed graphically, as in Figure 11.53, these categories of web pages look like a bowtie, and so the paper by Broder et al. came to be known as "the bowtie paper." To complete the picture of the bowtie structure of the web, we must note that not all web pages are included in Figure 11.53. (It's worth thinking through the reasons that we're not yet done—that is, the multiple ways in which a page on the web can fall outside of CORE, IN, and OUT.)

There are three further categories of nodes: (a) shortcuts from IN to OUT that bypass CORE (call these TUBES); (b) ways out of IN that lead nowhere, and ways into OUT that come from nowhere (call these TENDRILS); and (c) nodes that have no paths to

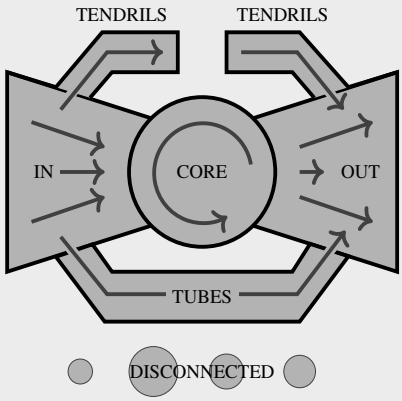


Figure 11.54 The remainder of the "bowtie structure" of the web graph. There were about 44M pages in TENDRILS and TUBES, and about 17M pages in DISCONNECTED.

any of the nodes we've discussed so far (call these DISCONNECTED). See Figure 11.54.

One of the unexpected facts found by Broder et al. was the extent to which the web is actually *not* particularly well connected. In particular, if we were to choose web pages p and q uniformly at random from the web graph, there was only a roughly 24% chance of that a directed path from p to q exists—far lower than the "small world" phenomenon would suggest. (For much more, see the original bowtie paper [22] or a book on the structure of social and technological networks, like [43].)

Computer Science Connections

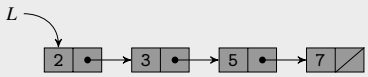
Garbage Collection

In many modern programming languages, including Python and Java, the burden of managing memory—allocating memory for new objects, deallocating memory for objects that are no longer needed—is lifted from the shoulders of the programmer. When a new object is needed, the programmer just creates it.

After a program has been running for a while, there may be objects that were stored in memory but are now *inaccessible* because the programmer has no way to refer to them ever again. This stored but inaccessible data is called *garbage*. Figure 11.55 shows an example. In Python- and Java-like languages, the system provides a *garbage collector* that periodically runs to clean up the garbage, which allows that memory to be reused for future allocations. (In contrast, in languages like C or C++, when you as a programmer are done using a chunk of memory, it's your responsibility to declare to the system that you're done using that memory by explicitly "deallocating" or "freeing" it.)

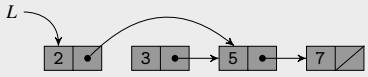
```
1 L := Node(7, NULL)
2 L := Node(5, L)
3 L := Node(3, L)
4 L := Node(2, L)
```

Suppose that `Node(data,next)` creates a new node for a singly linked list, with data `data` and with a pointer `next` to the next node in the list.



(a) Some code, and the state of memory after executing it.

```
5 L.next := L.next.next
```



(b) If we continue by executing `L.next := L.next.next`, then the node whose data field is 3 is now *garbage:* there is no way to access that memory again, because there is no way for the programmer to refer to it.

Figure 11.55 Garbage being created.

The garbage-collection algorithms employed in real systems are sophisticated, but fundamentally the algorithmic idea is based on finding reachable nodes in a graph. There is a *root set* of memory locations that are reachable—essentially every variable that's defined in any currently active function call on the stack. Furthermore, if a memory location ℓ is pointed to by a reachable memory location, then ℓ too is reachable. Two of the simpler algorithms that are sometimes used in garbage collection are based on some corresponding simple graph-theoretic approaches. Here's a brief description of these two garbage-collection algorithms:

Reference counting: For each block b of memory, we maintain a *reference count* of the number of other blocks of memory (or root set variables) that refer to b. When the garbage collector runs, any block b that has a reference count equal to 0 is marked as garbage and reclaimed for future use.

Mark-and-sweep: When the garbage collector runs, we iteratively *mark* each block b that is accessible. Specifically, for every variable v in the root set, we mark the block to which v refers. Then, for any block b that is marked, we also mark any block to which b refers. Once the marking process is completed, we *sweep* through memory, and reclaim all unmarked blocks.

In graph-theoretic terms, we view memory as a directed graph, with an edge from each block b to the block(s) to which b refers. Reference counting declares as garbage any node with in-degree 0; mark-and-sweep declares as garbage any node that is not reached by BFS starting

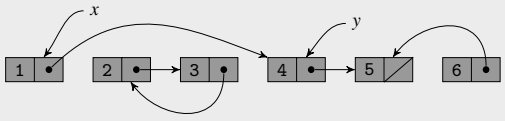


Figure 11.56 Six blocks of memory, and two root set variables x and y. Reference counting would show block #6 with a reference count of zero, and therefore it would be reclaimed. Mark-and-sweep would mark blocks #1, #4, and #5; thus it would reclaim blocks #2, #3, and #6.

from the root set. Reference counting is a simpler algorithm, but it has a problem with cyclical structures. If two inaccessible blocks of memory refer to each other, they both have nonzero reference count, and therefore won't be marked as garbage. An example is shown in Figure 11.56. There are issues of efficiency with mark-and-sweep (the entire system has to pause while the garbage collector runs), and so other, more sophisticated algorithms are generally used in real systems. (You can learn more about garbage collection in any good textbook on programming languages—or even more in a book devoted exclusively to garbage collection, like [65].)

EXERCISES

11.72 Find a path from `D` to `B` in Figure 11.57a.

11.73 Find two different paths from `C` to `H` in Figure 11.57a.

11.74 Find a path from `C` to `B` in Figure 11.57b.

11.75 Find two different paths from `A` to `H` in Figure 11.57b.

11.76 Find a path from `D` to `H` in Figure 11.57b that is *not* simple.

11.77 Find a path from `B` to `C` in the graph defined by the adjacency list in Figure 11.57c.

11.78 Find a shortest path from `B` to `F` in Figure 11.57d.

11.79 Find a *non–shortest* path from `B` to `C` in the graph defined by the adjacency matrix in Figure 11.57e.

11.80 Find all nodes reachable from `A` in Figure 11.57d.

11.81 Find all nodes reachable from `A` in Figure 11.57e.

11.82 Identify all of the connected components in the graph in Figure 11.57a. Is the graph connected?

11.83 Identify all of the strongly connected components in the graph in Figure 11.57b. Is the graph strongly connected?

11.84 Identify all of the connected components in the graph in Figure 11.57c.

11.85 Identify all of the strongly connected components in the graph in Figure 11.57d.

11.86 Decide whether the graph in Figure 11.57e is directed or undirected, and then identify all of its (strongly) connected components.

11.87 Let s and t be any two nodes in an undirected graph, and let $k \geq 0$ be any integer. Prove the following: if there's a path of length k from s to t, then there's a path of length k from t to s.

11.88 Let s and t be any two nodes in an undirected graph. Prove that every shortest path between s and t is a simple path.

11.89 The *diameter* of a graph G is the largest node-to-node distance in the graph. (Although the context is different, this definition of "diameter" matches the idea from geometry: the diameter of a circle is the distance between the two points in the circle that are farthest apart. That's still true for a graph.) In terms of n, what is the *smallest* diameter that an n-node undirected graph can have? Prove your answer.

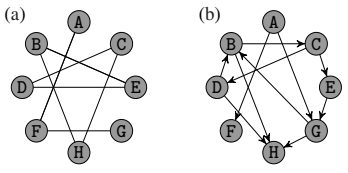
11.90 In terms of n, what is the *largest* diameter that a connected n-node undirected graph can have? Give an example of a graph where the diameter is this large. In other words, assuming that G is connected, what's the largest possible distance between two nodes in G? (Without the restriction that the graph be connected, the answer would be ∞.)

11.91 Consider an n-node 3-regular undirected graph G. (That is, $G = \langle V, E \rangle$ has $|V| = n$, and each node $u \in V$ has degree exactly equal to 3.) In terms of n, what is the *largest* possible number of connected components in a 3-regular graph?

11.92 In terms of n, what is the *smallest* possible number of connected components in a 3-regular graph with n nodes?

11.93 Describe a connected 3-regular graph with n nodes with a diameter that's at least $\frac{n}{8}$. (See Exercise 11.89.)

11.94 Describe a connected 3-regular graph with n nodes with a diameter that's at most $8 \log n$. (See Exercise 11.89.)



(c)

```
A: B, E, F
B: A
C: D
D: C, F
E: A
F: A, D
```

(d)

```
A: B, C
B: A, C
C: A, B, F
D: E
E: B, D
F: C
```

(e)

	A	B	C	D	E	F	G	H
A	0	0	0	1	0	0	0	1
B	0	0	1	0	1	0	1	0
C	0	1	0	1	0	0	0	0
D	1	0	1	0	0	1	0	0
E	0	1	0	0	0	0	0	1
F	0	0	0	1	0	0	0	1
G	0	1	0	0	0	0	0	0
H	1	0	0	0	1	1	0	0

Figure 11.57 Several graphs, described by picture, adjacency list, or adjacency matrix.

11.95 Prove or disprove: let $G = \langle L \cup R, E \rangle$ be a bipartite graph with $|L| = |R|$. Suppose that every node in the graph (that is, all nodes in L and R) has at least one neighbor. Then the graph is connected.

11.96 Consider an undirected graph G. Recall that a simple path from s to t in G does not go through any node more than once. A *Hamiltonian path* from s to t in G is a path from s to t that goes through each node of G *precisely* once. (Hamiltonian paths are named after William Rowan Hamilton (1805–1865), an Irish mathematician and physicist.) In general, finding Hamiltonian paths in a graph is believed to be computationally very difficult. But there are some specific graphs in which it's not too hard to find one. Here's one: find a Hamiltonian path in the Petersen graph (see Figure 11.58a).

11.97 Let $\mathcal{K}_n$ be a complete graph, and let s and t be two distinct nodes in the graph. How many different Hamiltonian paths are there from s to t? (See Exercise 11.96.)

11.98 Let $\mathcal{K}_{n,m}$ be a complete bipartite graph with $n + m$ nodes, and let s and t be two distinct nodes in the graph. How many different Hamiltonian paths are there from s to t? (See Exercise 11.96.) *(Careful; your answer may depend on s and t.)*

The diameter *of an undirected graph $G = \langle V, E \rangle$—the maximum distance between any two nodes $s \in V$ and $t \in V$ (see Exercise 11.89)—is one measure of how far a graph "sprawls." But another way of measuring this idea is by looking at the* average distance *instead. That is, for a pair of distinct nodes $\langle s, t \rangle$ chosen uniformly from the set V, what's the distance from s to t? In other words, the average distance of a graph $G = \langle V, E \rangle$ is defined as*

$$\text{the average distance of } G = \frac{\sum_{s \in V} \sum_{t \in V : t \neq s} \text{distance}(s, t)}{n(n-1)}.$$

(There are $n(n-1)$ ordered pairs of distinct nodes.) Often the average distance is a bit harder to calculate than the maximum distance, but in the next few exercises you'll look at the average distance for a few well-structured graphs.

11.99 Consider an n-node *cycle*, where n is odd. (We'll see a formal definition of a cycle in Section 11.4, but for now just look at the example with $n = 15$ in Figure 11.58b.) Compute the average distance in this n-node graph. *(Hint: every node is positioned symmetrically, so you can just figure out the average distance from some particular node u.)*

11.100 What is the average distance for an n-node cycle where n is even? (See the example with $n = 16$ in Figure 11.58c.)

11.101 What is the average distance for an n-node path? (See the examples in Figures 11.58d and 11.58e.) *(Hint: for any particular integer k, how many pairs of nodes have distance k? Then simplify the summation.)*

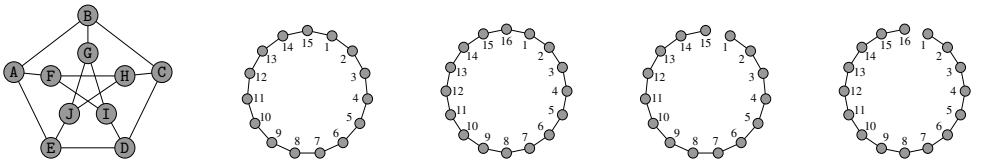
11.102 *(programming required.)* Write a program, in a language of your choice, to verify your answers to the last three exercises: build a graph of the appropriate size and structure, sum all of the node-to-node distances, and compute their average.

11.103 Suppose that G is a disconnected undirected graph with n nodes. In terms of n, what is the largest possible number of edges that G can contain?

11.104 Suppose that G is a connected undirected graph with n nodes. What is the smallest possible number of edges in G?

11.105 Suppose that G is a strongly connected directed graph with n nodes. What is the smallest number of edges that G can contain?

11.106 Suppose that G is a directed graph with n nodes. If every node of G is in its own strongly connected component (that is, there are n different SCCs, one per node), what is the largest number of edges that G can contain?



(a) The Petersen graph. (b) A 15-node cycle. (c) A 16-node cycle. (d) A 15-node path. (e) A 16-node path.

Figure 11.58 Five graphs.

11.107 A *metric* on a set V is a function $d : V \times V \to \mathbb{R}^{\geq 0}$ that obeys the following conditions (see Exercise 4.6):

- *reflexivity*: for any $u \in V$ and $v \in V$, we have $d(u,u) = 0$ and $d(u,v) \neq 0$ whenever $u \neq v$.
- *symmetry*: for any $u \in V$ and $v \in V$, we have $d(u,v) = d(v,u)$.
- *triangle inequality*: for any $u \in V$ and $v \in V$ and $z \in V$, we have $d(u,v) \leq d(u,z) + d(z,v)$.

Let $d_G(u,v)$ denote the distance (shortest path length) between nodes $u \in V$ and $v \in V$ for a graph $G = \langle V, E \rangle$. Prove that d_G is a metric if G is any connected undirected graph.

11.108 Prove that d_G is not necessarily a metric for a directed graph G, even if G is strongly connected. (See Exercise 11.107.)

11.109 Definition 11.25 defined a strong connected component in a graph $G = \langle V, E \rangle$ as *a set $C \subseteq V$ such that: (i) any two nodes $s \in C$ and $t \in C$ are strongly connected; and (ii) for any node $x \in V - C$, adding x to C would make (i) false.* Suppose that we'd instead defined clause (i) as *for any two nodes $s \in C$ and $t \in C$, the node t is reachable from node s.* (But we don't require that s be reachable from t.) This alternate definition is equivalent to the original. Why?

11.110 Prove that the strongly connected components (SCCs) of a directed graph partition the nodes of the graph: that is, prove that the relation $R(u,v)$ denoting mutual reachability (u is reachable from v, and v is reachable from u) is an equivalence relation (reflexive, symmetric, and transitive). (See Definition 8.13.)

11.111 Identify the strongly connected components in the graph in Figure 11.59a.

11.112 Identify the strongly connected components in the graph represented by the adjacency list in Figure 11.59b.

11.113 Suppose that we run breadth-first search from node `A` in the graph in Figure 11.59a. What is the *last* node that BFS discovers? (If there's a tie, then list all the tied nodes.)

11.114 What if we run BFS from node `B` in the graph in Figure 11.59a?

11.115 What if we run BFS from node `0` in the graph in Figure 11.59b?

11.116 What if we run BFS from node `12` in the graph in Figure 11.59b?

11.117 Breadth-first search as described in Figure 11.46b finds all nodes reachable from a given source node in a given graph, and, in fact, it discovers nodes in increasing order of their distance from s. But we didn't actually record distances during the computation. Modify the pseudocode for BFS to compute distances instead of just whether a path exists, by annotating every node added to *Frontier* with its distance from the source node s.

11.118 Argue that in your modified version of BFS, there are never more than two different distances stored in the Frontier.

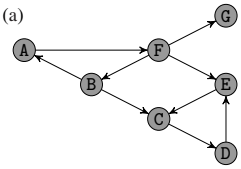
11.119 Argue that the claim from Exercise 11.118 may be false for *depth*-first search.

11.120 Consider a graph G represented by an adjacency matrix M. What does the $\langle i,j \rangle$th entry of MM (the matrix that results from squaring the matrix M) represent?

11.121 *(programming required.)* A *word ladder* is a sequence $\langle w_1, w_2, \ldots, w_k \rangle$ of words, where each w_i is a word in English, and w_{i+1} is one letter different from w_i. For example, a word ladder from `FROWN` to `SMILE` for my dictionary is

`FROWN` → `FLOWN` → `FLOWS` → `SLOWS` → `SLOTS` → `SLITS` → `SKITS` → `SKITE` → `SMITE` → `SMILE`.

(a)



(b)

```
0:   3, 7
1:   9, 2, 5
2:   1, 10, 9
3:   0, 7, 1
4:   10, 7
5:   1
6:   7, 11
```

(continued)

```
7:   0, 4, 6, 8
8:   11, 12
9:   1
10:  2, 4
11:  6, 8
12:  8
```

Figure 11.59 Two graphs.

(`SKITE` is an obscure word of Scottish origin, meaning "an oblique blow.") Write a program that uses a BFS-like algorithm to find a shortest word ladder between two given words w_1 and w_2 of the same length. (You can find a dictionary of English words on the web, or `/usr/share/dict/words` on Unix-based operating systems. You'll want to cull your dictionary to only words of the right length before you start.) There are faster solutions that involve searching "in both directions" out from w_1 and into w_2 until you find a match, but BFS from w_1 will work.

11.4 Trees

> I think that I shall never see
> A poem lovely as a tree.
>
> Joyce Kilmer (1886–1918)
> "Trees," *Trees and Other Poems* (1914)

Informally, a *tree* is a graph that grows from a *root*, branching outward and eventually leading to the *leaves*. (We computer scientists are always upside down compared to botanists: unlike an oak or maple or tamarack, the root of a tree in CS is at the top, and it grows downward toward the leaves.) See Figure 11.60 for a small example.

Trees arise very frequently in computer science: to name just a few examples, they're the class hierarchies of object-oriented programming, the binary search trees of data structures (see p. 11-73), the game trees describing the progression of Tic-Tac-Toe or chess (p. 3-54), the parse trees that describe formal or natural languages (p. 5-56), the recursion trees that describe the execution of recursive algorithms (Section 6.4). Trees are also frequently used in computational models of important phenomena from outside of CS: for example, in reconstructing evolutionary phylogenies (in computational biology), or in reconstructing the paths by which rumors and misinformation spread from the originator (in social network analysis). In this section, we'll introduce trees formally—including definitions, properties, algorithms, and applications—as a special type of graph.

11.4.1 Cycles

Before we can define trees properly, we must first define another notion about graphs in general—a *cycle*, which is way to get from a node back to itself:

Definition 11.30: Cycle.
A *cycle* $\langle u_1, u_2, \ldots, u_k, u_1 \rangle$ is a path of length ≥ 2 from a node u_1 back to node u_1 that does not traverse the same edge twice. Just as for any other path, the *length* of the cycle $\langle u_1, u_2, \ldots, u_k, u_1 \rangle$ is the number of edges it traverses—that is, k.

Figures 11.61a and 11.61b show examples of an undirected and directed graph with a cycle $\langle \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{A} \rangle$.

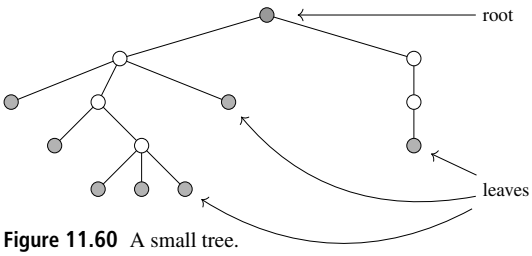
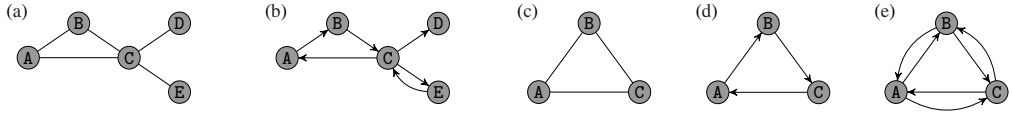


Figure 11.60 A small tree.



An undirected graph (a) and a directed graph (b) that contain the cycle $\langle \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{A} \rangle$.

Undirected (c) and directed (d) graphs containing only one cycle, and a directed graph (e) containing two.

Figure 11.61 A few graphs with cycles.

Note that the edges $\langle s, t \rangle$ and $\langle t, s \rangle$ in a directed graph are different; in an undirected graph, the edges $\{s, t\}$ and $\{t, s\}$ are the same. Thus a cycle in a directed graph *can* use both $\langle s, t \rangle$ and $\langle t, s \rangle$, but a cycle in an undirected graph cannot use both $\langle s, t \rangle$ and $\langle t, s \rangle$. In the directed graph in Figure 11.61b, the path $\langle \mathtt{C}, \mathtt{E}, \mathtt{C} \rangle$ *is* a cycle, but $\langle \mathtt{C}, \mathtt{E}, \mathtt{C} \rangle$ *is not* a cycle in the undirected graph in Figure 11.61a because it reuses an edge.

Technically, the definition of a cycle in Definition 11.30 says that the undirected graph in Figure 11.61a has six different cycles: $\langle \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{A} \rangle$, $\langle \mathtt{C}, \mathtt{A}, \mathtt{B}, \mathtt{C} \rangle$, and $\langle \mathtt{B}, \mathtt{C}, \mathtt{A}, \mathtt{B} \rangle$ (going clockwise); and $\langle \mathtt{A}, \mathtt{C}, \mathtt{B}, \mathtt{A} \rangle$, $\langle \mathtt{C}, \mathtt{B}, \mathtt{A}, \mathtt{C} \rangle$, and $\langle \mathtt{B}, \mathtt{A}, \mathtt{C}, \mathtt{B} \rangle$ (going counterclockwise). However, we will adopt the convention that there is one and only one cycle in this graph. Because we can "start anywhere" in a cycle, we consider a cycle to be defined only by the relative ordering of the nodes involved, regardless of where we start. In an undirected graph, we can "go either direction" (clockwise or counterclockwise), so we also ignore the direction of travel in distinguishing cycles. In a directed graph, the direction of travel *does* matter; there are graphs in which we are able to go in one direction around a cycle without being able to go in the other. In other words, we say that Figure 11.61c and Figure 11.61d have one cycle each, while Figure 11.61e has two.

A cycle is by definition forbidden from traversing the same edge twice. A *simple* cycle also does not visit any *node* more than once:

Definition 11.31: Simple cycle.
A cycle $\langle u_1, u_2, \ldots, u_k, u_1 \rangle$ is *simple* if each u_i is distinct—that is, no nodes in the cycle are duplicated aside from the last node (which equals the first node).

(We've now used the word "simple" in three different contexts: simple graphs have no parallel edges or self-loops, and simple paths and cycles have no repeated vertices. Intuitively, all three definitions correspond to an entity that's not unnecessarily complicated.) Here are a few examples:

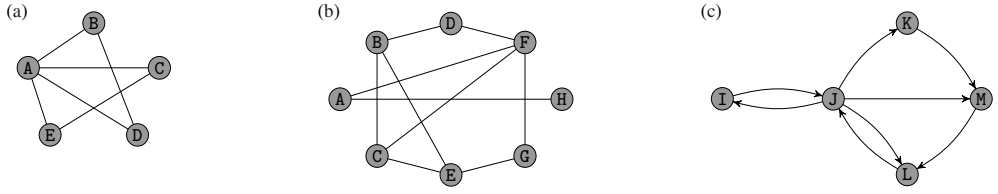


Figure 11.62 A few graphs that contain cycles.

Example 11.30: Simple and non-simple ways of getting around.
Figure 11.62a shows an undirected graph. In it, $\langle \mathtt{D}, \mathtt{B}, \mathtt{A}, \mathtt{C}, \mathtt{E}, \mathtt{A}, \mathtt{D} \rangle$ is a non-simple cycle. This graph also has two simple cycles: $\langle \mathtt{D}, \mathtt{B}, \mathtt{A}, \mathtt{D} \rangle$ and $\langle \mathtt{C}, \mathtt{E}, \mathtt{A}, \mathtt{C} \rangle$.

Example 11.31: Finding cycles.
Identify all simple cycles in the graphs in Figures 11.62b and 11.62c.

Solution. A nice way to identify cycles systematically is to look for cycles of all possible lengths: 2-node cycles, 3-node cycles, etc. (Actually 2-node cycles are possible only in directed graphs. *Exercise: why?*) Ordered by length, the simple cycles in the graph in Figure 11.62b are:

$$\langle \mathtt{B}, \mathtt{E}, \mathtt{C}, \mathtt{B} \rangle, \quad \langle \mathtt{B}, \mathtt{D}, \mathtt{F}, \mathtt{C}, \mathtt{B} \rangle, \quad \langle \mathtt{C}, \mathtt{F}, \mathtt{G}, \mathtt{E}, \mathtt{C} \rangle, \quad \langle \mathtt{B}, \mathtt{D}, \mathtt{F}, \mathtt{G}, \mathtt{E}, \mathtt{B} \rangle, \quad \text{and} \quad \langle \mathtt{B}, \mathtt{D}, \mathtt{F}, \mathtt{G}, \mathtt{E}, \mathtt{C}, \mathtt{B} \rangle.$$

The simple cycles in the graph in Figure 11.62c are:

$$\langle \mathtt{I}, \mathtt{J}, \mathtt{I} \rangle, \quad \langle \mathtt{J}, \mathtt{L}, \mathtt{J} \rangle, \quad \langle \mathtt{J}, \mathtt{M}, \mathtt{L}, \mathtt{J} \rangle, \quad \text{and} \quad \langle \mathtt{J}, \mathtt{K}, \mathtt{M}, \mathtt{L}, \mathtt{J} \rangle.$$

Note that (to name one of several examples) the sequence $\langle \mathtt{I}, \mathtt{J}, \mathtt{L}, \mathtt{J}, \mathtt{I} \rangle$ is also a cycle in Figure 11.62c—it traverses four distinct directed edges and goes from node `I` to `I`—but this cycle is not simple, because node `J` is repeated.

We can use a modification of breadth-first search to identify cycles algorithmically. Specifically, suppose that we wish to find out whether a node u is involved in a cycle in a directed graph. We run BFS starting at node u, and if we ever encounter a node v that has u as a neighbor, then we have found a cycle involving node u. (An extra modification is necessary for undirected graphs; see Exercise 11.129.)

Taking it further: Kidneys are the most frequently transplanted organ today, in part because—unlike for other organs—humans generally have a "spare": we're born with two kidneys, but only need one functioning kidney to live a healthy life. Thus patients suffering from kidney failure may be able to get a transplant from friends or family members who are willing to donate one of their kidneys. But this potential transplant relies on the donor and the patient being compatible in dimensions like blood type and the physical size of the organs. Recently a computational solution to the problem of incompatibility has emerged, using algorithms based on finding (short) cycles in a particular graph: there is now national exchange for matching up two (or a few) patients with willing-but-incompatible donors, and doing a multiway transplant. See p. 11-71.

Acyclic graphs

While cycles are important on their own, their relevance for trees is actually when they *don't* exist:

Definition 11.32: Acyclic graphs.
A graph is *acyclic* if it contains no cycles.

Let's prove a useful structural fact about acyclic graphs. (Recall that we are considering *finite* graphs, where the set of nodes in the graph is finite. The following claim would be false if graphs could have an infinite number of nodes!)

Lemma 11.33: Every acyclic graph has a node with degree 0 or 1.
If $G = \langle V, E \rangle$ is an acyclic undirected graph, then there exists a node in V whose degree is zero or one.

Proof. We'll give a constructive proof of the claim—specifically, we'll give an algorithm that finds a node with the stated property:

```
1 let u_0 be an arbitrary node in the graph, and let i := 0
2 while the current node u_i has no unvisited neighbors:
3     let u_{i+1} be a neighbor of u_i that has not previously been visited
4     increment i
```

Observe that this process must terminate in at most $|V|$ iterations, because we must visit a new node in each step. Suppose that this algorithm goes through k iterations of the **while** loop, and let t be the last node visited by the algorithm. (So $t = u_k$.)

If $k = 0$, then $t = u_0$ has degree zero, so the claim follows immediately. Otherwise $k \geq 1$; in this case, we'll argue that t has degree one. Because the algorithm terminated, there cannot be an edge between t and any unvisited node. Furthermore, if there were an edge from t to any previously visited node u_j for $j < k - 1$, then there would be a cycle in the graph, namely $\langle u_j, u_{j+1}, \ldots, u_{k-1}, u_k, u_j \rangle$. Therefore t's only neighbor is u_{k-1}, and the degree of t is one. □

For directed graphs, the claim analogous to Lemma 11.33 is *every directed acyclic graph contains a node with out-degree zero.* (You'll prove it in Exercise 11.130.)

> **Taking it further:** A *directed acyclic graph* (often just called a *DAG*) is, perhaps obviously, a directed graph that contains no cycles. A DAG G corresponds to a (strict) partial order (see Chapter 8); a cycle in G corresponds to a violation of transitivity. In fact, we can think of *any* directed graph $G = \langle V, E \rangle$ as a relation—specifically, the edge set E is a subset of $V \times V$. Like transitivity and acyclicity, many of the concepts that we explored in Chapter 8 have analogues in the world of graphs.

11.4.2 Trees

With the definition of cycles in hand, we can now define trees themselves:

Definition 11.34: Tree.
A *tree* is an undirected graph that is connected and acyclic.

We will also sometimes talk about graphs that satisfy only the latter requirement: a *forest* is an undirected graph that is acyclic (but not necessarily connected). Every connected component of a forest is a tree, and note that a tree is itself a forest. (An irrelevant note about Chinese: the character for *tree* is 木; the character for *forest* is 森—a disconnected collection of trees!)

Example 11.32: A small collection of trees.
Several examples of trees are shown in Figure 11.63: all six graphs have a single connected component and contain no cycles. Therefore all six are trees.

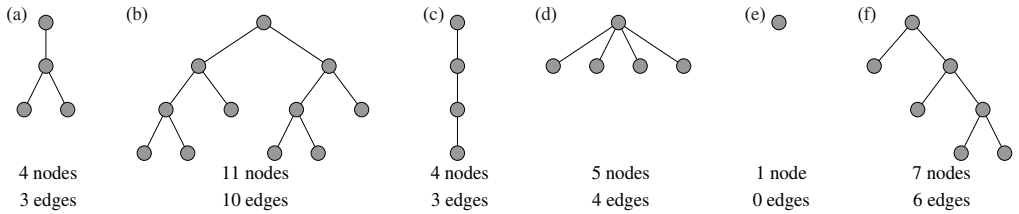


Figure 11.63 Six sample trees, each annotated with the number of nodes and edges that it contains.

We'll prove several structural facts about trees in this section, beginning with one concerning the number of edges in a tree. To start, let's look at the trees in Figure 11.63. The figure shows the number of nodes and edges in each of these trees. In each, the number of nodes is one more than the number of edges, and that's no coincidence; here's the statement and proof of the general fact:

Theorem 11.35: Number of edges in a tree.
Let $T = \langle V, E \rangle$ be a tree. Then $|E| = |V| - 1$.

Proof. Let $P(n)$ denote the property that any n-node tree has precisely $n - 1$ edges. We will prove that $P(n)$ holds for all $n \geq 1$ by induction on n.

Base case ($n = 1$). We must prove $P(1)$: any 1-node tree has $1 - 1 = 0$ edges. But the only (simple) 1-node graph is the one shown in Figure 11.63e, which has zero edges, and so we're done immediately.

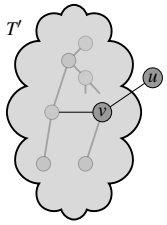
Inductive case ($n \geq 2$). We assume the inductive hypothesis $P(n-1)$—that is, every $(n-1)$-node tree has $n - 2$ edges. We must prove $P(n)$.

Consider an arbitrary tree $T = \langle V, E \rangle$ with $|V| = n$. By definition, T is acyclic and connected. By Lemma 11.33, then, there exists a node $u \in V$ with degree 0 or 1 in T. Furthermore, because T is connected, the degree of u cannot be 0. Thus u is a node with $degree(u) = 1$. Let $v \in V$ be the unique neighbor of u in T. Let T' be T with node u and the edge $\{u, v\}$ between u and v deleted. (See Figure 11.64a.) We claim that the graph $T' = \langle V - \{u\}, E - \{\{u, v\}\} \rangle$ is a tree, too. The acyclicity and connectivity of T' both follow from the fact that T was acyclic and connected, and the fact that the eliminated node u was of degree 1.

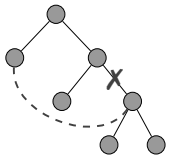
The tree T' contains $n - 1$ nodes, and thus, by the inductive hypothesis $P(n-1)$, contains $n - 2$ edges. Therefore T, whose edges are precisely the edges of T' plus the eliminated edge $\{u, v\}$, contains precisely $(n - 2) + 1 = n - 1$ edges. □

An immediate consequence of Theorem 11.35 is that every tree is teetering on the edge of being disconnected and of having a cycle (see Figure 11.64):

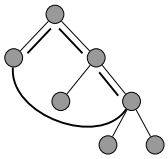
Corollary 11.36. Let $T = \langle V, E \rangle$ be any tree. Then (1) adding any edge $e \notin E$ to T creates a cycle; and (2) removing any edge $e \in E$ from T disconnects the graph.



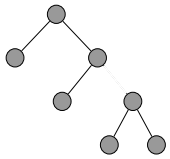
(a) A tree T, with a node u of degree $= 1$ and its neighbor v. The tree T' is T without the node u and the edge $\{u, v\}$.



(b) Imagine adding the dashed edge, or removing the edge marked with ✗.



(c) Adding an edge creates a cycle.



(d) Removing an edge disconnects the graph.

Figure 11.64 Illustrations of (a) the proof of Theorem 11.35, and of (b–d) the statement of Corollary 11.36.

Proof. For (1), define the graph $G = \langle V, E \cup \{e\} \rangle$ as the result of adding the new edge e to the tree T. Because adding an edge to a graph can never disrupt connectivity and T was already connected, we know that G must be connected too. Thus if G were acyclic, then G would be a tree. But G has one more edge than T—specifically, G has $(|V| - 1) + 1 = |V|$ edges—and therefore isn't a tree by Theorem 11.35.

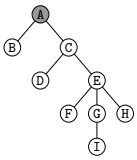
The proof for (2) is similar: let G' be T with e removed. Removing an edge cannot create a cycle, so G' is acyclic. But G' has too few edges to be a tree by Theorem 11.35, so G' must be disconnected. □

(Here's an alternative proof of the first statement in Corollary 11.36. Let $\langle u, v \rangle$ be an edge not in the tree T. Because T is connected, there is already a (simple) path P from u to v in T. If we add $\langle u, v \rangle$ to T, then there is a cycle: follow P from u to v and then follow the new edge from v back to u. Therefore G contains a cycle.)

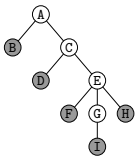
Rooted trees

We often designate a particular node of a tree T as the *root*, which is traditionally drawn as the topmost node. (Note that we could designate any node as the root and—just like that mobile of zoo animals from your crib from infancy—"hang" the tree by that node.) We will adopt the standard convention that, whenever we draw trees, the vertically highest node is the root.

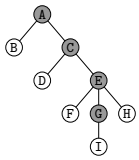
There's a lot of terminology about rooted trees in computer science that's borrowed from the world of family trees:



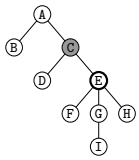
(a) Root



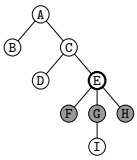
(b) Leaves



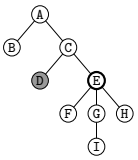
(c) Internal nodes



(d) Ⓔ's parent



(e) Ⓔ's children



(f) Ⓔ's sibling(s)

Figure 11.65 The root, leaves, and internal nodes of the tree; the parent, children, and siblings of one node.

- For a node u in a tree with root $r \neq u$, the *parent* of u is the unique neighbor of u that is closer to r than u is. (The root is the only node that has no parent.)
- A node v is one of the *children* of a node u if v's parent is u.
- A node v is a *sibling* of a node $u \neq v$ if v and u have the same parent.

A node with zero children is called a *leaf*. A node with one or more children is called an *internal node*. (Note that the root is an internal node unless the tree is just a one-node graph.)

Example 11.33: Some sample trees.
See Figure 11.65 for an illustration of all of these definitions. Note that Figure 11.65 is correct only when the root is the topmost node in the image; with a different root, all of the panels could change.

Here's a concrete example. Consider the two trees in Figure 11.66a and 11.66b, which differ only based on which node is designated as the root. A few properties of each tree are shown, too: the root, leaves, and internal nodes, along with the parent and children of nodes `A` and `B`. While the leaves and internal nodes are identical in these two trees, if we'd rerooted the tree at any of the erstwhile leaves instead, the new root would become an internal node instead of a leaf. For example, if we reroot this tree at `H`, then the leaves would be $\{\texttt{D}, \texttt{F}, \texttt{J}, \texttt{K}, \texttt{L}, \texttt{M}\}$ and the internal nodes would be $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{E}, \texttt{G}, \texttt{H}, \texttt{I}\}$.

Subtrees, descendants, and ancestors

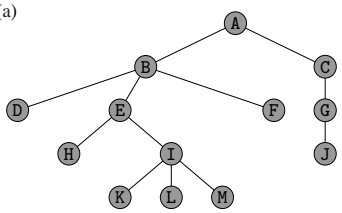
Let T be a rooted tree, and let u be any node in T. The *subtree rooted at* u consists of u and all those nodes and edges "below" u in T. (In other words, a node v is in the subtree rooted at u if and only if v is no closer to the root of T than u is; the subtree is the induced subgraph of these nodes.) Such a node v in the subtree rooted at u is called a *descendant of* u if $v \neq u$. The node u is called an *ancestor of* v.

Example 11.34: Descendants, ancestors, and subtrees.
See Figure 11.67a–11.67c for illustrations of these definitions. For another example, the trees from Figure 11.66a and Figure 11.66b are reproduced in Figure 11.68, with a few properties marked (and the subtrees rooted at a particular node highlighted).

We have one final pair of definitions to (at last!) conclude our parade of terminology about rooted trees, related to how "tall" a tree is. Consider a rooted tree T with root node r. The *depth* of a node u is the distance from u to r. The *height* of a tree is the maximum, over all nodes u in the tree, of the depth of node u.

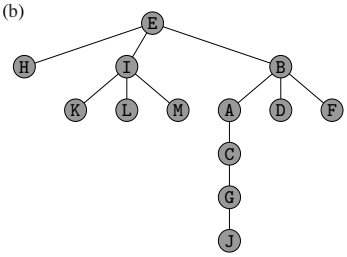
Example 11.35: Depth of nodes in a tree.
Every node in the tree in Figure 11.67d is labeled by its depth: the root has depth 0, its children have depth 1, their children (the "grandchildren" of the root) have depth 2, and so forth. The height of the tree is the largest depth of any of its nodes—in this case, the height is 4.

(a)



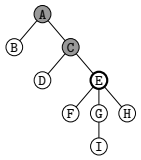
Root:	A
Leaves:	{D, F, H, J, K, L, M}
Internal nodes:	{A, B, C, E, G, I}
Parent of B:	A
Children of B:	{D, E, F}
Parent of A:	none
Children of A:	{B, C}

(b)

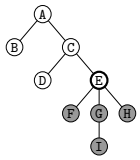


Root:	E
Leaves:	{D, F, H, J, K, L, M}
Internal nodes:	{A, B, C, E, G, I}
Parent of B:	E
Children of B:	{A, D, F}
Parent of A:	B
Children of A:	{C}

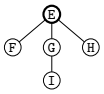
Figure 11.66 Two rooted trees. (The second tree is just the first, rerooted to make E the new root.)



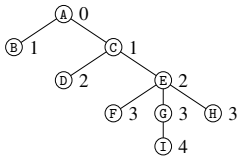
(a) Ancestors of E.



(b) Descendants of E.



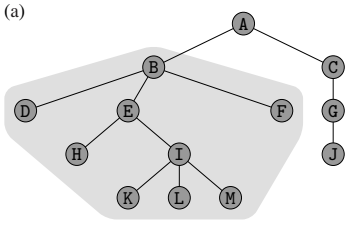
(c) Subtree rooted at E.



(d) Every node's depth.

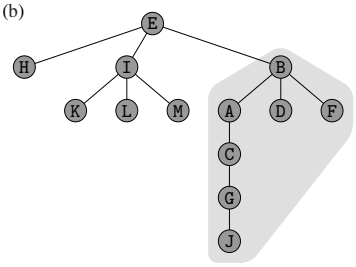
Figure 11.67 Ancestors, descendants, subtrees, and the depth of nodes in a rooted tree.

(a)



Descendants of B:	{D, E, F, H, I, K, L, M}
Ancestors of B:	{A}
Descendants of H:	none
Ancestors of H:	{A, B, E}

(b)



Descendants of B:	{A, C, D, F, G, J}
Ancestors of B:	{E}
Descendants of H:	none
Ancestors of H:	{E}

Figure 11.68 Two rooted trees (from Figure 11.66), with the subtrees rooted at B highlighted in both.

Taking it further: If you happen to be a lover of recursion, you may find it interesting to think about alternative recursive ways to give the definitions about rooted trees from this section. (See Section 5.4 for some similar ideas.) Ancestors and descendants in a rooted tree are a good example. We could say that a node v is an *ancestor* of u if (i) v is the parent of u; or (ii) v is the parent of any ancestor of u. Or a node v is a *descendant* of u if (i) v is a child of u; or (ii) v is a child of any descendant of u. We can also think of the depth of a node, or the height of a tree, recursively. The depth of the root is zero; the depth of a node with a parent p is $1 +$ (the depth of p). Or, for height: (i) the height of a one-node tree T is zero; and (ii) the height of a tree T with root r with children $\{c_1, c_2, \ldots, c_k\}$ is $1 + \max_i$(the height of the subtree rooted at c_i). These alternative versions of the definitions can be translated—almost word for word—into recursive code to compute these tree-related quantities.

Binary trees

We'll often encounter a special type of tree in which nodes have a limited number of children:

Definition 11.37: Binary trees and *k*-ary trees.
A *binary tree* is a rooted tree in which each node has 0, 1, or 2 children. More generally, if every node in a rooted tree T has k or fewer children, then T is called a *k-ary* tree. (In other words, a binary tree is 2-ary.)

Example 11.36: Some binary trees.
Of the trees in Figure 11.69, only the tree in Figure 11.69d is not a binary tree, because its root has four children. (This tree is a 4-ary tree.) But the other five trees are all binary: in each, every internal node has either 1 child or 2 children.

In a binary tree, the possible children of a node are called its *left child* and *right child*. (Even for a node u in a binary tree that has only one child, we'll insist that the lone child be designated as either the left child of u or the right child of u.) For a node u, we say that u's *left subtree* is the subtree rooted at u's left child; the *right subtree* is analogous. (See Section 5.4.1.)

11.4.3 Tree Traversal

We will sometimes want to do something with all of the nodes contained in a tree T: print them all out, make a list of them, count the number whose names are anagrams of swear words, whatever. There are three standard algorithms that are used for this purpose, called *pre-order*, *in-order*, and *post-order traversal*. While these algorithms can be generalized to non-binary trees, they're easier to understand for binary trees (and they're most frequently deployed for binary trees), so we'll consider them that way.

All three algorithms are recursive, and all three algorithms execute precisely the same steps—just in a different order. On an empty tree T, we do nothing; on a non-empty tree T, all three algorithms perform the following steps:

- we "visit" the root of T. (You can think of "visiting" the root as printing out the contents of the root node, or as adding it to the end of an accumulating list of the nodes that we've encountered in the tree.)

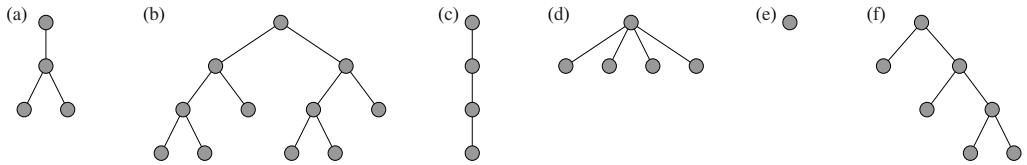
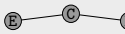


Figure 11.69 The trees from Figure 11.63, repeated.

- we recursively traverse the left subtree of T, finding all nodes there.
- we recursively traverse the right subtree of T, finding all nodes there.

But the three traversal algorithms execute the three steps in different orders, either visiting the root *before both recursive calls* ("pre-order"); *between the recursive calls* ("in-order"); or *after both recursive calls* ("post-order"). We always recurse on the left subtree before we recurse on the right subtree. See Figure 11.70. Let's take a look at traversing a small tree using these algorithms:

Example 11.37: Traversing a small tree: pre-order traversal.
Let's determine the order of nodes' visits by a pre-order traversal of the tree in Figure 11.71. In a pre-order traversal, we first visit the root, then pre-order-traverse the left subtree, then pre-order-traverse the right subtree. In other words, for this tree, we first visit the root A, then pre-order-traverse  , then pre-order-traverse  :

Step #1: visit the root. We visit the root A.

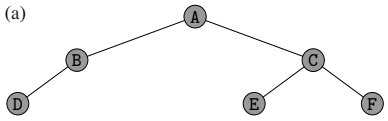
Step #2: pre-order-traverse the left subtree. To pre-order-traverse  , we first visit the root B, then pre-order-traverse the left subtree  , then pre-order-traverse the (empty) right-subtree. In order, these steps visit B and D.

pre-order-traverse(T):
```
1 if T is empty then
2     do nothing.
3 else
4     visit the root of T
5     pre-order-traverse(T_left)
6     pre-order-traverse(T_right)
```

in-order-traverse(T):
```
1 if T is empty then
2     do nothing.
3 else
4     in-order-traverse(T_left)
5     visit the root of T
6     in-order-traverse(T_right)
```

post-order-traverse(T):
```
1 if T is empty then
2     do nothing.
3 else
4     post-order-traverse(T_left)
5     post-order-traverse(T_right)
6     visit the root of T
```

Figure 11.70 Three different algorithms to traverse a binary tree. Write T_{left} to denote T's left subtree, and T_{right} to denote its right subtree.



(b)

pre-order traversal	A, B, D, C, E, F
in-order traversal	D, B, A, E, C, F
post-order traversal	D, B, E, F, C, A

Figure 11.71 A tree to be traversed, and its three traversals.

Step #3: pre-order-traverse the right subtree. To pre-order-traverse Ⓔ—Ⓒ—Ⓕ, we first visit C, then pre-order-traverse the left subtree Ⓔ, and then pre-order-traverse the right subtree Ⓕ. Pre-order-traversing Ⓔ just results in visiting E, and pre-order-traversing Ⓕ just visits F. In order, these steps visit C, E, and F.

Putting this all together, the pre-order traversal of the tree visits the nodes in this order:

$$\underbrace{\texttt{A}}_{\text{step \#1}}, \quad \underbrace{\texttt{B, D}}_{\text{step \#2}}, \quad \underbrace{\texttt{C, E, F}}_{\text{step \#3}}.$$

Here are examples of the other two traversal algorithms, on the same tree:

Example 11.38: Traversing a small tree: in-order traversal.
In what order are the nodes visited by an *in-order* traversal of the tree in Figure 11.71?

Solution. We first traverse Ⓓ—Ⓑ, then visit A, then traverse Ⓔ—Ⓒ—Ⓕ.

Traversing Ⓓ—Ⓑ visits D and B: first the left subtree, then the root.

Traversing Ⓔ—Ⓒ—Ⓕ visits E, then C, then F.

Thus an in-order traversal visits the nodes in the order D, B, A, E, C, F.

Example 11.39: Traversing a small tree: post-order traversal.
In what order are the nodes visited by an *post-order* traversal of the tree in Figure 11.71?

Solution. For a post-order traversal, the root of each subtree is the *last* node traversed in that subtree: we first traverse Ⓓ—Ⓑ, then traverse Ⓔ—Ⓒ—Ⓕ, then visit A.

Traversing Ⓓ—Ⓑ visits D and B: first the left subtree, then the nonexistent right subtree, then the root.

Traversing Ⓔ—Ⓒ—Ⓕ visits E, then F, then C.

Thus a post-order traversal visits the tree's nodes in the order D, B, E, F, C, A.

Here's another example, of using traversals to reconstruct a binary tree:

Example 11.40: Trees from traversals.
Here is the output of the pre-, in-, and post-order traversals of a binary tree T:

$$\underbrace{9, 2, 7, 4, 5, 3}_{\text{pre-order traversal}}, \quad \underbrace{2, 9, 5, 4, 3, 7}_{\text{in-order traversal}}, \quad \text{and} \quad \underbrace{2, 5, 3, 4, 7, 9}_{\text{post-order traversal}}.$$

What's T?

Solution. We'll reassemble T from the root down. The root is first in the pre-order traversal (and last in the post-order), so 9 is the root. The root separates the left subtree from the right subtree in the in-order traversal; thus the left subtree contains just 2 and the right contains $\{3, 4, 5, 7\}$. So the tree has the form shown in Figure 11.72a.

Figure 11.72 Reconstructing a tree from its pre-order, in-order, and post-order traversals.

The post-order 5, 3, 4, 7 and in-order 5, 4, 3, 7 show that 7 is the root of the unknown portion of the tree and that 7's right subtree is empty. The last three nodes are pre-ordered 4, 5, 3; in-ordered 5, 4, 3; and post-ordered 5, 3, 4. In sum, that says that 4 is the root, 5 is the left subtree, and 3 is the right subtree. Assembling these pieces yields the final tree, shown in Figure 11.72b.

Taking it further: One particularly important type of binary tree is the *binary search tree (BST)*, a widely used data structure—one that's probably very familiar if you've taken a course on data structures. A BST is a binary tree in which each node has some associated "key" (a piece of data), and the nodes of the tree are stored in a particular sorted order: all nodes in the left subtree have a key smaller than the root, and all nodes in the right subtree have a key larger than the root. Thus an in-order traversal of a binary search tree yields the tree's keys in sorted order. (See p. 11-73.) An even more specific form of binary search tree, called a *balanced binary search tree,* adds an additional structural property related to the depth of nodes in the tree. See p. 6-53 for a discussion of one scheme for balanced binary search trees, called *AVL trees.*

11.4.4 Spanning Trees

Let $G = \langle V, E \rangle$ be an undirected graph. For example, imagine that each node in V represents a dorm room on your campus, and each edge in E denotes a possible fiber optic cable that can be laid to build an ethernet connection throughout the residence halls. A reasonable goal is to actually place only some of those possible cables, a subset $E' \subseteq E$, while ensuring that network traffic can be sent between any two dorm rooms—that is, ensuring that the resulting network is connected. In other words, one seeks a *spanning tree* of the graph G:

Definition 11.38: Spanning tree.
Let $G = \langle V, E \rangle$ be a connected undirected graph. A *spanning tree* of G is a tree $T = \langle V, E' \rangle$ with the same nodes as G and with edges $E' \subseteq E$ that are a subset of G's edges.

A spanning tree of G is called "spanning" because it connects (that is, spans) all nodes in G.

Example 11.41: A complete collection of spanning trees.
Figure 11.73 shows a small example: a small graph, and all of the different spanning trees for it.

A graph G has a spanning tree if and only if G is connected: we can be sure to only remove "redundant" edges that aren't required for connectivity, and removing edges from G can never cause a disconnected graph to become connected. (For disconnected graphs, people sometimes talk about a *spanning forest:* a forest $F = \langle V, E' \rangle$ with $E' \subseteq E$, where the connected components of the original graph G and the connected components of the forest F are identical.)

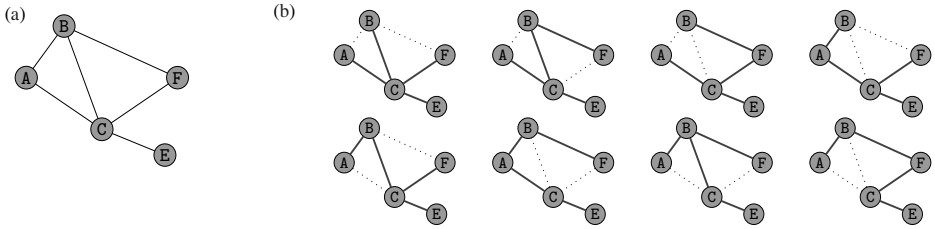


Figure 11.73 (a) An undirected graph, and (b) all 8 spanning trees for it.

Although we didn't talk about it this way when we introduced breadth- and depth-first search (see Figures 11.46b and 11.49b), these algorithms can find spanning trees, with a small change: as we explore the graph, we include in E' every edge $\langle u, v\rangle$ that leads from a previously known node u to a newly discovered node v.

We'll also see some other ways to find spanning trees in Section 11.5.2, but here's another, conceptually simpler technique. To find a spanning tree in a connected graph G, we repeatedly find an edge that can be deleted without disconnecting G—that is, an edge that's in a cycle—and delete it. See Figure 11.74a for the algorithm. Here's an example:

Example 11.42: Finding a spanning tree via cycle elimination.
Figure 11.74b shows the Cycle Elimination algorithm computing a spanning tree of a particular graph. We repeatedly do this: choose some cycle (it doesn't matter which cycle), and then remove a edge from that cycle (it doesn't matter which edge). For this particular graph, after three iterations, the resulting graph has no cycles, and remains connected; the resulting graph is a spanning tree of the original graph.

We'll finish the section by proving that the Cycle Elimination algorithm correctly finds the spanning tree of an arbitrary connected graph:

Theorem 11.39: Correctness of the Cycle Elimination algorithm.
Given any connected graph $G = \langle V, E\rangle$, the Cycle Elimination algorithm returns a spanning tree T of G.

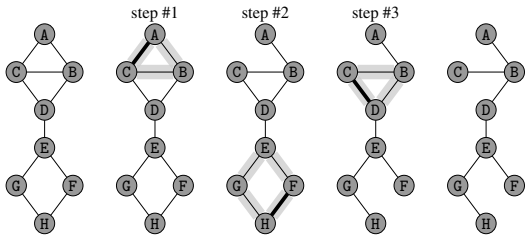
Proof. The algorithm only deletes edges from G, so certainly $T = \langle V, E'\rangle$ satisfies $E' \subseteq E$. We need to prove that T is a tree: that is, T is acyclic and T is connected.

T is acyclic. As long as there's a cycle remaining, the algorithm stays in the **while** loop. Thus we only exit the loop when the remaining graph is acyclic. (And the loop terminates in at most $|E|$ iterations, because an edge is deleted in every iteration.)

T is connected. We claim that the graph is connected throughout the algorithm. It's true at the beginning of the algorithm, by assumption. Now consider an iteration in which we delete the edge $\{u, v\}$ from a cycle C. Let s and t be arbitrary nodes; we will argue that there is still a path from s to t. Before we deleted $\{u, v\}$, there was a path P from s to t. If P didn't traverse the edge $\{u, v\}$, then P is still a path from s to t.

Cycle Elimination Algorithm:
Input: a connected graph $G = \langle V, E \rangle$
Output: a spanning tree of G
1 **while** there exists a cycle C in G:
2 let e be an arbitrary edge traversed by C
3 remove e from E
4 **return** the resulting graph $\langle V, E \rangle$.

(a) An algorithm to find a spanning tree.



(b) An example. In each iteration, we've selected an arbitrary cycle (lightly shaded) and then selected an arbitrary edge from that cycle (heavily shaded) and removed it.

Figure 11.74 The Cycle Elimination Algorithm, and an example of its execution.

Figure 11.75 Maintaining connectivity in the Cycle Elimination Algorithm. There are two ways from s to t: (a) the short way, via $\{u, v\}$; and (b) the long way, all the way around C.

Otherwise, we can still get from s to t by going "the long way around" the cycle C instead of following the single edge $\{u, v\}$. (See Figure 11.75.) Thus there is still a path from any node s to any node t, and so the graph stays connected. □

Computer Science Connections

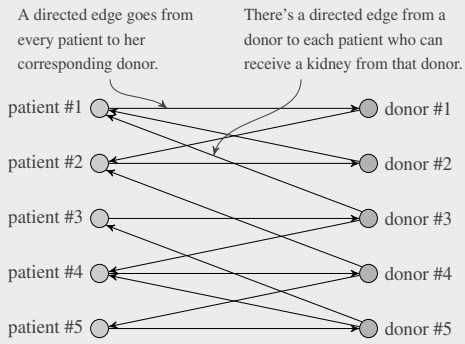
Directed Graphs, Cycles, and Kidney Transplants

Kidneys are essential to human life; they play an essential filtering role in the body without which we would all die. Although we are born with two kidneys, humans need only one functioning kidney to live healthy lives. Because we're all naturally equipped with a "spare," kidney transplants are the most common form of transplant surgery performed today; thousands of lives are saved annually through kidney transplants.

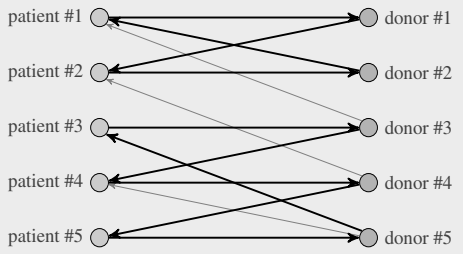
Typically a patient in need of a kidney finds a friend or relative who is willing to donate one of theirs. If the patient and donor are compatible—for example, blood type and physical size of the donor's kidney must be appropriate—then medical teams perform two simultaneous operations: one to remove the "spare" kidney from the donor, and one to implant it in the patient. (Some patients instead receive kidneys from strangers who chose to donate their organs in case of an untimely death, or from so-called "angel donors" who decide to donate one of their two working kidneys while they're alive.) Unfortunately, many patients who need kidneys have a friend or relative willing to donate to them—but they are incompatible with their prospective donor's kidney. These patients may spend years on a waiting list for a transplant, undergoing painful, expensive, and only partially effective dialysis while they wait and hope.

In recent years, medical personnel have begun a program of *kidney exchanges*. Suppose that a patient p_1 is incompatible with her prospective donor d_1, another patient p_2 is incompatible with his prospective donor d_2, but pairs $\langle p_1, d_2 \rangle$ and $\langle p_2, d_1 \rangle$ are both compatible with each other. *Four* teams of doctors can then do a "paired exchange" with four surgeries, in which d_1 donates to p_2 and d_2 donates to p_1. (To ensure that everybody follows through, the two pairs of surgeries must be done simultaneously: if d_1 donates to p_2 *before* d_2 undergoes surgery, then d_2 has no incentive to go through the surgery, as d_2's friend p_2 has already received his kidney.) We can even consider larger exchanges (three or more simultaneous donations)—though as the number of surgeries increases, the logistical difficulty increases as well.

Deciding which transplants to complete is done using an algorithm based on the graph of donor–patient compatabilities. Each patient p_i comes to the system with a donor d_i who is willing to donate to p_i. Define a directed graph G as follows. There is a node for each patient p_i and a node for each donor d_i. Add a directed edge $\langle p_i, d_i \rangle$ for every i. Also add a directed edge $\langle d_i, p_j \rangle$ if donor d_j is compatible with patient p_j. A cycle in G then corresponds to a set of surgeries that can be completed: every donor in the cycle donates a kidney, and every patient in the cycle receives a compatible kidney. See Figure 11.76 for an example.



(a) The graph of compatibilities.



(b) The selected transplants. We "cover" this graph with two cycles; if we perform the transplants that are highlighted (the darker donor-to-patient edges), then every patient receives a compatible kidney.

Figure 11.76 An example of a kidney exchange network, and the cycle-based algorithm to select transplants.

The algorithm that's actually used in the real kidney exchange network in the United States computes a *set* of node-disjoint cycles that will be performed. To limit the number of simultaneous surgeries that are required, the algorithm seeks a set of *cycles of length* 4 *or length* 6—that is, 2 or 3 transplants—in G that maximizes the total number of nodes included. (The constraint on cycle length makes the computational problem much more difficult, so the algorithm requires significant computational power to compute the surgeries to complete.) There's more information about "paired donation" (the usual form, in which a cycle contains exactly two donors and two recipients) from the National Kidney Foundation, `kidney.org`. You can read more about the computational research in [2].

Computer Science Connections

Binary Search Trees

Trees are the basis of many important data structures, of which *binary search trees* are perhaps most frequently used. Binary search trees are data structures that implement the abstract data type called a *dictionary*: we have a set of *keys*, each of which has a corresponding *value*. (For example, the keys might be words and the values definitions, or they might be student names and GPAs, or usernames and encrypted passwords.) The data structure must support operations like *insert*(k, v) (add a new key/value pair) and *lookup*(k) (report the value associated with key k, if any).

A *binary search tree (BST)* is a binary tree for which every node u satisfies the *BST condition* illustrated in Figure 11.77: every node v in u's left subtree has a key that is less than u's key, and every node v in u's right subtree has a key that is greater than u's key. For simplicity, assume that all keys are distinct. (Incidentally, the BST condition implies the following claim: *an in-order traversal of a binary search tree visits the keys in sorted order.* This claim can be proven formally by induction, but the intuition is straightforward: an in-order traversal of a node with key x first visits nodes $< x$ [while traversing the left subtree], then x itself, and then nodes $> x$ [while traversing the right subtree]. Because, recursively, the nodes of the left and right subtrees are themselves visited in sorted order, the entire tree's keys are visited in sorted order.) An example of a binary search tree is shown in Figure 11.78.

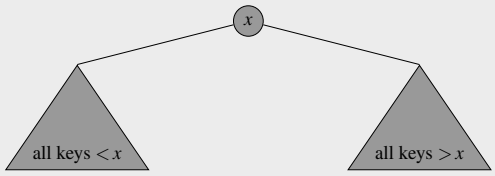


Figure 11.77 The binary search tree condition. For every node with key x: all keys in the left subtree of the node have a key $< x$, and all keys in the right subtree have a key $> x$.

Binary search trees are good data structures for dictionaries because *insert* and *lookup* can be implemented simply and efficiently. If we perform a lookup for a key k in an empty BST T, we return "not found." (For simplicity, we allow a BST to be empty—that is, to contain zero nodes.) Otherwise, compare k to the key r stored in the root node of T:

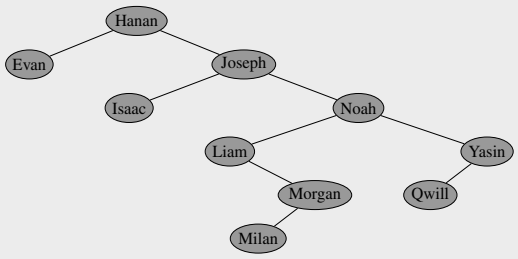


Figure 11.78 A binary search tree storing a set of 10 keys.

if $k = r$: return the value stored at the root.
if $k < r$: perform a lookup for k in the left subtree.
if $k > r$: perform a lookup for k in the right subtree.

The BST condition guarantees that we find the node with key k if it's in the tree. (You can prove this fact by induction.) The *insert* operation can be implemented similarly, by adding a new node exactly where a lookup for the key k would have found k.

The worst-case running time of *lookup* and *insert* is proportional to the height of the binary search tree. More "balanced" BSTs—in which every internal node has a left subtree with roughly the same height as its right subtree—have better performance. (There are

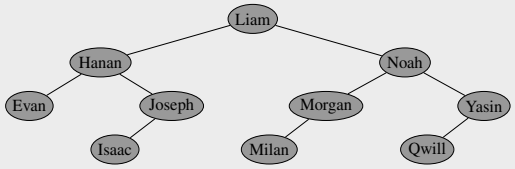


Figure 11.79 Another binary search tree with the same set of keys as in Figure 11.78, but a shallower depth.

many different BSTs with the same set of keys; for example, another BST that has the same keys as the BST in Figure 11.78 is shown in Figure 11.79.)

Most software therefore uses *balanced binary search trees* instead—for example, *AVL trees* or *red–black trees*. (See p. 6-53 for further discussion of AVL trees, and a proof of their efficiency.) The basic tool behind balanced BSTs is the *rotation,* which swaps a node in a BST with one of its children (while carefully rearranging the subtrees of the swapped nodes to maintain the BST property. (You can find the details in any good textbook on data structures.)

EXERCISES

11.122 Identify all of the simple cycles in the graph in Figure 11.80a.

11.123 Identify all of the simple cycles in the graph in Figure 11.80b.

11.124 Identify all of the simple cycles in the graph in Figure 11.80c.

11.125 Consider an undirected graph G with n nodes. In terms of n, what is the longest simple cycle that G can contain? Explain.

11.126 How does your answer to Exercise 11.125 change if the cycle is not required to be simple? Explain.

11.127 In the n-node complete graph $\mathcal{K}_n$, how many simple cycles is a particular node u involved in? Simplify your answer as n gets large. (You can use, without proof, the following useful fact: $\sum_{i=0}^{n} \frac{1}{i!}$ approaches $e = 2.71828\cdots$ as n grows.)

11.128 Let u be a node in a n-node complete *directed* graph: all edges except for self-loops are present. How many simple cycles is node u involved in? Again, simplify your answer for large n (you can again make use of the fact from Exercise 11.127).

11.129 A small modification to BFS can detect cycles involving a node s a directed graph, as shown in Figure 11.81. However, this modification doesn't quite work for undirected graphs. Give an example of an acyclic graph in which the algorithm Figure 11.81 falsely claims that there is a cycle. Then describe briefly how to modify this algorithm to correctly detect cycles involving node s in undirected graphs.

11.130 Recall Lemma 11.33: in any acyclic undirected graph, there exists a node whose degree is zero or one. Prove the following variation on this lemma: every *directed* acyclic graph contains a node with out-degree zero.

11.131 Prove the following extension to Lemma 11.33: there are *two* nodes of degree 1 in any acyclic undirected graph that contains at least one edge.

Recall Definition 11.30: a *cycle* $\langle u_0, u_1, \ldots, u_k, u_0\rangle$ is a path of length ≥ 2 from a node u_0 back to node u_0 that does not traverse the same edge twice. *At various times in class, I've tried to define cycles in all of the following ways—and they're all bogus definitions, in the sense that they describe something different from Definition 11.30.*

11.132 What's wrong with this definition? A *cycle* is a simple path from s to s.

11.133 What's wrong with this definition? A *cycle* is a path of length ≥ 2 from s to s.

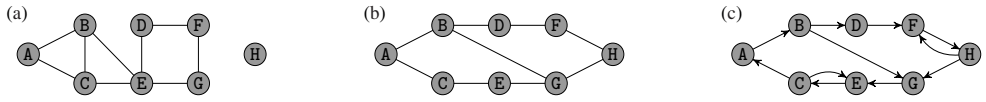


Figure 11.80 Three graphs.

(Buggily) Modified BFS:
Input: a graph $G = \langle V, E\rangle$ and a source node $s \in V$
Output: is s involved in a cycle in G?

```
1  Frontier := ⟨s⟩
2  Known := ∅
3  while Frontier is nonempty:
4      u := the first node in Frontier
5      remove u from Frontier
6      if s is a neighbor of u then
7          return "s is involved in a cycle."
8      for every neighbor v of u:
9          if v is in neither Frontier nor Known then
10             add v to the end of Frontier
11     add u to Known
12 return "s is not involved in a cycle."
```

Figure 11.81 BFS modified (slightly buggily) to detect cycles involving the node s.

11.134 What's wrong with this definition? A *cycle* is a path from s to s that doesn't traverse any edge more than once.

11.135 What's wrong with this definition? A *cycle* is a path from s to s that includes at least 3 distinct nodes.

11.136 What's wrong with this definition? A *cycle* is a path of length ≥ 2 from s to s that doesn't traverse any edge twice consecutively.

11.137 Definition 11.32 defines an acyclic graph as one containing no cycles, but it would have been equivalent to define acyclic graphs as those containing no *simple* cycles. Prove that G has a cycle if and only if G has a simple cycle.

11.138 Identify two different regular graphs that are trees. (Recall Definition 11.18: $G = \langle V, E \rangle$ is a *regular graph* if every $u \in V$ has $degree(u) = d$, for some fixed constant d.)

11.139 It turns out that there are two and only two different trees T that are regular graphs. Prove that, aside from the two you found in Exercise 11.138, there are no regular graphs that are trees.

11.140 A *square* is a simple cycle containing exactly four nodes. What is the largest number of squares possible in an undirected graph of n nodes?

11.141 A *triangle* is a simple cycle containing exactly three nodes. What is the largest number of triangles possible in an undirected graph of n nodes?

11.142 Let's figure out the largest number of edges that are possible in an n-node undirected graph $G = \langle E, V \rangle$ *that contains no triangles.* For two nodes u and v, argue that if $\{u, v\} \in E$, then we have $degree(u) + degree(v) \leq |V|$.

11.143 Using induction on the number of nodes, prove the following: if $G = \langle V, E \rangle$ is a triangle-free graph, then $|E| \leq |V|^2/4$. *(Hint: use Exercise 11.142.)*

11.144 Give an example of an n-node triangle-free graph that contains $\frac{n^2}{4}$ edges.

11.145 Is the graph represented by the adjacency list in Figure 11.82a a tree? Justify your answer.

11.146 What about the graph represented by the adjacency list in Figure 11.82b?

11.147 What about the graph represented by the adjacency list in Figure 11.82c?

11.148 What about the graph represented by the adjacency list in Figure 11.82d?

11.149 Prove or disprove: in any tree with 3 or more nodes, there is a node of degree equal to 2.

11.150 Prove or disprove: in any rooted binary tree, there are an even number of leaves. (In a binary tree, all nodes have 0, 1, or 2 children.)

11.151 Prove or disprove: if an undirected graph $G = \langle V, E \rangle$ has $|V| - 1$ edges, then G must be a forest.

11.152 The following pair of definitions is subtly broken: the *root* of a tree is a node that is not a child, and a *leaf* is a node that is a child but not a parent. What's wrong?

11.153 What are the leaves in the tree in Figure 11.83a?

11.154 For the tree in Figure 11.83a, which nodes are internal nodes?

11.155 For the tree in Figure 11.83a, what the are parent, children, and siblings of node `D`?

11.156 For the tree in Figure 11.83a, what nodes are descendants of node `D`?

11.157 For the tree in Figure 11.83a, what nodes are ancestors of node `F`?

11.158 What is the height of the tree in Figure 11.83a?

(a)
```
A: B, E
B: A
C: D
D: C, F
E: A
F: D
```

(b)
```
A: C
B: C, E
C: A, B, F
D: E
E: B, D
F: C
```

(c)
```
A: D
B: E, F
C: D, F
D: A, C
E: B
F: B, C
```

(d)
```
A: C, D, F
B: F
C: A, E, F
D: A
E: C
F: A, B, C
```

Figure 11.82 Adjacency lists for a few graphs, which may or may not be trees.

Figure 11.83 Two rooted trees, with A as the root.

11.159 Let T be an arbitrary n-node rooted tree, with root r and with ℓ different leaves. Prove or disprove: if we reroot T at a new node $r' \neq r$, then the number of leaves remains exactly ℓ.

A complete binary tree of height *h has "no holes": reading from top-to-bottom and left-to-right, every node exists. A* nearly complete binary tree *has every node until the last row, which is allowed to stop early. (Every complete binary tree is nearly complete.) See Figure 11.84, and see also p. 5-38 for a discussion of* heaps, *which are a data structure represented as a nearly complete binary tree.)*

11.160 Prove by induction that a complete binary tree of height h contains precisely $2^{h+1} - 1$ nodes.

11.161 How many leaves does a nearly complete binary tree of height h have? Give the smallest and largest possible values, and explain.

11.162 What is the diameter of a nearly complete binary tree of height h? Again, give the smallest and largest possible values, and explain. (Recall that the *diameter* of a graph is the maximum distance [length of shortest path] between nodes. See Exercise 11.89.)

11.163 Suppose that we "reroot" a complete binary tree of height h by instead designating one of the erstwhile leaves as the root. What is the height of the rerooted tree?

11.164 What is the diameter of a complete binary tree rerooted at one of its leaves (as in Exercise 11.163)?

11.165 How many leaves are in a complete binary tree rerooted at one of its leaves (as in Exercise 11.163)?

11.166 Describe an 1000-node binary tree with height as large as possible. Justify your answer.

11.167 Describe an 1000-node binary tree with height as small as possible. Justify.

11.168 Describe an 1000-node binary tree with as many leaves as possible. Justify.

11.169 Describe an 1000-node binary tree with as few leaves as possible. Justify.

11.170 In terms of n, what is the largest possible height for an n-node binary tree in which *every node has precisely zero or two children*? Justify your answer.

11.171 In what order are nodes of the tree in Figure 11.83b traversed by a pre-order traversal?

11.172 In what order are nodes of the tree in Figure 11.83b traversed by an in-order traversal?

11.173 In what order are nodes of the tree in Figure 11.83b traversed by a post-order traversal?

11.174 Draw the binary tree with in-order traversal $4, 1, 2, 3, 5$; pre-order traversal $1, 4, 3, 2, 5$; and post-order traversal $4, 2, 5, 3, 1$.

11.175 Do the same for the tree with in-order traversal $1, 3, 5, 4, 2$; pre-order traversal $1, 3, 5, 2, 4$; and post-order traversal $4, 2, 5, 3, 1$.

11.176 Describe (that is, fully explain the structure of) an n-node binary tree T for which the *pre-order* and *in-order* traversals of T result in precisely the same ordering of T's nodes. (That is, **pre-order-traverse**(T) = **in-order-traverse**(T).)

11.177 Describe a binary tree T for which the *pre-order* and *post-order* traversals result in precisely the same ordering of T's nodes. (That is, **pre-order-traverse**(T) = **post-order-traverse**(T).)

Figure 11.84 A complete and nearly complete binary tree of height 3.

11.178 Prove that there are two distinct binary trees T and T' such that pre-order and post-order traversals are both identical on the trees T and T'. (That is, **pre-order-traverse**(T) = **pre-order-traverse**(T') and **post-order-traverse**(T) = **post-order-traverse**(T') but $T \neq T'$.)

11.179 Give a recursive algorithm to reconstruct a tree from the in-order and post-order traversals.

11.180 Argue that we didn't leave out any spanning trees of the graph in Figure 11.73.

11.181 How many spanning trees does the graph in Figure 11.85a have? Explain.

11.182 How many spanning trees does the graph in Figure 11.85b have? Explain.

Figure 11.85 Two graphs.

11.5 Weighted Graphs

> Some minds improve by travel, others, rather,
> Resemble copper wire, or brass,
> Which gets the narrower by going farther!
>
> Thomas Hood (1799–1845)
> "Ode to Rae Wilson Esq." (1837)

Many real-world situations are naturally modeled by different edges having different "weights": the price of an airplane flight, the closeness of a friendship, the physical length of a road, the time required to transmit data across an internet connection. These graphs are called *weighted graphs:*

Definition 11.40: Weighted graph.
A *weighted graph* is a graph $G = \langle V, E \rangle$ and a weight function $w : E \to \mathbb{R}^{\geq 0}$, so that each edge $e \in E$ has a *weight* $w(e) \geq 0$. For simplicity of notation, we'll often write w_e instead of $w(e)$; we'll also sometimes refer to w_e as the *length* of the edge e.

In a weighted graph, the *length* of a path is the sum of the lengths of the edges traversed by the path. (A *shortest path* is, as before, one with the smallest length.)

Definition 11.40 considers only nonnegative weights—every $w_e \geq 0$—which is a genuine restriction. (For example, the "signed" social networks from Figure 11.16a have positive and negative weights signifying friendship and enmity.) Some, but not all, of the results that we'll discuss in this section carry over to the setting of negative weights.

Either undirected or directed graphs can be weighted. Aside from the length of a path, all of the other notions and terminology from unweighted graphs carry over: neighbors and degree, paths and connectivity, and so forth. Weighted graphs can be represented just as unweighted graphs were, using adjacency lists or adjacency matrices: we typically store the weight of edge $\langle u, v \rangle$ directly in the $\langle u, v \rangle$th entry of the adjacency matrix, or attach the edge weight as an additional slot in the adjacency list entries.

Example 11.43: A weighted graph.
Figure 11.86 shows the highway system from Example 11.4, with each road labeled by its length. There are two simple paths between Orlando and Lake City:

Orlando $\leftrightarrow$ Tampa $\leftrightarrow$ Lake City	$85 + 180 = 265$ miles

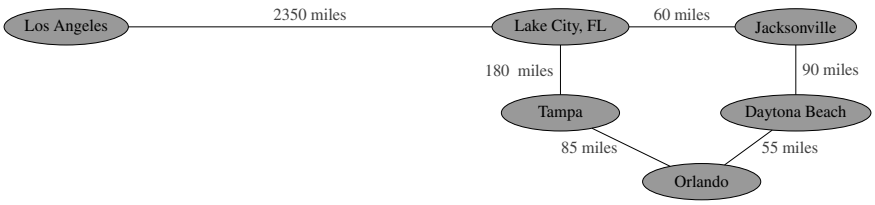


Figure 11.86 The highway system from Figure 11.4b, with each road labeled by its length.

Orlando $\leftrightarrow$ Daytona Beach $\leftrightarrow$ Jacksonville $\leftrightarrow$ Lake City $\quad 55 + 90 + 60 = 205$ miles.

The second path is shorter, even though it traverses more edges, as $265 > 205$.

Taking it further: The primary job of a web search engine is to respond to a user's search query ("give me web pages about Thomas Hood") with a list of relevant pages. There's a complex question of data structures, parallel computing, and networking infrastructure in solving even the simplest part of this task: identifying the set R of web pages (out of many billions) that contain the search term. A subtler challenge—and at least as important—is figuring out how to *rank* the set R. What pages in R are the "most important," the ones that we should display on the first page of results? Google uses a weighted graph (and probability) to do some of this ranking; see p. 11-90.

11.5.1 Shortest Paths in Weighted Graphs: Dijkstra's Algorithm

A *shortest path* from s to t in a weighted graph is the path connecting s and t that has shortest total length. In many natural applications where shortest paths are useful, we have weights on edges: you probably want the *shortest* walking route from the bar back to your apartment, for example, not necessarily the one with the fewest turns. In Example 11.43, we already saw a case in which the shortest path used more edges than necessary. Thus we cannot directly use breadth-first search to compute distances in weighted graphs. But we *can* compute distances using an algorithm that's very similar in spirit to BFS.

The basic idea of breadth-first search is to "expand outward" from the source node s in layers, accumulating a set of nodes u for which we know the distance from s to u. We add nodes in increasing order of their distance from s, and eventually we've computed distances from s to all nodes in the graph. (See Figure 11.87.) The trouble for weighted graphs is that the order in which BFS builds up its knowledge about shortest paths doesn't always work (as in Example 11.43). But we can use a cleverer way of building up knowledge about the network to find shortest paths in weighted graphs, too, following the same intuition. The algorithm that we'll describe is due to Edsger Dijkstra, and hence it is known as *Dijkstra's algorithm.*

Taking it further: Edsger Dijkstra (1930–2002) was a Dutch computer scientist—one of the founders of theoretical computer science, and the 1972 Turing Award winner. An irrelevant-to-this-section quotation attributed to Dijkstra, for your amusement: "computer science is no more about computers than astronomy is about telescopes." (This field *is* named strangely.) An irrelevant-to-this-section challenge, also for your amusement: name a common English word that, like `DIJKSTRA`, has at least five consecutive consonants (or 6 or even 7, which is technically possible—though, honestly, you'd probably be annoyed if

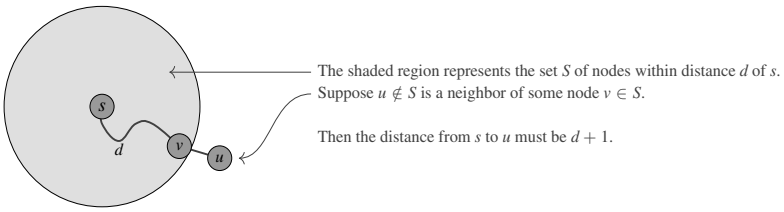


Figure 11.87 The intuition of breadth-first search, for an unweighted graph.

someone claimed the 7-consonant example as a word, and it's only in some dictionaries). (Not SYZYGY or RHYTHMS; Y is a vowel if it's used as a vowel!)

The key idea of Dijkstra's algorithm has parallels with BFS: *Suppose that we know the distance from a source node s to every node in some set S of nodes. (Assume that $s \in S$.) We will find some node* not *in S for which we can determine the shortest path from s.* For now, let's not worry about where this set S came from; the key point is just that we are assuming that we know distances to certain nodes (those in S), and we seek to leverage that existing knowledge to learn the distance to some other node (not previously in S). We'll then add that new node to S and iterate.

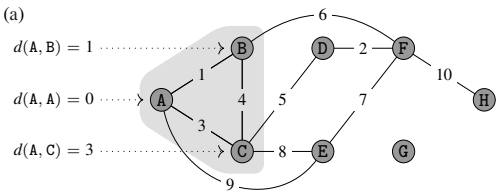
Before we state the formal result, let's look at an example:

Example 11.44: An example of distances.
Consider the weighted, undirected graph in Figure 11.88. Suppose we know the distances from A to every node in the shaded set $S = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$: in other words, we know $d(\mathtt{A}, \mathtt{A}) = 0$ and $d(\mathtt{A}, \mathtt{B}) = 1$ and $d(\mathtt{A}, \mathtt{C}) = 3$. We wish to expand our set of known nodes by adding a neighbor of an node whose distance is already known. The candidate nodes are neighbors of nodes with known distances that themselves do not have known distances—which are $\{\mathtt{D}, \mathtt{E}, \mathtt{F}\}$. Their candidate distances are shown in Figure 11.88b.

Let's argue that we can now conclude that $d(\mathtt{A}, \mathtt{F}) = 7$. The key reason is that, to get from A to F, we have to "escape" the set of shaded nodes in Figure 11.88a—and every "escape route" (path to F) must reach its last shaded node v (that's $d(\mathtt{A}, v)$) and then follow an edge to its first non-shaded node u (that's $w_{v,u}$). Because the table in Figure 11.88b tells us that every path out of the shaded region has length at least 7, and we've found a path from A to F with exactly that length, we can conclude that $d(\mathtt{A}, \mathtt{F}) = 7$.

Computing the distance to a new node

The same basic reasoning that we used in Example 11.44 will allow us to prove a general observation that's the foundation of Dijkstra's algorithm:



(b)

node	*distance*
F (via B)	$d(\mathtt{A}, \mathtt{B}) + w_{\mathtt{B},\mathtt{F}} = 1 + 6 = 7$
E (via A)	$d(\mathtt{A}, \mathtt{A}) + w_{\mathtt{A},\mathtt{E}} = 0 + 9 = 9$
E (via C)	$d(\mathtt{A}, \mathtt{C}) + w_{\mathtt{C},\mathtt{E}} = 3 + 8 = 11$
D (via C)	$d(\mathtt{A}, \mathtt{C}) + w_{\mathtt{C},\mathtt{D}} = 3 + 5 = 8$

Figure 11.88 (a) A weighted graph. If we know the distance from A to each node in the shaded set (as marked), then (b) the candidate nodes are those unmarked neighbors of nodes with known distances, with the candidate distances shown. See Example 11.44.

Lemma 11.41: Foundation of Dijkstra's Algorithm.
Let $G = \langle V, E \rangle$ be a graph with edge weights w, let $S \subset V$ be a set of nodes, and let $s \in S$ be a source node. Let $d(s, v)$ denote the distance from s to v for every node v in S. For a node $u \notin S$, define

$$d_u := \min_{v \in S : u \text{ is a neighbor of } v} d(s, v) + w_{v,u}.$$

Let u^* be the node $u \notin S$ for which d_u is minimized. Then the distance from s to u^* is d_{u^*}.

Before we prove the lemma, let's restate the claim in slightly less notation-heavy English. See Figure 11.89. We have a set S of shaded nodes for which we know the distance from s. We examine all unshaded nodes u that are neighbors of shaded nodes v. For each shaded/unshaded pair, we've computed the sum of the distance $d(s, v)$ and the edge weight $w_{v,u}$. And we've chosen the pair $\langle v^*, u^* \rangle$ that minimizes this quantity.

The lemma says that the shortest path from s to u^* must have length equal to $d_{u^*} := d(s, v^*) + w_{v^*,u^*}$. The intuition is the same as in Example 11.44: to get from s to u^*, we have to somehow "escape" the set of shaded nodes—and, by the way that we chose u^*, every "escape route" must have length at least d_{u^*}.

Problem-solving tip: When we want to prove that $x = y$, it's sometimes easier to prove $x \geq y$ and $x \leq y$ separately.

Proof of Lemma 11.41. We must show that the distance from s to u^* is d_{u^*}, and we'll do it in two steps: by showing that the distance is no more than d_{u^*}, and by showing that the distance is no less than d_{u^*}.

Claim #1: The distance from s to u^* is $\leq d_{u^*}$. We must argue that there *is* a path that has length $d(s, v^*) + w_{v^*,u^*}$ from s to u^*. By assumption and the fact that $v^* \in S$, we know that $d(s, v^*)$ is the distance from s to v^*, so there must exist a path P of length $d(s, v^*)$ from s to v^*. (It's the curved line in Figure 11.89.) By tacking u^* onto the end of P, we've constructed a path from s to u^* via v^* with length $d(s, v^*) + w_{v^*,u^*}$.

Claim #2: The distance from s to u^* is $\geq d_{u^*}$. Consider an arbitrary path P from s to u^*. We must show that P has length at least $d(s, v^*) + w_{v^*,u^*}$. What can P look like? The node s is in the set S, so P starts out at $s \in S$, then wanders around for a while inside S, then crosses outside of S for the first time, wanders around outside S for a while, and eventually ends up at $u^* \notin S$. Nothing prevents P from re-entering (and later re-exiting) S after its first departure—indeed, it can go in and out of S several times—but it definitely

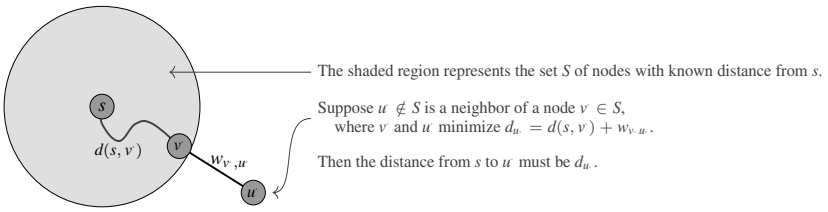


Figure 11.89 The intuition for Lemma 11.41. (Compare to the intuition for BFS, in Figure 11.87.)

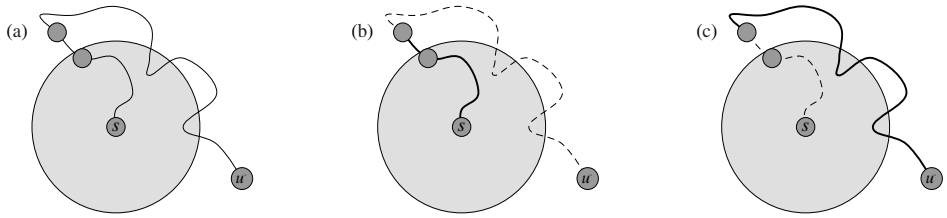


Figure 11.90 An illustration of the proof of Claim #2 in Lemma 11.41. (a) Any path P from s to u^* can be broken down into two parts: (b) the portion of P up to (and including) its first exit from S, and (c) the portion of P after the first exit from S.

has to leave S at least once. (See Figure 11.90.) Therefore:

$$
\begin{aligned}
&\text{the length of } P \\
&= (\text{the length of } P \text{ up to the first exit}) + (\text{the length of } P \text{ after the first exit}) \\
&\geq (\text{the length of the shortest path exiting } S) + (\text{the length of } P \text{ after the first exit}) \\
&\qquad\qquad \textit{P up to the first exit is a path exiting S, so its length is at least the length of the shortest such path} \\
&\geq d(s,v^*) + w_{v^*,u^*} + (\text{the length of } P \text{ after the first exit}) \\
&\qquad\qquad \textit{we chose u^* and v^* so that $d(s,v^*) + w_{v^*,u^*}$ is exactly the length of the shortest path exiting S} \\
&\geq d(s,v^*) + w_{v^*,u^*} + 0 \qquad\qquad \textit{all edge weights are nonnegative, so all path lengths are ≥ 0 too} \\
&= d_{u^*}. \qquad\qquad \textit{definition of d_{u^*}}
\end{aligned}
$$

Thus the length of P is at least d_{u^*}, which concludes the proof of Claim #2.

We've therefore argued that the distance from s to u^* is both $\leq d_{u^*}$ (Claim #1) and $\geq d_{u^*}$ (Claim #2). Thus the distance is precisely d_{u^*}, and the lemma follows. □

Dijkstra's Algorithm:
Input: a weighted graph $G = \langle V, E \rangle$, nonnegative edge weights $w_e \geq 0$, and a source node $s \in V$.
Output: the distance from s to every node in G

```
1 let S := {s} and let d(s,s) := 0                    // S is the set of nodes with known distances.
2 while there exists a node in S with a neighbor not in S:
3     for every node u ∉ S, define d_u as min_v d(s,v) + w_{v,u}, where v is any node in S that's a neighbor of u
4     u* := the node with the smallest d_u
5     add u* to S and set d(s,u*) := d_u
6 for every node u ∈ V − S:
7     d(s,u) := ∞
8 return the recorded values d(s,u)
```

Figure 11.91 The pseudocode for Dijkstra's algorithm.

Dijkstra's Algorithm

With Lemma 11.41 proven, we can now put together the pieces of the entire algorithm. The lemma describes a way to take a set S of nodes with known distance from the source node s, and correctly calculate the distance from s to a new node $u \notin S$. In Dijkstra's algorithm, the idea is to apply the calculation from Lemma 11.41 repeatedly to find all distances from the given source node s. We'll need a base case to get started, but that's straightforward: we start with the set of nodes with known distance from s as $S = \{s\}$, where the distance from s to s is zero. The full algorithm is shown in Figure 11.91.

Before we prove the algorithm's correctness, let's run through an example:

Example 11.45: Dijkstra's algorithm in action.
Figure 11.92 shows Dijkstra's algorithm running on the road network from Example 11.43.

The correctness of Dijkstra's Algorithm

We'll now prove the correctness of the algorithm, using Lemma 11.41 and induction:

Theorem 11.42: Correctness of Dijkstra's Algorithm.
Let $G = \langle V, E \rangle$ be a graph with nonnegative edge weights w_e for each edge. Let $s \in V$ be a source node, and let $d(s, \bullet)$ be the values computed by calling **Dijkstra**(G, w, s). Then, for every node u, we have that $d(s, u)$ is the length of the shortest path from s to u in G under w.

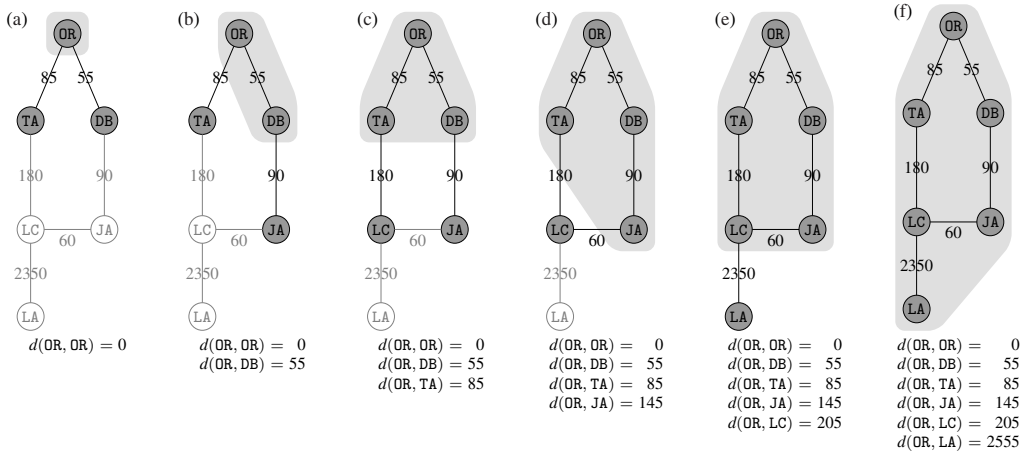


Figure 11.92 Dijkstra's algorithm on the graph in Figure 11.86 (with abbreviated names), starting from Orlando (OR). Nodes inside the shaded region have known distances (which are recorded in the table) from OR. In each iteration, the algorithm considers every "candidate" node—that is, a node that itself has an unknown distance but that has a neighbor with a known distance. (The light, unfilled nodes in the figure are not yet candidates.) In going from (a) to (b), for example, DB is the candidate with the smallest candidate distance (as in Lemma 11.41), so its distance can be recorded.

Proof. Looking at the algorithm, we see that Dijkstra's Algorithm records finite distances from s in Line 1 (for s itself) and Line 5 (for other nodes reachable from s). Suppose that Dijkstra's algorithm executes n iterations of the loop in Line 2, thus recording $n+1$ total distances in Lines 1 and 5—say in the order $u_0, u_1, \ldots, u_n$. Let $P(i)$ denote the claim that $d(s,u_i)$ is the length of the shortest s-to-u_i path. We claim by strong induction on i that $P(i)$ holds for all $i \in \{0,1,\ldots,n\}$.

For the base case ($i=0$), we must prove that $d(s,u_0)$ is recorded correctly. The 0th node u_0 is recorded in Line 1, so u_0 is the source node s itself. And the shortest path from s to s in any graph with nonnegative edge weights is the 0-hop path $\langle s \rangle$, of length 0.

For the inductive case ($i \geq 1$), we assume the inductive hypothesis $P(0), P(1), \ldots, P(i-1)$: that is, all recorded distances $d(s,u_0), d(s,u_1), \ldots, d(s,u_{i-1})$ are correct. We must prove $P(i)$: that is, that the recorded distance $d(s,u_i)$ is correct. But this follows immediately from Lemma 11.41: the algorithm chooses u_i as the $u \notin S$ minimizing

$$d_u := \min_{v \in S : u \text{ is a neighbor of } v} d(s,v) + w_{v,u},$$

where $S = \{u_0, u_1, \ldots, u_{i-1}\}$. Lemma 11.41 states precisely that this value d_u is the length of the shortest path from s to u.

Finally, observe that any node u that's only discovered in Line 6 is not reachable from s, and so indeed $d(s,u) = \infty$. (A fully detailed argument that the ∞ values are correct can follow the structure in Theorem 11.28, which proved the correctness of BFS.) □

> **Taking it further:** Dijkstra's algorithm as written in Figure 11.91 can be straightforwardly implemented to run in $O(|V| \cdot |E|)$ time: each iteration of the **while** loop (Line 2) can look at each edge to compute the smallest d_u. But with cleverer data structures, Dijkstra's algorithm can be made to run in $O(|E| \log |V|)$ time. This improved running-time analysis, as well as other shortest-path algorithms—for example, handling the case in which edge weights can be negative (it's worth thinking about where the proof of Lemma 11.41 fails if an edge e can have $w_e < 0$), or computing distances between *all pairs* of nodes instead of just every distance from a *single source*—is a standard topic in a course on algorithms. Any good algorithms text should cover these algorithms and their analysis.

Before we leave Dijkstra's algorithm, it's worth reflecting on its similarities with BFS. In both cases, we start from a seed set S of nodes for which we know the distance from s—namely $S = \{s\}$. Then we build up the set of nodes for which we know the distance from s by finding the unknown nodes that are closest to s, and adding them to S. Of course, BFS is conceptually simpler, but Dijkstra's algorithm solves a more complicated problem. It's a worthwhile exercise to think about what happens if Dijkstra's algorithm is run on an unweighted graph. (How does it relate to BFS?)

11.5.2 Spanning Trees in Weighted Graphs: Minimum Spanning Trees

Recall from Definition 11.38 that a *spanning tree* of a connected graph $G = \langle V, E \rangle$ is a tree $T = \langle V, E' \rangle$ where $E' \subseteq E$. As with shortest paths, in many of the applications in which spanning trees are interesting,

we actually want to find a spanning tree whose edges have minimum possible total cost. For example, when a college wants to put down networking cable in a new dorm building, they wish to ensure that the resulting network is connected, while minimizing the cost of construction.

In a weighted graph, the *cost* of a spanning tree T is the sum of the weights of its edges: $\sum_{e \in E'} w_e$. A *minimum spanning tree (MST)* is a spanning tree whose cost is as small as possible.

Example 11.46: Some minimum spanning trees.
Figure 11.93a and Figure 11.93b show two graphs (the road network from Example 11.43 and the larger connected component from Example 11.44). Their minimum spanning trees are shown in Figure 11.93c and Figure 11.93d. (For the first, every spanning tree omits exactly one edge from the lone cycle; the cheapest tree omits the most expensive edge.)

As with shortest paths in weighted graphs, the question of how to find a minimum spanning tree efficiently is more appropriate to an algorithms text than this one. But, between the Cycle Elimination Algorithm (Figure 11.74a) and Example 11.46, we've already done most of the work to develop a first algorithm.

Assume throughout that all edge weights are distinct. (This assumption lets us refer to "*the* most expensive edge" in a set of edges. Removing this assumption would complicate the language that we'd have to use, but it doesn't fundamentally change anything about the MST problem or its solution.)

Lemma 11.43: "Cycle Rule".
Let $G = \langle V, E \rangle$ be a connected undirected graph, and let C be a cycle in G. Let e be the heaviest edge in C. Then e is not in any minimum spanning tree of G.

Proof. Consider a spanning tree T of G, and suppose that $e = \{u, v\}$ is in T. See Figure 11.94a. We'll show that T is not a *minimum* spanning tree. (Thus the only minimum spanning trees of G do not include e.) By definition, the spanning tree T is connected. If we delete $\{u, v\}$ from T, the resulting graph will have two

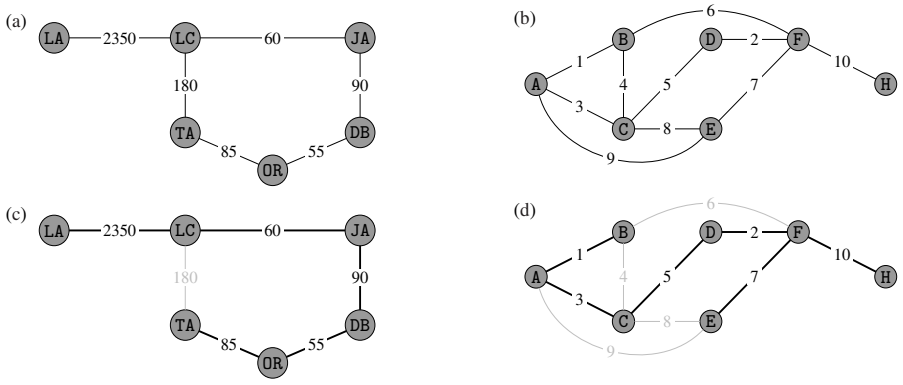
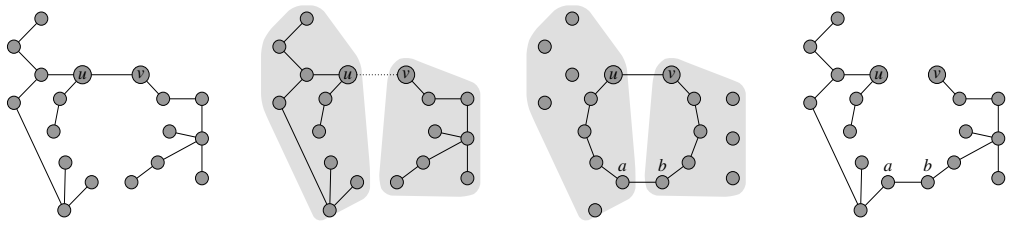


Figure 11.93 (a, b) Two weighted graphs, and (c, d) their minimum spanning trees.



(a) A spanning tree T that includes edge $\{u, v\}$ that is in some cycle C.

(b) Removing $\{u, v\}$ splits T into two connected components, U and V.

(c) Some other edge $\{a, b\}$ also in C must have $a \in U$ and $b \in V$.

(d) A new spanning tree T' that results from replacing $\{u, v\}$ by $\{a, b\}$.

Figure 11.94 The cycle rule for MSTs.

connected components, one containing u and the other containing v. (This fact follows by Corollary 11.36.) Call those connected components U and V, respectively. See Figure 11.94b.

Imagine following the cycle C from u to v the "long way" around C. This part of C starts at u, wanders around U for a while, and eventually crosses over into V, before finally arriving at v. Let $a \in U$ be the last node in U and b the first node in V as we go around C. (Note that C might go back and forth between U and V multiple times, but define a and b based on the *first* time C leaves U.) See Figure 11.94c.

Now define the graph T' as T with the edge $\{u, v\}$ removed and with the edge $\{a, b\}$ inserted instead. (See Figure 11.94d.) Crucially, T' is a spanning tree of G; because we've only swapped *which* edge connected the connected sets U and V. Thus T' remains connected and acyclic.

Now observe that the cost of T' is less than the cost of T, because the edge $\{u, v\}$ is heavier than the edge $\{a, b\}$. (Both $\{u, v\}$ and $\{a, b\}$ are in the cycle C, and by assumption $\{u, v\}$ is the heaviest edge in C.) But therefore T' is a cheaper spanning tree than T, and thus T isn't a minimum spanning tree. □

Finding MSTs by removing cycles

Lemma 11.43 immediately suggests that we can find minimum spanning trees using a modification of the Cycle Elimination Algorithm, which is shown in Figure 11.95. While the Weighted Cycle Elimination Algorithm is correct and reasonably efficient, there are more efficient algorithms based on Lemma 11.43. One such algorithm is called *Kruskal's Algorithm,* named after its discoverer Joseph Kruskal (1928–2010). (Kruskal published his MST algorithm in 1956, in part inspired by the 1920s-era work of Otakar Borůvka, a Czech mathematician interested in building efficient electrical grids in Eastern Europe [74, 19]—both remarkably early in the history of CS.)

The key idea of Kruskal's Algorithm is that by *sorting* the edges in increasing order, we can be more efficient: we add edges in increasing order of their weight, as long as doing so doesn't create a cycle. The insight of this algorithm is that, by considering edges in increasing order of weight, if including an edge e creates a cycle, then we know that e must be the heaviest edge in that cycle. See Figure 11.95. Kruskal's

Weighted Cycle Elimination Algorithm

```
1 while there exists any cycle C in G:
2     let e be the heaviest edge traversed by C
3     remove e from E
4 return the resulting graph ⟨V, E⟩.
```

Kruskal's Algorithm

```
1 Sort the edges e in increasing order of weight.
2 S := ∅
3 for each edge e (taken in increasing order of weight):
4     if the graph ⟨V, S ∪ {e}⟩ doesn't contain a cycle then
5         add e to S
6 return the resulting graph ⟨V, S⟩
```

Figure 11.95 Two algorithms to find a minimum spanning tree of a weighted connected graph $G = \langle V, E \rangle$.

algorithm is pretty efficient: the sorting step takes $O(|E| \log |E|)$ time, and each of the $|E|$ iterations of the **for** loop can be implemented using one call to BFS to test for a cycle. (And, in fact, there are some cleverer ways to implement Line 4 so that the entire algorithm runs in $O(|E| \log |E|)$ time.) Here's an example:

Example 11.47: Sample run of Kruskal's algorithm.
Figure 11.96 shows every iteration of Kruskal's algorithm when it's run on a small graph. The algorithm successively considers each edge of the original graph, in order from cheapest to most expensive, and discards the edge that it's considering if its inclusion would create a cycle.

Here is the general statement of correctness for both algorithms:

Theorem 11.44: Correctness of minimum spanning tree algorithms.
The Weighted Cycle Elimination Algorithm and Kruskal's Algorithm both return a minimum spanning tree for any weighted connected undirected graph.

Proof. The correctness of the Weighted Cycle Elimination Algorithm follows immediately from Lemma 11.43 (the cycle rule) and from Theorem 11.39 (the correctness of the Cycle Elimination Algorithm): the heaviest edge in any cycle does not appear in any MST, and we terminate with a spanning tree when we repeatedly eliminate any edge from an arbitrarily chosen cycle.

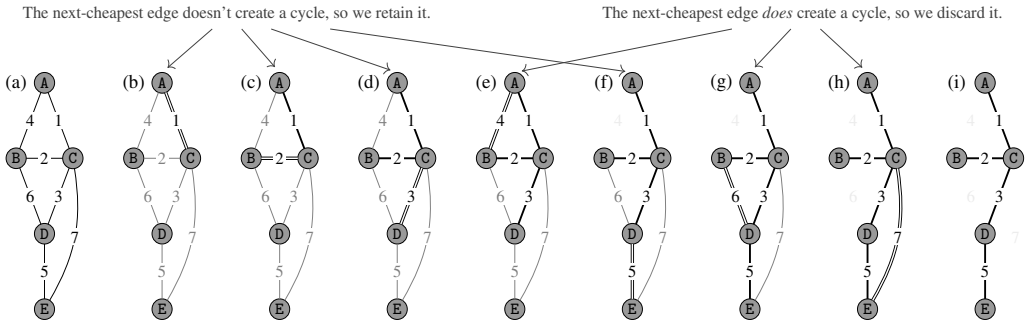


Figure 11.96 An example execution of Kruskal's Algorithm on the graph in (a), which produces the final minimum spanning tree in (i). In each iteration, we examine the next-cheapest edge (shown with a double line); if it doesn't create a cycle, we retain it in the spanning tree; if it does, we discard the edge.

For Kruskal's algorithm, consider an edge e that is *not* retained—that is, when e is considered, it is not included in the set S. The only reason that e wasn't included is that adding it would create a cycle C involving e and previously included edges—but because the edges are considered in increasing order of weight, that means that e is the heaviest edge in C. Thus by Lemma 11.43, Kruskal's algorithm removes only edges not contained in any minimum spanning tree. Because it only excludes edges that create cycles, the resulting graph is also connected—and thus a minimum spanning tree. □

Taking it further: There are several other commonly used algorithms for minimum spanning trees, using different structural properties than the Cycle Rule. For much more on these other algorithms, and for the clever data structures that allow Kruskal's Algorithm to be implemented in $O(|E| \log |E|)$ time, see any good textbook on algorithms.

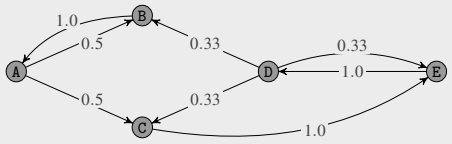
Computer Science Connections

Random Walks and Ranking Web Pages

When Google launched as a web search engine, one of its major innovations over its competition was in how it ranked the pages returned in response to a user's query. Here are two key ideas in Google's ranking system, called *PageRank* (named after Larry Page, one of Google's founders): (1) we can view a link from page u to page v as implicit "endorsement" of v by u; and (2) not all endorsements are equal: if a page u is endorsed by many other pages, then u's endorsements matter more. These points can be restated more glibly as: *a page is important if it is pointed to by many important pages.* The idea of PageRank is to break this apparent circularity using the *Random Surfer Model.* Imagine a hypothetical web user who starts at a random web page, and, at every time step, clicks on a randomly chosen link from the page they're currently visiting. The more frequently that this hypothetical user visits page u, the more important we'll say u is.

The Random Surfer explores the web using a so-called *random walk* on the web graph. In its simplest form, a random walk on a directed graph G visits a sequence of nodes: (1) we choose a starting node u_0 uniformly at random; and (2) in step $t = 1, 2, \ldots$, the next node u_t is chosen by picking a node uniformly at random from the out-neighborhood of the previous node u_{t-1}. (See Figure 11.97.) As you'll see in Exercises 11.204–11.208, under mild assumptions about G, there's a special probability distribution p over the nodes of the graph, called the *stationary distribution* of G, that has the following property: if we choose an initial node u with probability $p(u)$, and we then take one step of the random walk from u, the resulting probability distribution over the nodes is still p. And, it turns out, we can approximate p by the probability distribution observed simply by running the random walk for many steps, as in Figure 11.97b.

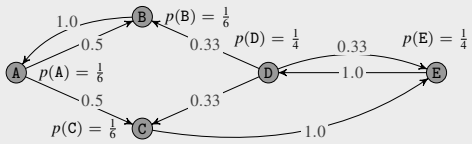
This stationary probability distribution p already has several good properties: $p(u)$ is higher when u has more in-neighbors, but it's increased even more when the in-neighbors of u have a high probability themselves. (In Figure 11.97c, for example, $p(\texttt{D}) > p(\texttt{B})$ and $p(\texttt{D}) > p(\texttt{C})$, despite B and C having higher in-degree than D.) But there are a few refinements to get to the full PageRank model. One issue is that the Random Surfer gets stuck at any page u that has no out-neighbors. (The next step isn't defined.) In this case, we'll have the Random Surfer jump to a completely random page (each of the $|V|$ nodes is chosen with



(a) A sample 5-node graph. Edges are annotated with their probabilities in a random walk; we can view the resulting weighted graph as defining the process.

node	*steps*
A	166,653
B	166,652
C	166,155
D	250,270
E	250,271

(b) A computer-generated random walk starting at A (this particular walk began ABABABABABABACEDCEDEDBABAC), with 1,000,000 steps: the first 25 steps visualized, and the number of these 1,000,000 steps spent at each node.



(c) The stationary distribution for G.

	A	B	C	D	E
A	0.03	0.45	0.45	0.03	0.03
B	0.88	0.03	0.03	0.03	0.03
C	0.03	0.03	0.03	0.03	0.88
D	0.03	0.31	0.31	0.03	0.31
E	0.03	0.03	0.03	0.88	0.03

(d) The updated link probabilities, with random restarts.

Figure 11.97 A random walk.

probability $\frac{1}{|V|}$). Second, the Random Surfer can also
get stuck in a "dead end" if there's a *group* of nodes that has no edges leaving it. Thus—and this change probably makes the Random Surfer more realistic anyway—we'll add a *restart probability* of 15% to every stage of the random walk: with probability 85%, we behave as previously described; with probability 15%, we jump to a randomly chosen node. See Figure 11.97d for the updated probabilities. (You can find more about the Random Surfer model and PageRank—including about how to calculate it on a graph with nodes numbering in the billions—in the original paper [21] or a good textbook on data mining, like [76]. There are also many other ingredients in Google's ranking recipe beyond PageRank, though PageRank was an early and important one.)

EXERCISES

11.183 Find all shortest paths from A to E in the graph in Figure 11.98a. Give both the path length and the paths themselves.

11.184 Do the same for all shortest paths from A to E in the graph in Figure 11.98b.

11.185 Do the same for all shortest paths from A to E in the graph in Figure 11.98c.

11.186 Do the same for all shortest paths from A to H in the graph in Figure 11.98d.

11.187 Do the same for all shortest paths from A to H in the graph in Figure 11.98e.

11.188 Let n be arbitrary. Give an example of an n-node weighted graph $G = \langle V, E \rangle$ with designated nodes $s \in V$ and $t \in V$ in which both of the following conditions hold:

(i) all edge weights are distinct (for any $e \in E$ and $e' \in E$, we have $w(e) \neq w(e')$ unless $e = e'$), and

(ii) for some $\alpha > 1$ and $c > 0$, there are at least $c \cdot \alpha^n$ different shortest paths between s and t.

11.189 Suppose that we are running Dijkstra's Algorithm on the graph in Figure 11.99a to compute distances from the node A. So far Dijkstra's Algorithm has computed four distances:

$$d(\texttt{A},\texttt{A}) = 0 \qquad d(\texttt{A},\texttt{B}) = 1 \qquad d(\texttt{A},\texttt{C}) = 3 \qquad d(\texttt{A},\texttt{F}) = 7$$

If we continue Dijkstra's algorithm for further iterations, it records the distance for a new node in each iteration. What is the next node recorded, and what is its distance?

11.190 What is the next node (after the one from Exercise 11.189) for which Dijkstra's algorithm records a distance, and what is its distance? And what's discovered after that node? List all subsequently discovered nodes, and their distances.

11.191 Trace Dijkstra's algorithm on the graph shown in Figure 11.99a to compute distances from the node H. List all discovered nodes and their distances, in the order in which they're discovered.

11.192 Identify *exactly* where the proof of correctness for Dijkstra's algorithm (specifically, Lemma 11.41) the argument fails if edge weights can be negative. Then give an example of a graph with negative edge weights in which Dijkstra's algorithm fails.

11.193 Suppose that G is a weighted, directed graph in which nodes represent physical states of a system, and an edge $\langle u, v \rangle$ indicates that it's possible to move from state u to state v. The weight $w_{u,v}$ denotes the *multiplicative* cost of the exchange: I can swap $w_{u,v}$ units

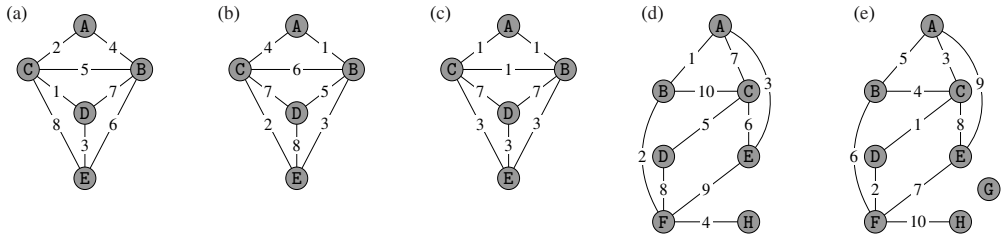
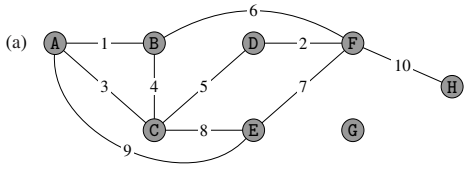
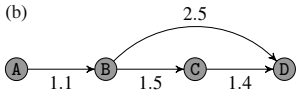


Figure 11.98 A few graphs.





The shortest multiplicative path from A to D is $\texttt{A} \to \texttt{B} \to \texttt{C} \to \texttt{D}$ at cost $1.1 \cdot 1.5 \cdot 1.4 = 2.31$, which is better than $\texttt{A} \to \texttt{B} \to \texttt{D}$ at cost $1.1 \cdot 2.5 = 2.75$.

Figure 11.99 Two graphs.

of u for 1 unit of v. (For example, if there's an edge $\langle \mathtt{X}, \mathtt{Y} \rangle$ with weight 1.04, then I can trade 2.08 units of energy in state X for 2 units of energy in state Y.) Now suppose that we wish to find a *shortest multiplicative path (SMP)* from a given node s to a given node t in G, where the cost of the path is the *product* of the edge weights along it. (Until now, we've cared about the *sum* of the costs.) See Figure 11.99b. Describe how to modify Dijkstra's algorithm to find the SMP in a given weighted graph G. Alternatively, describe how to modify a given weighted graph G into a graph G' so that Dijkstra's algorithm (unmodified) run on G' finds an SMP in G.

11.194 As you argued in Exercise 11.192, Dijkstra's algorithm may fail if edge weights are negative. State the condition that guarantees that your algorithm from Exercise 11.193 properly computes SMPs.

11.195 List all minimum spanning trees for the graph in Figure 11.98a.

11.196 Do the same for the graph in Figure 11.98b.

11.197 Do the same for the graph in Figure 11.98c. (Note that this graph has edges with nondistinct weights.)

11.198 Do the same for the graph in Figure 11.98d.

11.199 Do the same for the graph in Figure 11.98e.

11.200 Consider the undirected 9-node complete graph $\mathcal{K}_9$. There are $\binom{9}{2} = \frac{9 \cdot 8}{2} = 36$ unordered pairs of nodes in this graph, so there are 36 different edges. You're asked to assign each of these 36 edges a distinct weight from the set $\{1, 2, \ldots, 36\}$. (You get to choose which edges have which weights.) What's the cheapest possible minimum spanning tree of $\mathcal{K}_9$?

11.201 In the scenario from Exercise 11.200, what's the most expensive edge that can ever appear in a minimum spanning tree of $\mathcal{K}_9$? (Remember that you get to choose which edges have which weights.)

11.202 In the scenario from Exercise 11.200, what's the costliest possible minimum spanning tree of $\mathcal{K}_9$?

11.203 Generalize Exercise 11.200 and 11.202: what are the cheapest and most expensive possible MSTs for the graph $\mathcal{K}_n$ if all edges have distinct weights chosen from $\left\{1, 2, \ldots, \binom{n}{2}\right\}$? *(Hint: see Exercise 9.176.)*

A random walk *in a graph $G = \langle V, E \rangle$ proceeds as follows: we start at a node $u_0 \in V$, and, at every time step, we select as the next node u_{i+1} a uniformly chosen (out-)neighbor of u_i. (See p. 11-90.) Suppose we choose an initial node u_0 according to a probability distribution p, and we then take one step of the random walk from u_0 to get a new node u_1. The probability distribution p is called a* stationary distribution *if it satisfies the following condition: for every node $s \in V$, we have that $\Pr[u_0 = s] = \Pr[u_1 = s] = p(s)$. Such a distribution is called "stationary" because, if p is the probability distribution before a step of the walk, then p is still the probability distribution after a step of the walk (and thus the distribution "hasn't moved"—that is, is stationary).*

11.204 Argue that $p(\mathtt{A}) = p(\mathtt{B}) = p(\mathtt{C}) = \frac{1}{3}$ is a stationary distribution for the graph in Figure 11.100a.

11.205 Argue that the graph in Figure 11.100b has at least two distinct stationary distributions.

Suppose that we start a random walk at node A *in the graph in Figure 11.100a. Figure 11.101 shows the probability of being at any particular node after each of the first six steps of the random walk. Let $p_k(u)$ denote the probability of the kth step of this random*

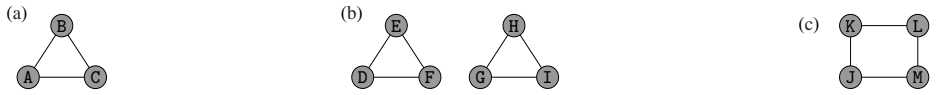


Figure 11.100 Some undirected graphs upon which a random walk can be performed.

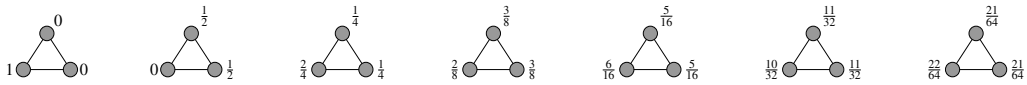


Figure 11.101 The distribution after the first few steps of a random walk on a triangle.

walk being at node u. Although we'll skip the proof, the following theorem turns out to be true of random walks on undirected graphs G: If G is connected and nonbipartite, then a unique stationary distribution p exists for this random walk on G (regardless of which node we choose as the initial node for the walk). Furthermore, the stationary distribution is the limit of the probability distributions p_k of where the random walk is in the kth step.

11.206 *(programming required.)* Write a random-walk simulator: take an undirected graph G as input, and simulate 2000 steps of a random walk starting at an arbitrary node. Repeat 2000 times, and report the fraction of walks that are at each node. What are your results on the graph from Figure 11.100a?

11.207 Argue that the above process doesn't converge to a unique stationary distribution in a bipartite graph. (For example, what's p_{1000} if a random walk starts at node J in the graph in Figure 11.100c? Node K?)

11.208 Let $G = \langle V, E \rangle$ be an arbitrary connected undirected graph. For any $u \in V$, define $p(u) = \frac{degree(u)}{2|E|}$. Prove that the probability distribution p is a stationary distribution for the random walk on G.

11.6 Chapter at a Glance

Formal Introduction

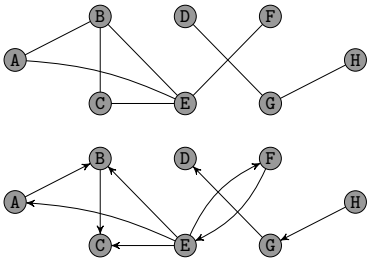


Figure 11.102 Two sample graphs.

A *graph* is a pair $G = \langle V, E \rangle$ where V is a set of *vertices* or *nodes*, and E is a set of *edges*. In a *directed graph*, the edges $E \subseteq V \times V$ are ordered pairs of vertices; in an *undirected graph*, the edges $E \subseteq \{\{u, v\} : u, v \in V\}$ are unordered pairs. A directed edge $\langle u, v \rangle$ goes from u to v; an undirected edge $\langle u, v \rangle$ goes between u and v. We sometimes write $\langle u, v \rangle$ even for an undirected graphs. A *simple graph* has no *parallel edges* joining the same two nodes and also has no *self loops* joining a node to itself. Two simple graphs, one undirected and one directed, are shown in Figure 11.102.

For an edge $e = \langle u, v \rangle$, we say that u and v are *adjacent*; v is a *neighbor* of u; u and v are the *endpoints* of e; and u and v are both *incident* to e. The *neighborhood* of a node u is $\{v : \langle u, v \rangle \in E\}$, its set of neighbors. The *degree* of u is the cardinality of u's neighborhood. In a directed graph, the *in-neighbors* of u are the nodes that have an edge pointing to u; the *out-neighbors* are the nodes to which u has an edge pointing; and the *in-degree* and *out-degree* of u are the number of in- and out-neighbors, respectively.

An *adjacency list* stores a graph using an array with $|V|$ entries; the slot for node u is a linked list of u's neighbors. An *adjacency matrix* stores the graph using a two-dimensional Boolean array of size $|V| \times |V|$; the value in $\langle$row u, column $v\rangle$ indicates whether the edge $\langle u, v \rangle$ exists.

Two graphs are *isomorphic* if they are identical except for the naming of the nodes. A *subgraph* of G contains a subset V' of G's nodes and a subset E' of G's edges joining elements of V'. An *induced subgraph* is a subgraph in which every edge that joins elements of V' is included in E'. A *complete graph* or *clique* is a graph $\mathcal{K}_n$ in which every possible edge exists. A *bipartite graph* is one in which nodes can be partitioned into sets L and R such that every edge joins a node in L to a node in R. A *regular graph* is one in which every node has identical degree. A *planar graph* is one that can be drawn on paper without any edges crossing.

Paths, Connectivity, and Distances

A *path* is a sequence of nodes $\langle v_1, v_2, \ldots, v_k \rangle$, with $k \geq 1$, where $\langle v_{i-1}, v_i \rangle \in E$ for every index $i \in \{1, 2, \ldots, k-1\}$. The path is *simple* if each v_i is distinct. This path has *length* $k - 1$—the number of edges that it traverses—and is a *path from* v_1 *to* v_k.

In an undirected graph, nodes u and v are *connected* if there exists a path from u to v. A *connected component* of $G = \langle V, E \rangle$ is a set $S \subseteq V$ such that (i) every $u \in S$ and $v \in S$ are connected; and (ii) for

every $w \notin S$, the set $S \cup \{w\}$ does not satisfy condition (i). The entire graph is *connected* if it has only one connected component, namely V.

In a directed graph, node u is *reachable from node* v if there exists a path from v to u; u and v are *strongly connected* if each is reachable from the other. A *strongly connected component* is a set S of nodes such that any two nodes in S are strongly connected and no node $x \notin S$ is strongly connected to any node $s \in S$.

Connectivity can be tested efficiently using *breadth-first search (BFS;* see Figure 11.103) or *depth-first search (DFS)* (which is similar, but storing newly discovered nodes at the front of *Frontier* rather than at the back—in other words, treating it as a stack rather than as a queue). The *distance* from node s to node t is the length of a shortest path from s to t. BFS can also be used to compute distances.

Trees

A *cycle* $\langle v_1, v_2, \ldots, v_k, v_1 \rangle$ is a path of length ≥ 2 from a node v_1 back to itself that does not traverse the same edge twice. The *length* of the cycle is k. The cycle is *simple* if each v_i is distinct. Cycles can be identified using BFS.

A graph is *acyclic* if it contains no cycles. Every acyclic graph has a node of degree 0 or 1. A *tree* is a connected, acyclic graph. (A *forest* is any acyclic graph.) A tree has one more node than it has vertex. A tree becomes disconnected if any edge is deleted; it becomes cyclic if any edge is added.

One node in a tree can be designated as the *root*. Every node other than the root has a *parent* (its neighbor that's closer to the root). If p is v's parent, then v is one of p's *children*. Two nodes with the same parent are *siblings*. A *leaf* is a node with no children; an *internal node* is a node with children. The *depth* of a node is its distance from the root; the *height* of the entire tree is the depth of deepest node. The *descendants* of u are those nodes that go through u to get the root; the *ancestors* are those nodes through which u's path to the root goes. The *subtree* rooted at u is the induced subgraph consisting of u and all descendants of u.

Breadth-First Search (BFS):
Input: a graph $G = \langle V, E \rangle$ and a source node $s \in V$
Output: the set of nodes reachable from s in G

```
1  Frontier := ⟨s⟩                  // Frontier will be a list of nodes to process, in order.
2  Known := ∅                       // Known will be the set of already-processed nodes.
3  while Frontier is nonempty:
4      u := the first node in Frontier
5      remove u from Frontier
6      for every neighbor v of u:
7          if v is in neither Frontier nor Known then
8              add v to the end of Frontier
9      add u to Known
10 return Known
```

Figure 11.103 Breadth-first search.

All nodes in *binary trees* have at most two children, called *left* and *right*. A *traversal* of a binary tree visits every node of the tree. An *in-order* traversal recursively traverses the root's left subtree, visits the root, and recursively traverses the root's right subtree. A *pre-order* traversal visits the root and recursively traverses the root's left and right subtrees; a *post-order* traversal recursively traverses the root's left and right subtrees and then visits the root.

A *spanning tree* of a connected graph $G = \langle V, E \rangle$ is a graph $T = \langle V, E' \subseteq E \rangle$ that's a tree. A spanning tree can by found by repeatedly identifying a cycle in G and deleting any edge in that cycle.

Weighted Graphs

In a *weighted graph*, each edge e has a weight $w_e \in \mathbb{R}^{\geq 0}$. (Although graphs with negative edge weights are possible, we haven't addressed them in any detail.) The length of a path in a weighted graph is the sum of the weights of the edges that it traverses. Shortest paths in weighted graphs can be found with Dijkstra's Algorithm (Figure 11.91), which expands a set of nodes of known distance one by one. Minimum spanning trees—spanning trees of the smallest possible total weight—in weighted graphs can be found with Kruskal's Algorithm (Figure 11.95) or by repeatedly identifying a cycle in G and deleting the heaviest edge in that cycle.

Key Terms and Results

Key Terms

Formal Introduction

- undirected and directed graphs
- nodes/vertices, edges
- parallel edges, self loops
- simple graphs
- adjacent node, incident edge
- (in/out-)neighbors, neighborhood
- (in/out-)degree
- adjacency list, adjacency matrix
- isomorphic graphs
- subgraphs
- complete, bipartite, regular, and planar graphs

Paths and Connectivity

- path
- connected (nodes), connected (graph)
- connected component
- reachability
- strongly connected component
- shortest path/distance
- breadth-first search (BFS)
- depth-first search (DFS)

Trees

- cycle
- tree, forest
- root, leaf, internal node
- child, parent, ancestor, descendant
- depth, height, subtree
- spanning tree

Weighted graphs

- Dijkstra's algorithm
- minimum spanning trees
- Kruskal's algorithm

Key Results

Formal Introduction

1 The "handshaking lemma": for any undirected graph $G = \langle V, E \rangle$, we have $\sum_{u \in V} degree(u) = 2|E|$.

2 Representing G with an adjacency matrix requires $\Theta(|V|^2)$ space; we can answer "what are all of u's neighbors?" in $\Theta(|V|)$ time and "is there an edge between u and v?" in $\Theta(1)$ time. Representing $G = \langle V, E \rangle$ with an adjacency list requires $\Theta(|V| + |E|)$ space; both questions take $1 + \Theta(degree(u))$ time.

Paths, Connectivity, and Distances

1 Connectivity can be tested using *breadth-first search (BFS)* (Figure 11.46b) or *depth-first search (DFS)* (Figure 11.49b). BFS can also be used to compute the distance between nodes in a graph, and it runs in $\Theta(|V| + |E|)$ time.

Trees

1 Any tree with n nodes has exactly $n - 1$ edges. Adding any edge to a tree creates a cycle; deleting any edge disconnects the graph.

2 A spanning tree of a graph G can by found by repeatedly identifying a cycle in G and deleting an arbitrary edge in that cycle.

Weighted Graphs

1 Shortest paths in weighted graphs can be found with Dijkstra's Algorithm (Figure 11.91) if all edges have nonnegative weights.

2 Minimum spanning trees in weighted graphs can be found with Kruskal's Algorithm (Figure 11.95) or by repeatedly identifying a cycle in G and deleting the heaviest edge in that cycle.