Chapter 3

Graphs

Our focus in this book is on problems with a discrete flavor. Just as continuous mathematics is concerned with certain basic structures such as real numbers, vectors, and matrices, discrete mathematics has developed basic combinatorial structures that lie at the heart of the subject. One of the most fundamental and expressive of these is the *graph*.

The more one works with graphs, the more one tends to see them everywhere. Thus, we begin by introducing the basic definitions surrounding graphs, and list a spectrum of different algorithmic settings where graphs arise naturally. We then discuss some basic algorithmic primitives for graphs, beginning with the problem of *connectivity* and developing some fundamental graph search techniques.

3.1 Basic Definitions and Applications

Recall from Chapter 1 that a graph G is simply a way of encoding pairwise relationships among a set of objects: it consists of a collection V of *nodes* and a collection E of *edges*, each of which "joins" two of the nodes. We thus represent an edge $e \in E$ as a two-element subset of V: $e = \{u, v\}$ for some $u, v \in V$, where we call u and v the *ends* of e.

Edges in a graph indicate a symmetric relationship between their ends. Often we want to encode asymmetric relationships, and for this we use the closely related notion of a *directed graph*. A directed graph G' consists of a set of nodes V and a set of *directed edges* E'. Each $e' \in E'$ is an *ordered pair* (u, v); in other words, the roles of u and v are not interchangeable, and we call u the *tail* of the edge and v the *head*. We will also say that edge e' *leaves node* u and *enters node* v.

When we want to emphasize that the graph we are considering is not directed, we will call it an *undirected graph*; by default, however, the term "graph" will mean an undirected graph. It is also worth mentioning two warnings in our use of graph terminology. First, although an edge e in an undirected graph should properly be written as a *set* of nodes $\{u, v\}$, one will more often see it written (even in this book) in the notation used for ordered pairs: $e = (u, v)$. Second, a *node* in a graph is also frequently called a *vertex*; in this context, the two words have exactly the same meaning.

Examples of Graphs Graphs are very simple to define: we just take a collection of things and join some of them by edges. But at this level of abstraction, it's hard to appreciate the typical kinds of situations in which they arise. Thus, we propose the following list of specific contexts in which graphs serve as important models. The list covers a lot of ground, and it's not important to remember everything on it; rather, it will provide us with a lot of useful examples against which to check the basic definitions and algorithmic problems that we'll be encountering later in the chapter. Also, in going through the list, it's useful to digest the meaning of the nodes and the meaning of the edges in the context of the application. In some cases the nodes and edges both correspond to physical objects in the real world, in others the nodes are real objects while the edges are virtual, and in still others both nodes and edges are pure abstractions.

1. *Transportation networks.* The map of routes served by an airline carrier naturally forms a graph: the nodes are airports, and there is an edge from u to v if there is a nonstop flight that departs from u and arrives at v. Described this way, the graph is directed; but in practice when there is an edge (u, v), there is almost always an edge (v, u), so we would not lose much by treating the airline route map as an undirected graph with edges joining pairs of airports that have nonstop flights each way. Looking at such a graph (you can generally find them depicted in the backs of in-flight airline magazines), we'd quickly notice a few things: there are often a small number of hubs with a very large number of incident edges; and it's possible to get between any two nodes in the graph via a very small number of intermediate stops.

   Other transportation networks can be modeled in a similar way. For example, we could take a rail network and have a node for each terminal, and an edge joining u and v if there's a section of railway track that goes between them without stopping at any intermediate terminal. The standard depiction of the subway map in a major city is a drawing of such a graph.

2. *Communication networks.* A collection of computers connected via a communication network can be naturally modeled as a graph in a few

different ways. First, we could have a node for each computer and an edge joining u and v if there is a direct physical link connecting them. Alternatively, for studying the large-scale structure of the Internet, people often define a node to be the set of all machines controlled by a single Internet service provider, with an edge joining u and v if there is a direct *peering relationship* between them—roughly, an agreement to exchange data under the standard BGP protocol that governs global Internet routing. Note that this latter network is more "virtual" than the former, since the links indicate a formal agreement in addition to a physical connection.

In studying wireless networks, one typically defines a graph where the nodes are computing devices situated at locations in physical space, and there is an edge from u to v if v is close enough to u to receive a signal from it. Note that it's often useful to view such a graph as directed, since it may be the case that v can hear u's signal but u cannot hear v's signal (if, for example, u has a stronger transmitter). These graphs are also interesting from a geometric perspective, since they roughly correspond to putting down points in the plane and then joining pairs that are close together.

3. *Information networks.* The World Wide Web can be naturally viewed as a directed graph, in which nodes correspond to Web pages and there is an edge from u to v if u has a hyperlink to v. The directedness of the graph is crucial here; many pages, for example, link to popular news sites, but these sites clearly do not reciprocate all these links. The structure of all these hyperlinks can be used by algorithms to try inferring the most important pages on the Web, a technique employed by most current search engines.

   The hypertextual structure of the Web is anticipated by a number of information networks that predate the Internet by many decades. These include the network of cross-references among articles in an encyclopedia or other reference work, and the network of bibliographic citations among scientific papers.

4. *Social networks.* Given any collection of people who interact (the employees of a company, the students in a high school, or the residents of a small town), we can define a network whose nodes are people, with an edge joining u and v if they are friends with one another. We could have the edges mean a number of different things instead of friendship: the undirected edge (u, v) could mean that u and v have had a romantic relationship or a financial relationship; the directed edge (u, v) could mean that u seeks advice from v, or that u lists v in his or her e-mail address book. One can also imagine bipartite social networks based on a

notion of *affiliation*: given a set X of people and a set Y of organizations, we could define an edge between $u \in X$ and $v \in Y$ if person u belongs to organization v.

Networks such as this are used extensively by sociologists to study the dynamics of interaction among people. They can be used to identify the most "influential" people in a company or organization, to model trust relationships in a financial or political setting, and to track the spread of fads, rumors, jokes, diseases, and e-mail viruses.

5. *Dependency networks.* It is natural to define directed graphs that capture the interdependencies among a collection of objects. For example, given the list of courses offered by a college or university, we could have a node for each course and an edge from u to v if u is a prerequisite for v. Given a list of functions or modules in a large software system, we could have a node for each function and an edge from u to v if u invokes v by a function call. Or given a set of species in an ecosystem, we could define a graph—a *food web*—in which the nodes are the different species and there is an edge from u to v if u consumes v.

This is far from a complete list, too far to even begin tabulating its omissions. It is meant simply to suggest some examples that are useful to keep in mind when we start thinking about graphs in an algorithmic context.

Paths and Connectivity One of the fundamental operations in a graph is that of traversing a sequence of nodes connected by edges. In the examples just listed, such a traversal could correspond to a user browsing Web pages by following hyperlinks; a rumor passing by word of mouth from you to someone halfway around the world; or an airline passenger traveling from San Francisco to Rome on a sequence of flights.

With this notion in mind, we define a *path* in an undirected graph $G = (V, E)$ to be a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in G. P is often called a path *from* v_1 *to* v_k, or a v_1-v_k path. For example, the nodes $4, 2, 1, 7, 8$ form a path in Figure 3.1. A path is called *simple* if all its vertices are distinct from one another. A *cycle* is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $k > 2$, the first $k-1$ nodes are all distinct, and $v_1 = v_k$—in other words, the sequence of nodes "cycles back" to where it began. All of these definitions carry over naturally to directed graphs, with the following change: in a directed path or cycle, each pair of consecutive nodes has the property that (v_i, v_{i+1}) is an edge. In other words, the sequence of nodes in the path or cycle must respect the directionality of edges.

We say that an undirected graph is *connected* if, for every pair of nodes u and v, there is a path from u to v. Choosing how to define connectivity of a

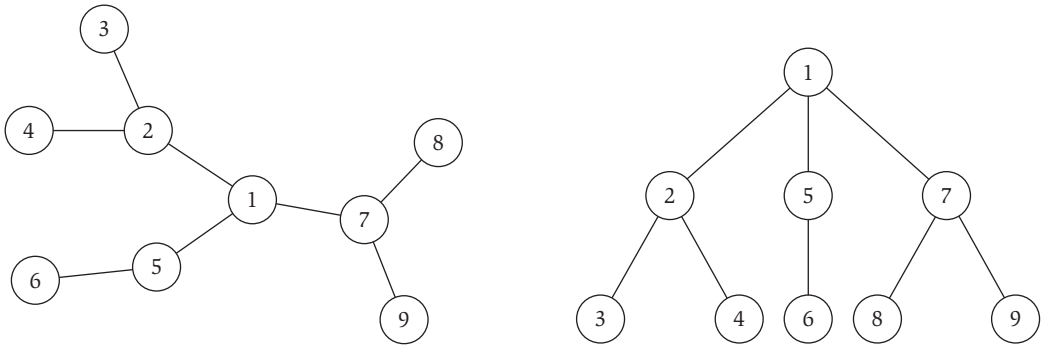

Figure 3.1 Two drawings of the same tree. On the right, the tree is rooted at node 1.

directed graph is a bit more subtle, since it's possible for u to have a path to v while v has no path to u. We say that a directed graph is *strongly connected* if, for every two nodes u and v, there is a path from u to v and a path from v to u.

In addition to simply knowing about the existence of a path between some pair of nodes u and v, we may also want to know whether there is a *short* path. Thus we define the *distance* between two nodes u and v to be the minimum number of edges in a u-v path. (We can designate some symbol like ∞ to denote the distance between nodes that are not connected by a path.) The term *distance* here comes from imagining G as representing a communication or transportation network; if we want to get from u to v, we may well want a route with as few "hops" as possible.

Trees We say that an undirected graph is a *tree* if it is connected and does not contain a cycle. For example, the two graphs pictured in Figure 3.1 are trees. In a strong sense, trees are the simplest kind of connected graph: deleting any edge from a tree will disconnect it.

For thinking about the structure of a tree T, it is useful to *root* it at a particular node r. Physically, this is the operation of grabbing T at the node r and letting the rest of it hang downward under the force of gravity, like a mobile. More precisely, we "orient" each edge of T away from r; for each other node v, we declare the *parent* of v to be the node u that directly precedes v on its path from r; we declare w to be a *child* of v if v is the parent of w. More generally, we say that w is a *descendant* of v (or v is an *ancestor* of w) if v lies on the path from the root to w; and we say that a node x is a *leaf* if it has no descendants. Thus, for example, the two pictures in Figure 3.1 correspond to the same tree T—the same pairs of nodes are joined by edges—but the drawing on the right represents the result of rooting T at node 1.

Rooted trees are fundamental objects in computer science, because they encode the notion of a *hierarchy*. For example, we can imagine the rooted tree in Figure 3.1 as corresponding to the organizational structure of a tiny nine-person company; employees 3 and 4 report to employee 2; employees 2, 5, and 7 report to employee 1; and so on. Many Web sites are organized according to a tree-like structure, to facilitate navigation. A typical computer science department's Web site will have an entry page as the root; the *People* page is a child of this entry page (as is the *Courses* page); pages entitled *Faculty* and *Students* are children of the *People* page; individual professors' home pages are children of the *Faculty* page; and so on.

For our purposes here, rooting a tree T can make certain questions about T conceptually easy to answer. For example, given a tree T on n nodes, how many edges does it have? Each node other than the root has a single edge leading "upward" to its parent; and conversely, each edge leads upward from precisely one non-root node. Thus we have very easily proved the following fact.

(3.1) *Every n-node tree has exactly $n - 1$ edges.*

In fact, the following stronger statement is true, although we do not prove it here.

(3.2) *Let G be an undirected graph on n nodes. Any two of the following statements implies the third.*

(i) *G is connected.*

(ii) *G does not contain a cycle.*

(iii) *G has $n - 1$ edges.*

We now turn to the role of trees in the fundamental algorithmic idea of *graph traversal*.

3.2 Graph Connectivity and Graph Traversal

Having built up some fundamental notions regarding graphs, we turn to a very basic algorithmic question: node-to-node connectivity. Suppose we are given a graph $G = (V, E)$ and two particular nodes s and t. We'd like to find an efficient algorithm that answers the question: Is there a path from s to t in G? We will call this the problem of determining *s-t connectivity*.

For very small graphs, this question can often be answered easily by visual inspection. But for large graphs, it can take some work to search for a path. Indeed, the *s-t* Connectivity Problem could also be called the *Maze-Solving* Problem. If we imagine G as a maze with a room corresponding to each node, and a hallway corresponding to each edge that joins nodes (rooms) together,

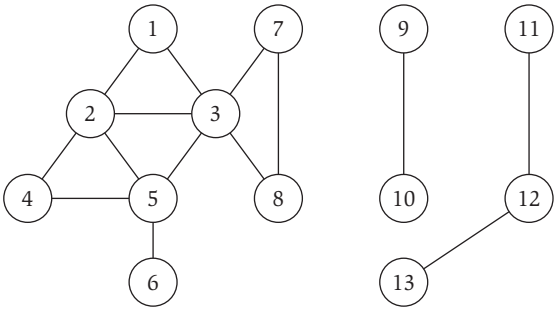

Figure 3.2 In this graph, node 1 has paths to nodes 2 through 8, but not to nodes 9 through 13.

then the problem is to start in a room s and find your way to another designated room t. How efficient an algorithm can we design for this task?

In this section, we describe two natural algorithms for this problem at a high level: breadth-first search (BFS) and depth-first search (DFS). In the next section we discuss how to implement each of these efficiently, building on a data structure for representing a graph as the input to an algorithm.

Breadth-First Search

Perhaps the simplest algorithm for determining *s-t* connectivity is *breadth-first search* (BFS), in which we explore outward from s in all possible directions, adding nodes one "layer" at a time. Thus we start with s and include all nodes that are joined by an edge to s—this is the first layer of the search. We then include all additional nodes that are joined by an edge to any node in the first layer—this is the second layer. We continue in this way until no new nodes are encountered.

In the example of Figure 3.2, starting with node 1 as s, the first layer of the search would consist of nodes 2 and 3, the second layer would consist of nodes 4, 5, 7, and 8, and the third layer would consist just of node 6. At this point the search would stop, since there are no further nodes that could be added (and in particular, note that nodes 9 through 13 are never reached by the search).

As this example reinforces, there is a natural physical interpretation to the algorithm. Essentially, we start at s and "flood" the graph with an expanding wave that grows to visit all nodes that it can reach. The layer containing a node represents the point in time at which the node is reached.

We can define the layers $L_1, L_2, L_3, \ldots$ constructed by the BFS algorithm more precisely as follows.

- Layer L_1 consists of all nodes that are neighbors of s. (For notational reasons, we will sometimes use layer L_0 to denote the set consisting just of s.)
- Assuming that we have defined layers $L_1, \ldots, L_j$, then layer L_{j+1} consists of all nodes that do not belong to an earlier layer and that have an edge to a node in layer L_j.

Recalling our definition of the distance between two nodes as the minimum number of edges on a path joining them, we see that layer L_1 is the set of all nodes at distance 1 from s, and more generally layer L_j is the set of all nodes at distance exactly j from s. A node fails to appear in any of the layers if and only if there is no path to it. Thus, BFS is not only determining the nodes that s can reach, it is also computing shortest paths to them. We sum this up in the following fact.

(3.3) *For each $j \geq 1$, layer L_j produced by BFS consists of all nodes at distance exactly j from s. There is a path from s to t if and only if t appears in some layer.*

A further property of breadth-first search is that it produces, in a very natural way, a tree T rooted at s on the set of nodes reachable from s. Specifically, for each such node v (other than s), consider the moment when v is first "discovered" by the BFS algorithm; this happens when some node u in layer L_j is being examined, and we find that it has an edge to the previously unseen node v. At this moment, we add the edge (u, v) to the tree T—u becomes the parent of v, representing the fact that u is "responsible" for completing the path to v. We call the tree T that is produced in this way a *breadth-first search tree*.

Figure 3.3 depicts the construction of a BFS tree rooted at node 1 for the graph in Figure 3.2. The solid edges are the edges of T; the dotted edges are edges of G that do not belong to T. The execution of BFS that produces this tree can be described as follows.

(a) Starting from node 1, layer L_1 consists of the nodes $\{2, 3\}$.

(b) Layer L_2 is then grown by considering the nodes in layer L_1 in order (say, first 2, then 3). Thus we discover nodes 4 and 5 as soon as we look at 2, so 2 becomes their parent. When we consider node 2, we also discover an edge to 3, but this isn't added to the BFS tree, since we already know about node 3.

We first discover nodes 7 and 8 when we look at node 3. On the other hand, the edge from 3 to 5 is another edge of G that does not end up in

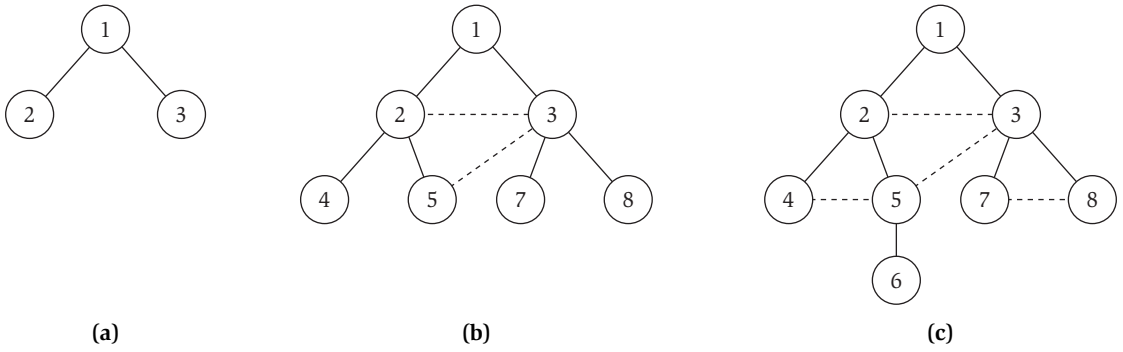

Figure 3.3 The construction of a breadth-first search tree T for the graph in Figure 3.2, with (a), (b), and (c) depicting the successive layers that are added. The solid edges are the edges of T; the dotted edges are in the connected component of G containing node 1, but do not belong to T.

the BFS tree, because by the time we look at this edge out of node 3, we already know about node 5.

(c) We then consider the nodes in layer L_2 in order, but the only new node discovered when we look through L_2 is node 6, which is added to layer L_3. Note that the edges $(4, 5)$ and $(7, 8)$ don't get added to the BFS tree, because they don't result in the discovery of new nodes.

(d) No new nodes are discovered when node 6 is examined, so nothing is put in layer L_4, and the algorithm terminates. The full BFS tree is depicted in Figure 3.3(c).

We notice that as we ran BFS on this graph, the nontree edges all either connected nodes in the same layer, or connected nodes in adjacent layers. We now prove that this is a property of BFS trees in general.

(3.4) *Let T be a breadth-first search tree, let x and y be nodes in T belonging to layers L_i and L_j respectively, and let (x, y) be an edge of G. Then i and j differ by at most* 1.

Proof. Suppose by way of contradiction that i and j differed by more than 1; in particular, suppose $i < j - 1$. Now consider the point in the BFS algorithm when the edges incident to x were being examined. Since x belongs to layer L_i, the only nodes discovered from x belong to layers L_{i+1} and earlier; hence, if y is a neighbor of x, then it should have been discovered by this point at the latest and hence should belong to layer L_{i+1} or earlier. ■

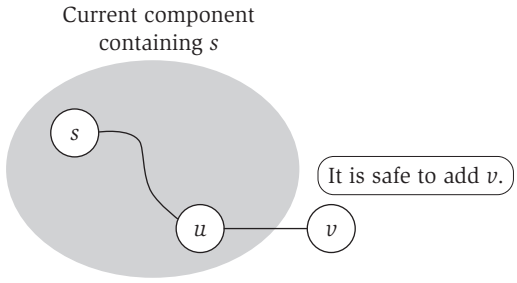

Figure 3.4 When growing the connected component containing s, we look for nodes like v that have not yet been visited.

Exploring a Connected Component

The set of nodes discovered by the BFS algorithm is precisely those reachable from the starting node s. We will refer to this set R as the *connected component* of G containing s; and once we know the connected component containing s, we can simply check whether t belongs to it so as to answer the question of s-t connectivity.

Now, if one thinks about it, it's clear that BFS is just one possible way to produce this component. At a more general level, we can build the component R by "exploring" G in any order, starting from s. To start off, we define $R = \{s\}$. Then at any point in time, if we find an edge (u, v) where $u \in R$ and $v \notin R$, we can add v to R. Indeed, if there is a path P from s to u, then there is a path from s to v obtained by first following P and then following the edge (u, v). Figure 3.4 illustrates this basic step in growing the component R.

Suppose we continue growing the set R until there are no more edges leading out of R; in other words, we run the following algorithm.

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```

Here is the key property of this algorithm.

(3.5) *The set R produced at the end of the algorithm is precisely the connected component of G containing s.*

Proof. We have already argued that for any node $v \in R$, there is a path from s to v.

Now, consider a node $w \notin R$, and suppose by way of contradiction, that there is an s-w path P in G. Since $s \in R$ but $w \notin R$, there must be a first node v on P that does not belong to R; and this node v is not equal to s. Thus there is a node u immediately preceding v on P, so (u, v) is an edge. Moreover, since v is the first node on P that does not belong to R, we must have $u \in R$. It follows that (u, v) is an edge where $u \in R$ and $v \notin R$; this contradicts the stopping rule for the algorithm. ■

For any node t in the component R, observe that it is easy to recover the actual path from s to t along the lines of the argument above: we simply record, for each node v, the edge (u, v) that was considered in the iteration in which v was added to R. Then, by tracing these edges backward from t, we proceed through a sequence of nodes that were added in earlier and earlier iterations, eventually reaching s; this defines an s-t path.

To conclude, we notice that the general algorithm we have defined to grow R is underspecified, so how do we decide which edge to consider next? The BFS algorithm arises, in particular, as a particular way of ordering the nodes we visit—in successive layers, based on their distance from s. But there are other natural ways to grow the component, several of which lead to efficient algorithms for the connectivity problem while producing search patterns with different structures. We now go on to discuss a different one of these algorithms, *depth-first search*, and develop some of its basic properties.

Depth-First Search

Another natural method to find the nodes reachable from s is the approach you might take if the graph G were truly a maze of interconnected rooms and you were walking around in it. You'd start from s and try the first edge leading out of it, to a node v. You'd then follow the first edge leading out of v, and continue in this way until you reached a "dead end"—a node for which you had already explored all its neighbors. You'd then backtrack until you got to a node with an unexplored neighbor, and resume from there. We call this algorithm *depth-first search* (DFS), since it explores G by going as deeply as possible and only retreating when necessary.

DFS is also a particular implementation of the generic component-growing algorithm that we introduced earlier. It is most easily described in recursive form: we can invoke DFS from any starting point but maintain global knowledge of which nodes have already been explored.

```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u, v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```

To apply this to *s-t* connectivity, we simply declare all nodes initially to be not explored, and invoke *DFS*(s).

There are some fundamental similarities and some fundamental differences between DFS and BFS. The similarities are based on the fact that they both build the connected component containing s, and we will see in the next section that they achieve qualitatively similar levels of efficiency.

While DFS ultimately visits exactly the same set of nodes as BFS, it typically does so in a very different order; it probes its way down long paths, potentially getting very far from s, before backing up to try nearer unexplored nodes. We can see a reflection of this difference in the fact that, like BFS, the DFS algorithm yields a natural rooted tree T on the component containing s, but the tree will generally have a very different structure. We make s the root of the tree T, and make u the parent of v when u is responsible for the discovery of v. That is, whenever *DFS*(v) is invoked directly during the call to *DFS*(u), we add the edge (u, v) to T. The resulting tree is called a *depth-first search tree* of the component R.

Figure 3.5 depicts the construction of a DFS tree rooted at node 1 for the graph in Figure 3.2. The solid edges are the edges of T; the dotted edges are edges of G that do not belong to T. The execution of DFS begins by building a path on nodes 1, 2, 3, 5, 4. The execution reaches a dead end at 4, since there are no new nodes to find, and so it "backs up" to 5, finds node 6, backs up again to 3, and finds nodes 7 and 8. At this point there are no new nodes to find in the connected component, so all the pending recursive DFS calls terminate, one by one, and the execution comes to an end. The full DFS tree is depicted in Figure 3.5(g).

This example suggests the characteristic way in which DFS trees look different from BFS trees. Rather than having root-to-leaf paths that are as short as possible, they tend to be quite narrow and deep. However, as in the case of BFS, we can say something quite strong about the way in which nontree edges of G must be arranged relative to the edges of a DFS tree T: as in the figure, nontree edges can only connect ancestors of T to descendants.

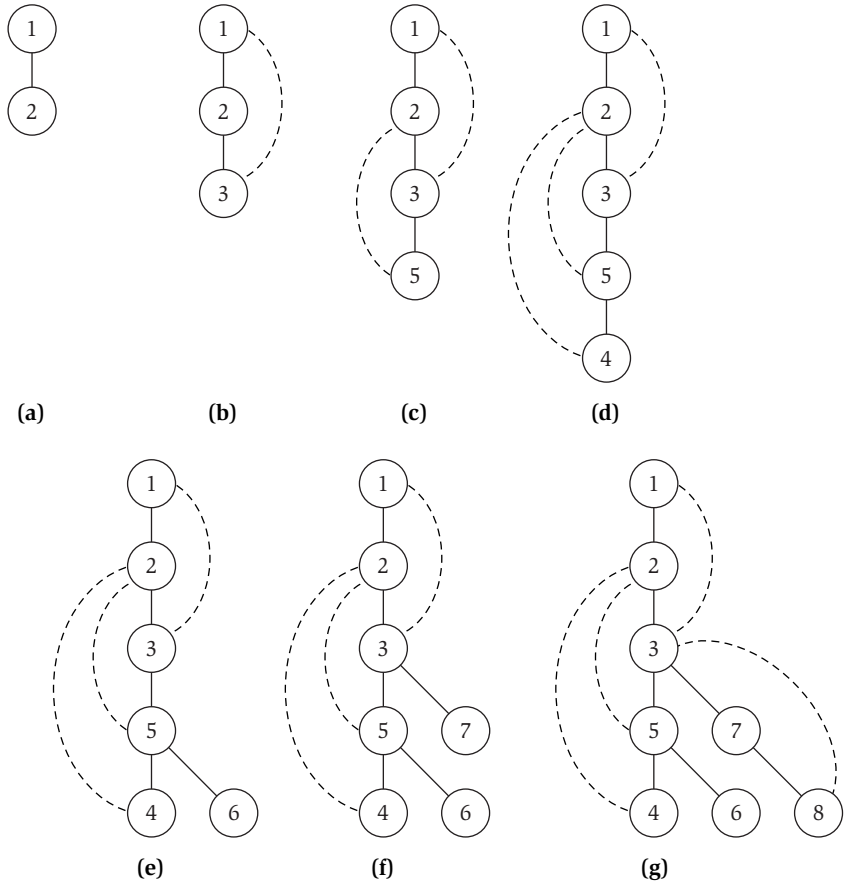

Figure 3.5 The construction of a depth-first search tree T for the graph in Figure 3.2, with (a) through (g) depicting the nodes as they are discovered in sequence. The solid edges are the edges of T; the dotted edges are edges of G that do not belong to T.

To establish this, we first observe the following property of the DFS algorithm and the tree that it produces.

(3.6) *For a given recursive call DFS(u), all nodes that are marked "Explored" between the invocation and end of this recursive call are descendants of u in T.*

Using (3.6), we prove

(3.7) *Let T be a depth-first search tree, let x and y be nodes in T, and let (x, y) be an edge of G that is not an edge of T. Then one of x or y is an ancestor of the other.*

Proof. Suppose that (x, y) is an edge of G that is not an edge of T, and suppose without loss of generality that x is reached first by the DFS algorithm. When the edge (x, y) is examined during the execution of $DFS(x)$, it is not added to T because y is marked "Explored." Since y was not marked "Explored" when $DFS(x)$ was first invoked, it is a node that was discovered between the invocation and end of the recursive call $DFS(x)$. It follows from (3.6) that y is a descendant of x. ■

The Set of All Connected Components

So far we have been talking about the connected component containing a particular node s. But there is a connected component associated with each node in the graph. What is the relationship between these components?

In fact, this relationship is highly structured and is expressed in the following claim.

(3.8) *For any two nodes s and t in a graph, their connected components are either identical or disjoint.*

This is a statement that is very clear intuitively, if one looks at a graph like the example in Figure 3.2. The graph is divided into multiple pieces with no edges between them; the largest piece is the connected component of nodes 1 through 8, the medium piece is the connected component of nodes 11, 12, and 13, and the smallest piece is the connected component of nodes 9 and 10. To prove the statement in general, we just need to show how to define these "pieces" precisely for an arbitrary graph.

Proof. Consider any two nodes s and t in a graph G with the property that there is a path between s and t. We claim that the connected components containing s and t are the same set. Indeed, for any node v in the component of s, the node v must also be reachable from t by a path: we can just walk from t to s, and then on from s to v. The same reasoning works with the roles of s and t reversed, and so a node is in the component of one if and only if it is in the component of the other.

On the other hand, if there is no path between s and t, then there cannot be a node v that is in the connected component of each. For if there were such a node v, then we could walk from s to v and then on to t, constructing a path between s and t. Thus, if there is no path between s and t, then their connected components are disjoint. ■

This proof suggests a natural algorithm for producing all the connected components of a graph, by growing them one component at a time. We start with an arbitrary node s, and we use BFS (or DFS) to generate its connected

component. We then find a node v (if any) that was not visited by the search from s, and iterate, using BFS starting from v, to generate its connected component—which, by (3.8), will be disjoint from the component of s. We continue in this way until all nodes have been visited.

3.3 Implementing Graph Traversal Using Queues and Stacks

So far we have been discussing basic algorithmic primitives for working with graphs without mentioning any implementation details. Here we discuss how to use lists and arrays to represent graphs, and we discuss the trade-offs between the different representations. Then we use these data structures to implement the graph traversal algorithms breadth-first search (BFS) and depth-first search (DFS) efficiently. We will see that BFS and DFS differ essentially only in that one uses a *queue* and the other uses a *stack*, two simple data structures that we will describe later in this section.

Representing Graphs

There are two basic ways to represent graphs: by an *adjacency matrix* and by an *adjacency list* representation. Throughout the book we will use the adjacency list representation. We start, however, by reviewing both of these representations and discussing the trade-offs between them.

A graph $G = (V, E)$ has two natural input parameters, the number of nodes $|V|$, and the number of edges $|E|$. We will use $n = |V|$ and $m = |E|$ to denote these, respectively. Running times will be given in terms of both of these two parameters. As usual, we will aim for polynomial running times, and lower-degree polynomials are better. However, with two parameters in the running time, the comparison is not always so clear. Is $O(m^2)$ or $O(n^3)$ a better running time? This depends on what the relation is between n and m. With at most one edge between any pair of nodes, the number of edges m can be at most $\binom{n}{2} \leq n^2$. On the other hand, in many applications the graphs of interest are connected, and by (3.1), connected graphs must have at least $m \geq n - 1$ edges. But these comparisons do not always tell us which of two running times (such as m^2 and n^3) are better, so we will tend to keep the running times in terms of both of these parameters. In this section we aim to implement the basic graph search algorithms in time $O(m + n)$. We will refer to this as *linear time*, since it takes $O(m + n)$ time simply to read the input. Note that when we work with connected graphs, a running time of $O(m + n)$ is the same as $O(m)$, since $m \geq n - 1$.

Consider a graph $G = (V, E)$ with n nodes, and assume the set of nodes is $V = \{1, \ldots, n\}$. The simplest way to represent a graph is by an *adjacency*

matrix, which is an $n \times n$ matrix A where $A[u, v]$ is equal to 1 if the graph contains the edge (u, v) and 0 otherwise. If the graph is undirected, the matrix A is symmetric, with $A[u, v] = A[v, u]$ for all nodes $u, v \in V$. The adjacency matrix representation allows us to check in $O(1)$ time if a given edge (u, v) is present in the graph. However, the representation has two basic disadvantages.

- The representation takes $\Theta(n^2)$ space. When the graph has many fewer edges than n^2, more compact representations are possible.
- Many graph algorithms need to examine all edges incident to a given node v. In the adjacency matrix representation, doing this involves considering all other nodes w, and checking the matrix entry $A[v, w]$ to see whether the edge (v, w) is present—and this takes $\Theta(n)$ time. In the worst case, v may have $\Theta(n)$ incident edges, in which case checking all these edges will take $\Theta(n)$ time regardless of the representation. But many graphs in practice have significantly fewer edges incident to most nodes, and so it would be good to be able to find all these incident edges more efficiently.

The representation of graphs used throughout the book is the adjacency list, which works better for sparse graphs—that is, those with many fewer than n^2 edges. In the *adjacency list* representation there is a record for each node v, containing a list of the nodes to which v has edges. To be precise, we have an array `Adj`, where `Adj`$[v]$ is a record containing a list of all nodes adjacent to node v. For an undirected graph $G = (V, E)$, each edge $e = (v, w) \in E$ occurs on two adjacency lists: node w appears on the list for node v, and node v appears on the list for node w.

Let's compare the adjacency matrix and adjacency list representations. First consider the space required by the representation. An adjacency matrix requires $O(n^2)$ space, since it uses an $n \times n$ matrix. In contrast, we claim that the adjacency list representation requires only $O(m + n)$ space. Here is why. First, we need an array of pointers of length n to set up the lists in `Adj`, and then we need space for all the lists. Now, the lengths of these lists may differ from node to node, but we argued in the previous paragraph that overall, each edge $e = (v, w)$ appears in exactly two of the lists: the one for v and the one for w. Thus the total length of all lists is $2m = O(m)$.

Another (essentially equivalent) way to justify this bound is as follows. We define the *degree* n_v of a node v to be the number of incident edges it has. The length of the list at `Adj`$[v]$ is list is n_v, so the total length over all nodes is $O\left(\sum_{v \in V} n_v\right)$. Now, the sum of the degrees in a graph is a quantity that often comes up in the analysis of graph algorithms, so it is useful to work out what this sum is.

(3.9) $\sum_{v \in V} n_v = 2m$.

Proof. Each edge $e = (v, w)$ contributes exactly twice to this sum: once in the quantity n_v and once in the quantity n_w. Since the sum is the total of the contributions of each edge, it is $2m$. ■

We sum up the comparison between adjacency matrices and adjacency lists as follows.

(3.10) *The adjacency matrix representation of a graph requires $O(n^2)$ space, while the adjacency list representation requires only $O(m + n)$ space.*

Since we have already argued that $m \leq n^2$, the bound $O(m + n)$ is never worse than $O(n^2)$; and it is much better when the underlying graph is *sparse*, with m much smaller than n^2.

Now we consider the ease of accessing the information stored in these two different representations. Recall that in an adjacency matrix we can check in $O(1)$ time if a particular edge (u, v) is present in the graph. In the adjacency list representation, this can take time proportional to the degree $O(n_v)$: we have to follow the pointers on u's adjacency list to see if edge v occurs on the list. On the other hand, if the algorithm is currently looking at a node u, it can read the list of neighbors in constant time per neighbor.

In view of this, the adjacency list is a natural representation for exploring graphs. If the algorithm is currently looking at a node u, it can read this list of neighbors in constant time per neighbor; move to a neighbor v once it encounters it on this list in constant time; and then be ready to read the list associated with node v. The list representation thus corresponds to a physical notion of "exploring" the graph, in which you learn the neighbors of a node u once you arrive at u, and can read them off in constant time per neighbor.

Queues and Stacks

Many algorithms have an inner step in which they need to process a set of elements, such the set of all edges adjacent to a node in a graph, the set of visited nodes in BFS and DFS, or the set of all free men in the Stable Matching algorithm. For this purpose, it is natural to maintain the set of elements to be considered in a linked list, as we have done for maintaining the set of free men in the Stable Matching algorithm.

One important issue that arises is the order in which to consider the elements in such a list. In the Stable Matching algorithm, the order in which we considered the free men did not affect the outcome, although this required a fairly subtle proof to verify. In many other algorithms, such as DFS and BFS, the order in which elements are considered is crucial.

Two of the simplest and most natural options are to maintain a set of elements as either a queue or a stack. A *queue* is a set from which we extract elements in *first-in, first-out* (FIFO) order: we select elements in the same order in which they were added. A *stack* is a set from which we extract elements in *last-in, first-out* (LIFO) order: each time we select an element, we choose the one that was added most recently. Both queues and stacks can be easily implemented via a doubly linked list. In both cases, we always select the first element on our list; the difference is in where we insert a new element. In a queue a new element is added to the end of the list as the last element, while in a stack a new element is placed in the first position on the list. Recall that a doubly linked list has explicit `First` and `Last` pointers to the beginning and end, respectively, so each of these insertions can be done in constant time.

Next we will discuss how to implement the search algorithms of the previous section in linear time. We will see that BFS can be thought of as using a queue to select which node to consider next, while DFS is effectively using a stack.

Implementing Breadth-First Search

The adjacency list data structure is ideal for implementing breadth-first search. The algorithm examines the edges leaving a given node one by one. When we are scanning the edges leaving u and come to an edge (u, v), we need to know whether or not node v has been previously discovered by the search. To make this simple, we maintain an array `Discovered` of length n and set `Discovered`$[v] = true$ as soon as our search first sees v. The algorithm, as described in the previous section, constructs layers of nodes $L_1, L_2, \ldots$, where L_i is the set of nodes at distance i from the source s. To maintain the nodes in a layer L_i, we have a list $L[i]$ for each $i = 0, 1, 2, \ldots$.

```
BFS(s):
  Set Discovered[s] = true and Discovered[v] = false for all other v
  Initialize L[0] to consist of the single element s
  Set the layer counter i = 0
  Set the current BFS tree T = ∅
  While L[i] is not empty
    Initialize an empty list L[i + 1]
    For each node u ∈ L[i]
      Consider each edge (u, v) incident to u
      If Discovered[v] = false then
        Set Discovered[v] = true
        Add edge (u, v) to the tree T
```

```
      Add v to the list L[i + 1]
    Endif
  Endfor
  Increment the layer counter i by one
Endwhile
```

In this implementation it does not matter whether we manage each list $L[i]$ as a queue or a stack, since the algorithm is allowed to consider the nodes in a layer L_i in any order.

(3.11) *The above implementation of the BFS algorithm runs in time $O(m + n)$ (i.e., linear in the input size), if the graph is given by the adjacency list representation.*

Proof. As a first step, it is easy to bound the running time of the algorithm by $O(n^2)$ (a weaker bound than our claimed $O(m + n)$). To see this, note that there are at most n lists $L[i]$ that we need to set up, so this takes $O(n)$ time. Now we need to consider the nodes u on these lists. Each node occurs on at most one list, so the `For` loop runs at most n times over all iterations of the `While` loop. When we consider a node u, we need to look through all edges (u, v) incident to u. There can be at most n such edges, and we spend $O(1)$ time considering each edge. So the total time spent on one iteration of the `For` loop is at most $O(n)$. We've thus concluded that there are at most n iterations of the `For` loop, and that each iteration takes at most $O(n)$ time, so the total time is at most $O(n^2)$.

To get the improved $O(m + n)$ time bound, we need to observe that the `For` loop processing a node u can take less than $O(n)$ time if u has only a few neighbors. As before, let n_u denote the degree of node u, the number of edges incident to u. Now, the time spent in the `For` loop considering edges incident to node u is $O(n_u)$, so the total over all nodes is $O(\sum_{u \in V} n_u)$. Recall from (3.9) that $\sum_{u \in V} n_u = 2m$, and so the total time spent considering edges over the whole algorithm is $O(m)$. We need $O(n)$ additional time to set up lists and manage the array `Discovered`. So the total time spent is $O(m + n)$ as claimed. ■

We described the algorithm using up to n separate lists $L[i]$ for each layer L_i. Instead of all these distinct lists, we can implement the algorithm using a single list L that we maintain as a queue. In this way, the algorithm processes nodes in the order they are first discovered: each time a node is discovered, it is added to the end of the queue, and the algorithm always processes the edges out of the node that is currently first in the queue.

If we maintain the discovered nodes in this order, then all nodes in layer L_i will appear in the queue ahead of all nodes in layer L_{i+1}, for $i = 0, 1, 2 \ldots$. Thus, all nodes in layer L_i will be considered in a contiguous sequence, followed by all nodes in layer L_{i+1}, and so forth. Hence this implementation in terms of a single queue will produce the same result as the BFS implementation above.

Implementing Depth-First Search

We now consider the depth-first search algorithm. In the previous section we presented DFS as a recursive procedure, which is a natural way to specify it. However, it can also be viewed as almost identical to BFS, with the difference that it maintains the nodes to be processed in a stack, rather than in a queue. Essentially, the recursive structure of DFS can be viewed as pushing nodes onto a stack for later processing, while moving on to more freshly discovered nodes. We now show how to implement DFS by maintaining this stack of nodes to be processed explicitly.

In both BFS and DFS, there is a distinction between the act of *discovering* a node v—the first time it is seen, when the algorithm finds an edge leading to v—and the act of *exploring* a node v, when all the incident edges to v are scanned, resulting in the potential discovery of further nodes. The difference between BFS and DFS lies in the way in which discovery and exploration are interleaved.

In BFS, once we started to explore a node u in layer L_i, we added all its newly discovered neighbors to the next layer L_{i+1}, and we deferred actually exploring these neighbors until we got to the processing of layer L_{i+1}. In contrast, DFS is more impulsive: when it explores a node u, it scans the neighbors of u until it finds the first not-yet-explored node v (if any), and then it immediately shifts attention to exploring v.

To implement the exploration strategy of DFS, we first add *all* of the nodes adjacent to u to our list of nodes to be considered, but after doing this we proceed to explore a new neighbor v of u. As we explore v, in turn, we add the neighbors of v to the list we're maintaining, but we do so in stack order, so that these neighbors will be explored before we return to explore the other neighbors of u. We only come back to other nodes adjacent to u when there are no other nodes left.

In addition, we use an array `Explored` analogous to the `Discovered` array we used for BFS. The difference is that we only set `Explored`[v] to be `true` when we scan v's incident edges (when the DFS search is at v), while BFS sets `Discovered`[v] to `true` as soon as v is first discovered. The implementation in full looks as follows.

```
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
       Set Explored[u] = true
       For each edge (u, v) incident to u
         Add v to the stack S
       Endfor
    Endif
  Endwhile
```

There is one final wrinkle to mention. Depth-first search is underspecified, since the adjacency list of a node being explored can be processed in any order. Note that the above algorithm, because it pushes all adjacent nodes onto the stack before considering any of them, in fact processes each adjacency list in the reverse order relative to the recursive version of DFS in the previous section.

(3.12) *The above algorithm implements DFS, in the sense that it visits the nodes in exactly the same order as the recursive DFS procedure in the previous section (except that each adjacency list is processed in reverse order).*

If we want the algorithm to also find the DFS tree, we need to have each node u on the stack S maintain the node that "caused" u to get added to the stack. This can be easily done by using an array `parent` and setting `parent`$[v] = u$ when we add node v to the stack due to edge (u, v). When we mark a node $u \neq s$ as `Explored`, we also can add the edge (u,`parent`$[u]$) to the tree T. Note that a node v may be in the stack S multiple times, as it can be adjacent to multiple nodes u that we explore, and each such node adds a copy of v to the stack S. However, we will only use one of these copies to explore node v, the copy that we add last. As a result, it suffices to maintain one value `parent`$[v]$ for each node v by simply overwriting the value `parent`$[v]$ every time we add a new copy of v to the stack S.

The main step in the algorithm is to add and delete nodes to and from the stack S, which takes $O(1)$ time. Thus, to bound the running time, we need to bound the number of these operations. To count the number of stack operations, it suffices to count the number of nodes added to S, as each node needs to be added once for every time it can be deleted from S.

How many elements ever get added to S? As before, let n_v denote the degree of node v. Node v will be added to the stack S every time one of its n_v adjacent nodes is explored, so the total number of nodes added to S is at

most $\sum_u n_v = 2m$. This proves the desired $O(m + n)$ bound on the running time of DFS.

(3.13) *The above implementation of the DFS algorithm runs in time $O(m + n)$ (i.e., linear in the input size), if the graph is given by the adjacency list representation.*

Finding the Set of All Connected Components

In the previous section we talked about how one can use BFS (or DFS) to find all connected components of a graph. We start with an arbitrary node s, and we use BFS (or DFS) to generate its connected component. We then find a node v (if any) that was not visited by the search from s and iterate, using BFS (or DFS) starting from v to generate its connected component—which, by (3.8), will be disjoint from the component of s. We continue in this way until all nodes have been visited.

Although we earlier expressed the running time of BFS and DFS as $O(m + n)$, where m and n are the total number of edges and nodes in the graph, both BFS and DFS in fact spend work only on edges and nodes in the connected component containing the starting node. (They never see any of the other nodes or edges.) Thus the above algorithm, although it may run BFS or DFS a number of times, only spends a constant amount of work on a given edge or node in the iteration when the connected component it belongs to is under consideration. Hence the overall running time of this algorithm is still $O(m + n)$.

3.4 Testing Bipartiteness: An Application of Breadth-First Search

Recall the definition of a bipartite graph: it is one where the node set V can be partitioned into sets X and Y in such a way that every edge has one end in X and the other end in Y. To make the discussion a little smoother, we can imagine that the nodes in the set X are colored red, and the nodes in the set Y are colored blue. With this imagery, we can say a graph is bipartite if it is possible to color its nodes red and blue so that every edge has one red end and one blue end.

The Problem

In the earlier chapters, we saw examples of bipartite graphs. Here we start by asking: What are some natural examples of a nonbipartite graph, one where no such partition of V is possible?

Clearly a triangle is not bipartite, since we can color one node red, another one blue, and then we can't do anything with the third node. More generally, consider a cycle C of odd length, with nodes numbered $1, 2, 3, \ldots, 2k, 2k+1$. If we color node 1 red, then we must color node 2 blue, and then we must color node 3 red, and so on—coloring odd-numbered nodes red and even-numbered nodes blue. But then we must color node $2k+1$ red, and it has an edge to node 1, which is also red. This demonstrates that there's no way to partition C into red and blue nodes as required. More generally, if a graph G simply *contains* an odd cycle, then we can apply the same argument; thus we have established the following.

(3.14) *If a graph G is bipartite, then it cannot contain an odd cycle.*

It is easy to recognize that a graph is bipartite when appropriate sets X and Y (i.e., red and blue nodes) have actually been identified for us; and in many settings where bipartite graphs arise, this is natural. But suppose we encounter a graph G with no annotation provided for us, and we'd like to determine for ourselves whether it is bipartite—that is, whether there exists a partition into red and blue nodes, as required. How difficult is this? We see from (3.14) that an odd cycle is one simple "obstacle" to a graph's being bipartite. Are there other, more complex obstacles to bipartitness?

Designing the Algorithm

In fact, there is a very simple procedure to test for bipartiteness, and its analysis can be used to show that odd cycles are the *only* obstacle. First we assume the graph G is connected, since otherwise we can first compute its connected components and analyze each of them separately. Next we pick any node $s \in V$ and color it red; there is no loss in doing this, since s must receive some color. It follows that all the neighbors of s must be colored blue, so we do this. It then follows that all the neighbors of *these* nodes must be colored red, their neighbors must be colored blue, and so on, until the whole graph is colored. At this point, either we have a valid red/blue coloring of G, in which every edge has ends of opposite colors, or there is some edge with ends of the same color. In this latter case, it seems clear that there's nothing we could have done: G simply is not bipartite. We now want to argue this point precisely and also work out an efficient way to perform the coloring.

The first thing to notice is that the coloring procedure we have just described is essentially identical to the description of BFS: we move outward from s, coloring nodes as soon as we first encounter them. Indeed, another way to describe the coloring algorithm is as follows: we perform BFS, coloring

s red, all of layer L_1 blue, all of layer L_2 red, and so on, coloring odd-numbered layers blue and even-numbered layers red.

We can implement this on top of BFS, by simply taking the implementation of BFS and adding an extra array `Color` over the nodes. Whenever we get to a step in BFS where we are adding a node v to a list $L[i+1]$, we assign `Color`$[v]$ `= red` if $i+1$ is an even number, and `Color`$[v]$ `= blue` if $i+1$ is an odd number. At the end of this procedure, we simply scan all the edges and determine whether there is any edge for which both ends received the same color. Thus, the total running time for the coloring algorithm is $O(m+n)$, just as it is for BFS.

Analyzing the Algorithm

We now prove a claim that shows this algorithm correctly determines whether G is bipartite, and it also shows that we can find an odd cycle in G whenever it is not bipartite.

(3.15) *Let G be a connected graph, and let $L_1, L_2, \ldots$ be the layers produced by BFS starting at node s. Then exactly one of the following two things must hold.*

(i) There is no edge of G joining two nodes of the same layer. In this case G is a bipartite graph in which the nodes in even-numbered layers can be colored red, and the nodes in odd-numbered layers can be colored blue.

(ii) There is an edge of G joining two nodes of the same layer. In this case, G contains an odd-length cycle, and so it cannot be bipartite.

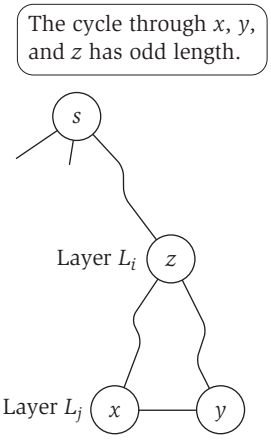

Figure 3.6 If two nodes x and y in the same layer are joined by an edge, then the cycle through x, y, and their lowest common ancestor z has odd length, demonstrating that the graph cannot be bipartite.

Proof. First consider case (i), where we suppose that there is no edge joining two nodes of the same layer. By (3.4), we know that every edge of G joins nodes either in the same layer or in adjacent layers. Our assumption for case (i) is precisely that the first of these two alternatives never happens, so this means that *every* edge joins two nodes in adjacent layers. But our coloring procedure gives nodes in adjacent layers the opposite colors, and so every edge has ends with opposite colors. Thus this coloring establishes that G is bipartite.

Now suppose we are in case (ii); why must G contain an odd cycle? We are told that G contains an edge joining two nodes of the same layer. Suppose this is the edge $e = (x, y)$, with $x, y \in L_j$. Also, for notational reasons, recall that L_0 ("layer 0") is the set consisting of just s. Now consider the BFS tree T produced by our algorithm, and let z be the node whose layer number is as large as possible, subject to the condition that z is an ancestor of both x and y in T; for obvious reasons, we can call z the *lowest common ancestor* of x and y. Suppose $z \in L_i$, where $i < j$. We now have the situation pictured in Figure 3.6. We consider the cycle C defined by following the z-x path in T, then the edge e,

and then the y-z path in T. The length of this cycle is $(j - i) + 1 + (j - i)$, adding the length of its three parts separately; this is equal to $2(j - i) + 1$, which is an odd number. ■

3.5 Connectivity in Directed Graphs

Thus far, we have been looking at problems on undirected graphs; we now consider the extent to which these ideas carry over to the case of directed graphs.

Recall that in a directed graph, the edge (u, v) has a direction: it goes from u to v. In this way, the relationship between u and v is asymmetric, and this has qualitative effects on the structure of the resulting graph. In Section 3.1, for example, we discussed the World Wide Web as an instance of a large, complex directed graph whose nodes are pages and whose edges are hyperlinks. The act of browsing the Web is based on following a sequence of edges in this directed graph; and the directionality is crucial, since it's not generally possible to browse "backwards" by following hyperlinks in the reverse direction.

At the same time, a number of basic definitions and algorithms have natural analogues in the directed case. This includes the adjacency list representation and graph search algorithms such as BFS and DFS. We now discuss these in turn.

Representing Directed Graphs

In order to represent a directed graph for purposes of designing algorithms, we use a version of the adjacency list representation that we employed for undirected graphs. Now, instead of each node having a single list of neighbors, each node has two lists associated with it: one list consists of nodes *to which* it has edges, and a second list consists of nodes *from which* it has edges. Thus an algorithm that is currently looking at a node u can read off the nodes reachable by going one step forward on a directed edge, as well as the nodes that would be reachable if one went one step in the reverse direction on an edge from u.

The Graph Search Algorithms

Breadth-first search and depth-first search are almost the same in directed graphs as they are in undirected graphs. We will focus here on BFS. We start at a node s, define a first layer of nodes to consist of all those to which s has an edge, define a second layer to consist of all additional nodes to which these first-layer nodes have an edge, and so forth. In this way, we discover nodes layer by layer as they are reached in this outward search from s, and the nodes in layer j are precisely those for which the shortest path *from* s has exactly j edges. As in the undirected case, this algorithm performs at most constant work for each node and edge, resulting in a running time of $O(m + n)$.

It is important to understand what this directed version of BFS is computing. In directed graphs, it is possible for a node s to have a path to a node t even though t has no path to s; and what directed BFS is computing is the set of all nodes t with the property that s has a path to t. Such nodes may or may not have paths back to s.

There is a natural analogue of depth-first search as well, which also runs in linear time and computes the same set of nodes. It is again a recursive procedure that tries to explore as deeply as possible, in this case only following edges according to their inherent direction. Thus, when DFS is at a node u, it recursively launches a depth-first search, in order, for each node to which u has an edge.

Suppose that, for a given node s, we wanted the set of nodes with paths to s, rather than the set of nodes to which s has paths. An easy way to do this would be to define a new directed graph, G^{rev}, that we obtain from G simply by reversing the direction of every edge. We could then run BFS or DFS in G^{rev}; a node has a path *from* s in G^{rev} if and only if it has a path *to* s in G.

Strong Connectivity

Recall that a directed graph is *strongly connected* if, for every two nodes u and v, there is a path from u to v and a path from v to u. It's worth also formulating some terminology for the property at the heart of this definition; let's say that two nodes u and v in a directed graph are *mutually reachable* if there is a path from u to v and also a path from v to u. (So a graph is strongly connected if every pair of nodes is mutually reachable.)

Mutual reachability has a number of nice properties, many of them stemming from the following simple fact.

(3.16) *If u and v are mutually reachable, and v and w are mutually reachable, then u and w are mutually reachable.*

Proof. To construct a path from u to w, we first go from u to v (along the path guaranteed by the mutual reachability of u and v), and then on from v to w (along the path guaranteed by the mutual reachability of v and w). To construct a path from w to u, we just reverse this reasoning: we first go from w to v (along the path guaranteed by the mutual reachability of v and w), and then on from v to u (along the path guaranteed by the mutual reachability of u and v). ■

There is a simple linear-time algorithm to test if a directed graph is strongly connected, implicitly based on (3.16). We pick any node s and run BFS in G starting from s. We then also run BFS starting from s in G^{rev}. Now, if one of these two searches fails to reach every node, then clearly G is not strongly connected. But suppose we find that s has a path to every node, and that

every node has a path to s. Then s and v are mutually reachable for every v, and so it follows that *every* two nodes u and v are mutually reachable: s and u are mutually reachable, and s and v are mutually reachable, so by (3.16) we also have that u and v are mutually reachable.

By analogy with connected components in an undirected graph, we can define the *strong component* containing a node s in a directed graph to be the set of all v such that s and v are mutually reachable. If one thinks about it, the algorithm in the previous paragraph is really computing the strong component containing s: we run BFS starting from s both in G and in G^{rev}; the set of nodes reached by *both* searches is the set of nodes with paths to *and* from s, and hence this set is the strong component containing s.

There are further similarities between the notion of connected components in undirected graphs and strong components in directed graphs. Recall that connected components naturally partitioned the graph, since any two were either identical or disjoint. Strong components have this property as well, and for essentially the same reason, based on (3.16).

(3.17) *For any two nodes s and t in a directed graph, their strong components are either identical or disjoint.*

Proof. Consider any two nodes s and t that are mutually reachable; we claim that the strong components containing s and t are identical. Indeed, for any node v, if s and v are mutually reachable, then by (3.16), t and v are mutually reachable as well. Similarly, if t and v are mutually reachable, then again by (3.16), s and v are mutually reachable.

On the other hand, if s and t are not mutually reachable, then there cannot be a node v that is in the strong component of each. For if there were such a node v, then s and v would be mutually reachable, and v and t would be mutually reachable, so from (3.16) it would follow that s and t were mutually reachable. ■

In fact, although we will not discuss the details of this here, with more work it is possible to compute the strong components for all nodes in a total time of $O(m + n)$.

3.6 Directed Acyclic Graphs and Topological Ordering

If an undirected graph has no cycles, then it has an extremely simple structure: each of its connected components is a tree. But it is possible for a directed graph to have no (directed) cycles and still have a very rich structure. For example, such graphs can have a large number of edges: if we start with the node

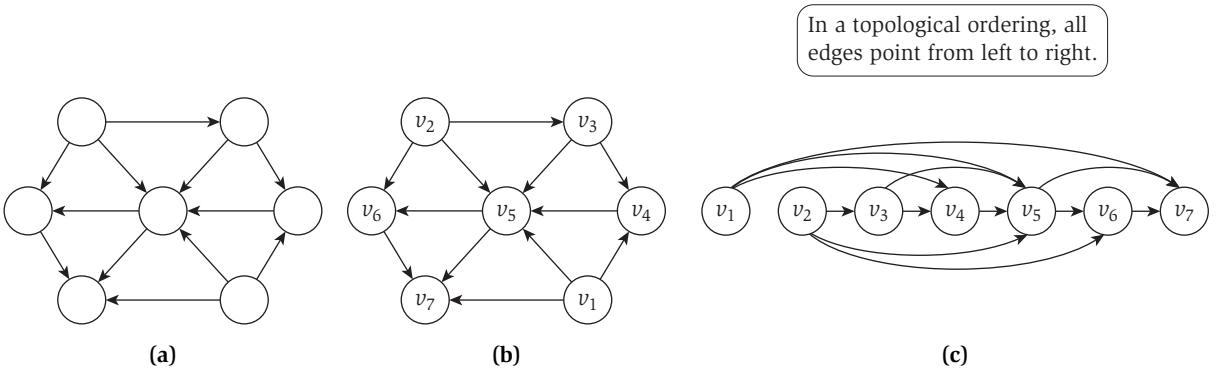

Figure 3.7 (a) A directed acyclic graph. (b) The same DAG with a topological ordering, specified by the labels on each node. (c) A different drawing of the same DAG, arranged so as to emphasize the topological ordering.

set $\{1, 2, \ldots, n\}$ and include an edge (i, j) whenever $i < j$, then the resulting directed graph has $\binom{n}{2}$ edges but no cycles.

If a directed graph has no cycles, we call it—naturally enough—a *directed acyclic graph*, or a *DAG* for short. (The term *DAG* is typically pronounced as a word, not spelled out as an acronym.) In Figure 3.7(a) we see an example of a DAG, although it may take some checking to convince oneself that it really has no directed cycles.

The Problem

DAGs are a very common structure in computer science, because many kinds of dependency networks of the type we discussed in Section 3.1 are acyclic. Thus DAGs can be used to encode *precedence relations* or *dependencies* in a natural way. Suppose we have a set of tasks labeled $\{1, 2, \ldots, n\}$ that need to be performed, and there are dependencies among them stipulating, for certain pairs i and j, that i must be performed before j. For example, the tasks may be courses, with prerequisite requirements stating that certain courses must be taken before others. Or the tasks may correspond to a pipeline of computing jobs, with assertions that the output of job i is used in determining the input to job j, and hence job i must be done before job j.

We can represent such an interdependent set of tasks by introducing a node for each task, and a directed edge (i, j) whenever i must be done before j. If the precedence relation is to be at all meaningful, the resulting graph G must be a DAG. Indeed, if it contained a cycle C, there would be no way to do any of the tasks in C: since each task in C cannot begin until some other one completes, no task in C could ever be done, since none could be done first.

Let's continue a little further with this picture of DAGs as precedence relations. Given a set of tasks with dependencies, it would be natural to seek a valid order in which the tasks could be performed, so that all dependencies are respected. Specifically, for a directed graph G, we say that a *topological ordering* of G is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j), we have $i < j$. In other words, all edges point "forward" in the ordering. A topological ordering on tasks provides an order in which they can be safely performed; when we come to the task v_j, all the tasks that are required to precede it have already been done. In Figure 3.7(b) we've labeled the nodes of the DAG from part (a) with a topological ordering; note that each edge indeed goes from a lower-indexed node to a higher-indexed node.

In fact, we can view a topological ordering of G as providing an immediate "proof" that G has no cycles, via the following.

(3.18) *If G has a topological ordering, then G is a DAG.*

Proof. Suppose, by way of contradiction, that G has a topological ordering $v_1, v_2, \ldots, v_n$, and also has a cycle C. Let v_i be the lowest-indexed node on C, and let v_j be the node on C just before v_i—thus (v_j, v_i) is an edge. But by our choice of i, we have $j > i$, which contradicts the assumption that $v_1, v_2, \ldots, v_n$ was a topological ordering. ■

The proof of acyclicity that a topological ordering provides can be very useful, even visually. In Figure 3.7(c), we have drawn the same graph as in (a) and (b), but with the nodes laid out in the topological ordering. It is immediately clear that the graph in (c) is a DAG since each edge goes from left to right.

Computing a Topological Ordering The main question we consider here is the converse of (3.18): Does every DAG have a topological ordering, and if so, how do we find one efficiently? A method to do this for every DAG would be very useful: it would show that for any precedence relation on a set of tasks without cycles, there is an efficiently computable order in which to perform the tasks.

Designing and Analyzing the Algorithm

In fact, the converse of (3.18) does hold, and we establish this via an efficient algorithm to compute a topological ordering. The key to this lies in finding a way to get started: which node do we put at the beginning of the topological ordering? Such a node v_1 would need to have no incoming edges, since any such incoming edge would violate the defining property of the topological

ordering, that all edges point forward. Thus, we need to prove the following fact.

(3.19) *In every DAG G, there is a node v with no incoming edges.*

Proof. Let G be a directed graph in which every node has at least one incoming edge. We show how to find a cycle in G; this will prove the claim. We pick any node v, and begin following edges backward from v: since v has at least one incoming edge (u, v), we can walk backward to u; then, since u has at least one incoming edge (x, u), we can walk backward to x; and so on. We can continue this process indefinitely, since every node we encounter has an incoming edge. But after $n + 1$ steps, we will have visited some node w twice. If we let C denote the sequence of nodes encountered between successive visits to w, then clearly C forms a cycle. ■

In fact, the existence of such a node v is all we need to produce a topological ordering of G by induction. Specifically, let us claim by induction that every DAG has a topological ordering. This is clearly true for DAGs on one or two nodes. Now suppose it is true for DAGs with up to some number of nodes n. Then, given a DAG G on $n + 1$ nodes, we find a node v with no incoming edges, as guaranteed by (3.19). We place v first in the topological ordering; this is safe, since all edges out of v will point forward. Now $G - \{v\}$ is a DAG, since deleting v cannot create any cycles that weren't there previously. Also, $G - \{v\}$ has n nodes, so we can apply the induction hypothesis to obtain a topological ordering of $G - \{v\}$. We append the nodes of $G - \{v\}$ in this order after v; this is an ordering of G in which all edges point forward, and hence it is a topological ordering.

Thus we have proved the desired converse of (3.18).

(3.20) *If G is a DAG, then G has a topological ordering.*

The inductive proof contains the following algorithm to compute a topological ordering of G.

```
To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of G−{v}
  and append this order after v
```

In Figure 3.8 we show the sequence of node deletions that occurs when this algorithm is applied to the graph in Figure 3.7. The shaded nodes in each iteration are those with no incoming edges; the crucial point, which is what