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,

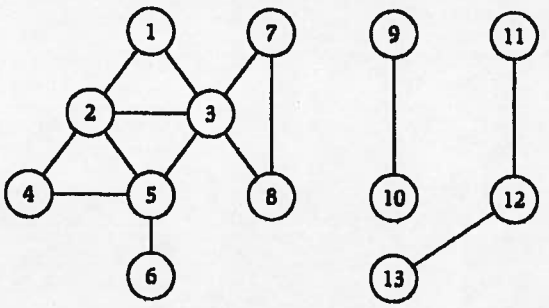

Figure 3.2 In this graph, node 1 has paths to nodes 2 through 8, but not to nodes 9 through 13.

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

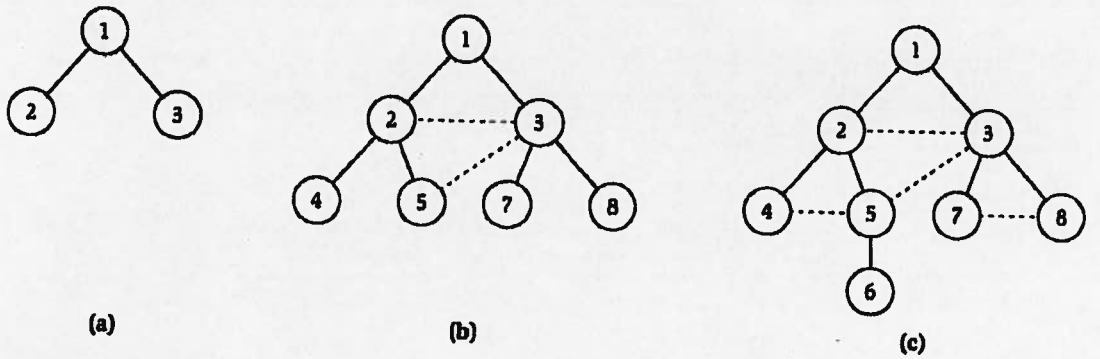

Figure 3.3 The construction of a breadth-first search tree T for the graph in Figure 3.2, with (a), (b), and (c) depicting the successive layers that are added. The solid edges are the edges of T; the dotted edges are in the connected component of G containing node 1, but do not belong to T.

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. ■

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 i $O\left(\sum_{v \in V} n_v\right)$. Now, the sum of the degrees in a graph is a quantity that ofte comes up in the analysis of graph algorithms, so it is useful to work out wha 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 w are scanning the edges leaving u and come to an edge (u, v), we need t know whether or not node v has been previously discovered by the searc To make this simple, we maintain an array `Discovered` of length n and s `Discovered`$[v] = true$ as soon as our search first sees v. The algorithm, described in the previous section, constructs layers of nodes $L_1, L_2, \ldots$, whe L_i is the set of nodes at distance i from the source s. To maintain the nodes 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
  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_l will appear in the queue ahead of all nodes in layer L_{l+1}, for $l = 0, 1, 2, \ldots$. Thus, all nodes in layer L_l will be considered in a contiguous sequence, followed by all nodes in layer L_{l+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_l, we added all its newly discovered neighbors to the next layer L_{l+1}, and we deferred actually exploring these neighbors until we got to the processing of layer L_{l+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.