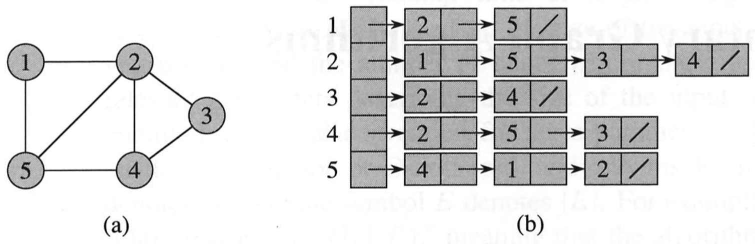
22 Elementary Graph Algorithms

This chapter presents methods for representing a graph and for searching a graph. Searching a graph means systematically following the edges of the graph so as to visit the vertices of the graph. A graph-searching algorithm can discover much about the structure of a graph. Many algorithms begin by searching their input graph to obtain this structural information. Other graph algorithms are organized as simple elaborations of basic graph-searching algorithms. Techniques for searching a graph are at the heart of the field of graph algorithms.

Section 22.1 discusses the two most common computational representations of graphs: as adjacency lists and as adjacency matrices. Section 22.2 presents a simple graph-searching algorithm called breadth-first search and shows how to create a breadth-first tree. Section 22.3 presents depth-first search and proves some standard results about the order in which depth-first search visits vertices. Section 22.4 provides our first real application of depth-first search: topologically sorting a directed acyclic graph. A second application of depth-first search, finding the strongly connected components of a directed graph, is given in Section 22.5.

22.1 Representations of graphs

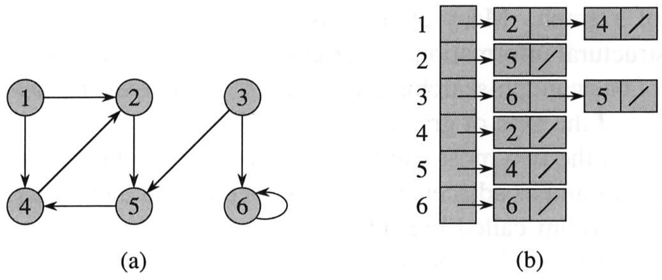
There are two standard ways to represent a graph $G = (V, E)$: as a collection of adjacency lists or as an adjacency matrix. Either way is applicable to both directed and undirected graphs. The adjacency-list representation is usually preferred, because it provides a compact way to represent ***sparse*** graphs—those for which $|E|$ is much less than $|V|^2$. Most of the graph algorithms presented in this book assume that an input graph is represented in adjacency-list form. An adjacency-matrix representation may be preferred, however, when the graph is ***dense***—$|E|$ is close to $|V|^2$—or when we need to be able to tell quickly if there is an edge connecting two given vertices. For example, two of the all-pairs shortest-paths algorithms pre-



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \end{array}$$

(c)

Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G having five vertices and seven edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.



$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 & 1 \\ 4 & 0 & 1 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

(c)

Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G having six vertices and eight edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

sented in Chapter 25 assume that their input graphs are represented by adjacency matrices.

The ***adjacency-list representation*** of a graph $G = (V, E)$ consists of an array *Adj* of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$. That is, $Adj[u]$ consists of all the vertices adjacent to u in G. (Alternatively, it may contain pointers to these vertices.) The vertices in each adjacency list are typically stored in an arbitrary order. Figure 22.1(b) is an adjacency-list representation of the undirected graph in Figure 22.1(a). Similarly, Figure 22.2(b) is an adjacency-list representation of the directed graph in Figure 22.2(a).

If G is a directed graph, the sum of the lengths of all the adjacency lists is $|E|$, since an edge of the form (u, v) is represented by having v appear in $Adj[u]$. If G is an undirected graph, the sum of the lengths of all the adjacency lists is $2\,|E|$, since if (u, v) is an undirected edge, then u appears in v's adjacency list and vice versa.

For both directed and undirected graphs, the adjacency-list representation has the desirable property that the amount of memory it requires is $\Theta(V + E)$.

Adjacency lists can readily be adapted to represent ***weighted graphs***, that is, graphs for which each edge has an associated ***weight***, typically given by a ***weight function*** $w : E \to \mathbf{R}$. For example, let $G = (V, E)$ be a weighted graph with weight function w. The weight $w(u, v)$ of the edge $(u, v) \in E$ is simply stored with vertex v in u's adjacency list. The adjacency-list representation is quite robust in that it can be modified to support many other graph variants.

A potential disadvantage of the adjacency-list representation is that there is no quicker way to determine if a given edge (u, v) is present in the graph than to search for v in the adjacency list $Adj[u]$. This disadvantage can be remedied by an adjacency-matrix representation of the graph, at the cost of using asymptotically more memory. (See Exercise 22.1-8 for suggestions of variations on adjacency lists that permit faster edge lookup.)

For the ***adjacency-matrix representation*** of a graph $G = (V, E)$, we assume that the vertices are numbered $1, 2, \ldots, |V|$ in some arbitrary manner. Then the adjacency-matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E , \\ 0 & \text{otherwise} . \end{cases}$$

Figures 22.1(c) and 22.2(c) are the adjacency matrices of the undirected and directed graphs in Figures 22.1(a) and 22.2(a), respectively. The adjacency matrix of a graph requires $\Theta(V^2)$ memory, independent of the number of edges in the graph.

Observe the symmetry along the main diagonal of the adjacency matrix in Figure 22.1(c). We define the ***transpose*** of a matrix $A = (a_{ij})$ to be the matrix $A^{\mathrm{T}} = (a^{\mathrm{T}}_{ij})$ given by $a^{\mathrm{T}}_{ij} = a_{ji}$. Since in an undirected graph, (u, v) and (v, u) represent the same edge, the adjacency matrix A of an undirected graph is its own transpose: $A = A^{\mathrm{T}}$. In some applications, it pays to store only the entries on and above the diagonal of the adjacency matrix, thereby cutting the memory needed to store the graph almost in half.

Like the adjacency-list representation of a graph, the adjacency-matrix representation can be used for weighted graphs. For example, if $G = (V, E)$ is a weighted graph with edge-weight function w, the weight $w(u, v)$ of the edge $(u, v) \in E$ is simply stored as the entry in row u and column v of the adjacency matrix. If an edge does not exist, a NIL value can be stored as its corresponding matrix entry, though for many problems it is convenient to use a value such as 0 or ∞.

Although the adjacency-list representation is asymptotically at least as efficient as the adjacency-matrix representation, the simplicity of an adjacency matrix may make it preferable when graphs are reasonably small. Moreover, if the graph is unweighted, there is an additional advantage in storage for the adjacency-matrix representation. Rather than using one word of computer memory for each matrix entry, the adjacency matrix uses only one bit per entry.

Exercises

22.1-1

Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex? How long does it take to compute the in-degrees?

22.1-2

Give an adjacency-list representation for a complete binary tree on 7 vertices. Give an equivalent adjacency-matrix representation. Assume that vertices are numbered from 1 to 7 as in a binary heap.

22.1-3

The ***transpose*** of a directed graph $G = (V, E)$ is the graph $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, where $E^{\mathrm{T}} = \{(v, u) \in V \times V : (u, v) \in E\}$. Thus, G^{T} is G with all its edges reversed. Describe efficient algorithms for computing G^{T} from G, for both the adjacency-list and adjacency-matrix representations of G. Analyze the running times of your algorithms.

22.1-4

Given an adjacency-list representation of a multigraph $G = (V, E)$, describe an $O(V + E)$-time algorithm to compute the adjacency-list representation of the "equivalent" undirected graph $G' = (V, E')$, where E' consists of the edges in E with all multiple edges between two vertices replaced by a single edge and with all self-loops removed.

22.1-5

The ***square*** of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$ such that $(u, w) \in E^2$ if and only if for some $v \in V$, both $(u, v) \in E$ and $(v, w) \in E$. That is, G^2 contains an edge between u and w whenever G contains a path with exactly two edges between u and w. Describe efficient algorithms for computing G^2 from G for both the adjacency-list and adjacency-matrix representations of G. Analyze the running times of your algorithms.

22.1-6

When an adjacency-matrix representation is used, most graph algorithms require time $\Omega(V^2)$, but there are some exceptions. Show that determining whether a directed graph G contains a ***universal sink***—a vertex with in-degree $|V| - 1$ and out-degree 0—can be determined in time $O(V)$, given an adjacency matrix for G.

22.1-7

The ***incidence matrix*** of a directed graph $G = (V, E)$ is a $|V| \times |E|$ matrix $B = (b_{ij})$ such that

$$b_{ij} = \begin{cases} -1 & \text{if edge } j \text{ leaves vertex } i \ , \\ 1 & \text{if edge } j \text{ enters vertex } i \ , \\ 0 & \text{otherwise} \ . \end{cases}$$

Describe what the entries of the matrix product BB^{T} represent, where B^{T} is the transpose of B.

22.1-8

Suppose that instead of a linked list, each array entry $Adj[u]$ is a hash table containing the vertices v for which $(u, v) \in E$. If all edge lookups are equally likely, what is the expected time to determine whether an edge is in the graph? What disadvantages does this scheme have? Suggest an alternate data structure for each edge list that solves these problems. Does your alternative have disadvantages compared to the hash table?

22.2 Breadth-first search

Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms. Prim's minimum-spanning-tree algorithm (Section 23.2) and Dijkstra's single-source shortest-paths algorithm (Section 24.3) use ideas similar to those in breadth-first search.

Given a graph $G = (V, E)$ and a distinguished ***source*** vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a "breadth-first tree" with root s that contains all reachable vertices. For any vertex v reachable from s, the path in the breadth-first tree from s to v corresponds to a "shortest path" from s to v in G, that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs.

Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$.

To keep track of progress, breadth-first search colors each vertex white, gray, or black. All vertices start out white and may later become gray and then black. A vertex is ***discovered*** the first time it is encountered during the search, at which time it becomes nonwhite. Gray and black vertices, therefore, have been discovered, but breadth-first search distinguishes between them to ensure that the search proceeds in a breadth-first manner. If $(u, v) \in E$ and vertex u is black, then vertex v is either gray or black; that is, all vertices adjacent to black vertices have been discovered. Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.

Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex s. Whenever a white vertex v is discovered in the course of scanning the adjacency list of an already discovered vertex u, the vertex v and the edge (u, v) are added to the tree. We say that u is the ***predecessor*** or ***parent*** of v in the breadth-first tree. Since a vertex is discovered at most once, it has at most one parent. Ancestor and descendant relationships in the breadth-first tree are defined relative to the root s as usual: if u is on a path in the tree from the root s to vertex v, then u is an ancestor of v and v is a descendant of u.

The breadth-first-search procedure BFS below assumes that the input graph $G = (V, E)$ is represented using adjacency lists. It maintains several additional data structures with each vertex in the graph. The color of each vertex $u \in V$ is stored in the variable *color*$[u]$, and the predecessor of u is stored in the variable $\pi[u]$. If u has no predecessor (for example, if $u = s$ or u has not been discovered), then $\pi[u] =$ NIL. The distance from the source s to vertex u computed by the algorithm is stored in $d[u]$. The algorithm also uses a first-in, first-out queue Q (see Section 10.1) to manage the set of gray vertices.

```
BFS(G, s)
for each vertex $u \in V[G] - \{s\}$
     do $color[u] \leftarrow$ WHITE
        $d[u] \leftarrow \infty$
        $\pi[u] \leftarrow$ NIL
$color[s] \leftarrow$ GRAY
$d[s] \leftarrow 0$
$\pi[s] \leftarrow$ NIL
$Q \leftarrow \emptyset$
ENQUEUE(Q, s)
while $Q \neq \emptyset$
     do $u \leftarrow$ DEQUEUE(Q)
        for each $v \in Adj[u]$
             do if $color[v] =$ WHITE
                  then $color[v] \leftarrow$ GRAY
                       $d[v] \leftarrow d[u] + 1$
                       $\pi[v] \leftarrow u$
                       ENQUEUE(Q, v)
        $color[u] \leftarrow$ BLACK
```

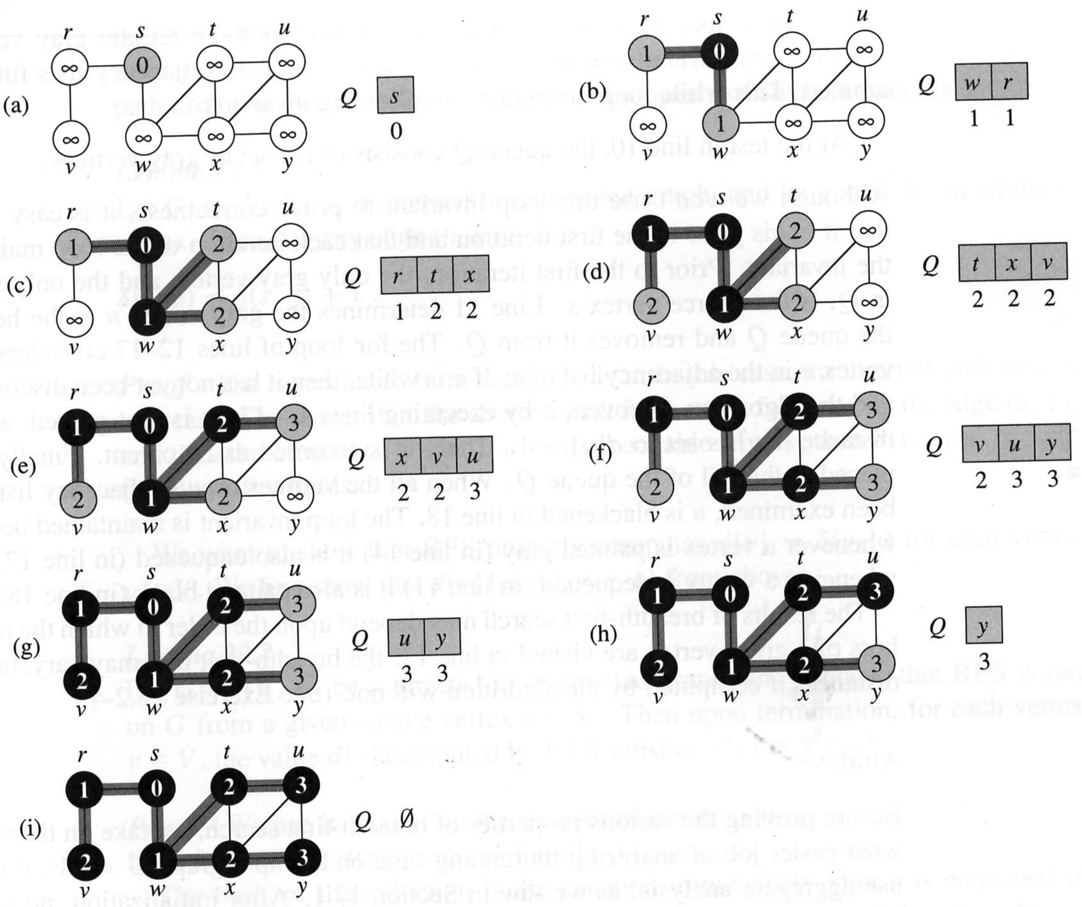


Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in the queue.

Figure 22.3 illustrates the progress of BFS on a sample graph.

The procedure BFS works as follows. Lines 1–4 paint every vertex white, set $d[u]$ to be infinity for each vertex u, and set the parent of every vertex to be NIL. Line 5 paints the source vertex s gray, since it is considered to be discovered when the procedure begins. Line 6 initializes $d[s]$ to 0, and line 7 sets the predecessor of the source to be NIL. Lines 8–9 initialize Q to the queue containing just the vertex s.

The **while** loop of lines 10–18 iterates as long as there remain gray vertices, which are discovered vertices that have not yet had their adjacency lists fully examined. This **while** loop maintains the following invariant:

At the test in line 10, the queue Q consists of the set of gray vertices.

Although we won't use this loop invariant to prove correctness, it is easy to see that it holds prior to the first iteration and that each iteration of the loop maintains the invariant. Prior to the first iteration, the only gray vertex, and the only vertex in Q, is the source vertex s. Line 11 determines the gray vertex u at the head of the queue Q and removes it from Q. The **for** loop of lines 12–17 considers each vertex v in the adjacency list of u. If v is white, then it has not yet been discovered, and the algorithm discovers it by executing lines 14–17. It is first grayed, and its distance $d[v]$ is set to $d[u] + 1$. Then, u is recorded as its parent. Finally, it is placed at the tail of the queue Q. When all the vertices on u's adjacency list have been examined, u is blackened in line 18. The loop invariant is maintained because whenever a vertex is painted gray (in line 14) it is also enqueued (in line 17), and whenever a vertex is dequeued (in line 11) it is also painted black (in line 18).

The results of breadth-first search may depend upon the order in which the neighbors of a given vertex are visited in line 12: the breadth-first tree may vary, but the distances d computed by the algorithm will not. (See Exercise 22.2-4.)

Analysis

Before proving the various properties of breadth-first search, we take on the somewhat easier job of analyzing its running time on an input graph $G = (V, E)$. We use aggregate analysis, as we saw in Section 17.1. After initialization, no vertex is ever whitened, and thus the test in line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once. The operations of enqueuing and dequeuing take $O(1)$ time, so the total time devoted to queue operations is $O(V)$. Because the adjacency list of each vertex is scanned only when the vertex is dequeued, each adjacency list is scanned at most once. Since the sum of the lengths of all the adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(E)$. The overhead for initialization is $O(V)$, and thus the total running time of BFS is $O(V + E)$. Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G.

Shortest paths

At the beginning of this section, we claimed that breadth-first search finds the distance to each reachable vertex in a graph $G = (V, E)$ from a given source vertex $s \in V$. Define the ***shortest-path distance*** $\delta(s, v)$ from s to v as the minimum number of edges in any path from vertex s to vertex v; if there is no path from s to v, then $\delta(s, v) = \infty$. A path of length $\delta(s, v)$ from s to v is said to be a ***shortest path***[1] from s to v. Before showing that breadth-first search actually computes shortest-path distances, we investigate an important property of shortest-path distances.

Lemma 22.1
Let $G = (V, E)$ be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$,

$$\delta(s, v) \le \delta(s, u) + 1 .$$

Proof If u is reachable from s, then so is v. In this case, the shortest path from s to v cannot be longer than the shortest path from s to u followed by the edge (u, v), and thus the inequality holds. If u is not reachable from s, then $\delta(s, u) = \infty$, and the inequality holds. ■

We want to show that BFS properly computes $d[v] = \delta(s, v)$ for each vertex $v \in V$. We first show that $d[v]$ bounds $\delta(s, v)$ from above.

Lemma 22.2
Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then upon termination, for each vertex $v \in V$, the value $d[v]$ computed by BFS satisfies $d[v] \ge \delta(s, v)$.

Proof We use induction on the number of ENQUEUE operations. Our inductive hypothesis is that $d[v] \ge \delta(s, v)$ for all $v \in V$.

The basis of the induction is the situation immediately after s is enqueued in line 9 of BFS. The inductive hypothesis holds here, because $d[s] = 0 = \delta(s, s)$ and $d[v] = \infty \ge \delta(s, v)$ for all $v \in V - \{s\}$.

For the inductive step, consider a white vertex v that is discovered during the search from a vertex u. The inductive hypothesis implies that $d[u] \ge \delta(s, u)$. From the assignment performed by line 15 and from Lemma 22.1, we obtain

$$\begin{aligned} d[v] &= d[u] + 1 \\ &\ge \delta(s, u) + 1 \\ &\ge \delta(s, v) . \end{aligned}$$

[1] In Chapters 24 and 25, we shall generalize our study of shortest paths to weighted graphs, in which every edge has a real-valued weight and the weight of a path is the sum of the weights of its constituent edges. The graphs considered in the present chapter are unweighted or, equivalently, all edges have unit weight.

Vertex v is then enqueued, and it is never enqueued again because it is also grayed and the **then** clause of lines 14–17 is executed only for white vertices. Thus, the value of $d[v]$ never changes again, and the inductive hypothesis is maintained. ■

To prove that $d[v] = \delta(s, v)$, we must first show more precisely how the queue Q operates during the course of BFS. The next lemma shows that at all times, there are at most two distinct d values in the queue.

Lemma 22.3
Suppose that during the execution of BFS on a graph $G = (V, E)$, the queue Q contains the vertices $\langle v_1, v_2, \ldots, v_r \rangle$, where v_1 is the head of Q and v_r is the tail. Then, $d[v_r] \le d[v_1] + 1$ and $d[v_i] \le d[v_{i+1}]$ for $i = 1, 2, \ldots, r - 1$.

Proof The proof is by induction on the number of queue operations. Initially, when the queue contains only s, the lemma certainly holds.

For the inductive step, we must prove that the lemma holds after both dequeuing and enqueuing a vertex. If the head v_1 of the queue is dequeued, v_2 becomes the new head. (If the queue becomes empty, then the lemma holds vacuously.) By the inductive hypothesis, $d[v_1] \le d[v_2]$. But then we have $d[v_r] \le d[v_1] + 1 \le d[v_2] + 1$, and the remaining inequalities are unaffected. Thus, the lemma follows with v_2 as the head.

Enqueuing a vertex requires closer examination of the code. When we enqueue a vertex v in line 17 of BFS, it becomes v_{r+1}. At that time, we have already removed vertex u, whose adjacency list is currently being scanned, from the queue Q, and by the inductive hypothesis, the new head v_1 has $d[v_1] \ge d[u]$. Thus, $d[v_{r+1}] = d[v] = d[u] + 1 \le d[v_1] + 1$. From the inductive hypothesis, we also have $d[v_r] \le d[u] + 1$, and so $d[v_r] \le d[u] + 1 = d[v] = d[v_{r+1}]$, and the remaining inequalities are unaffected. Thus, the lemma follows when v is enqueued. ■

The following corollary shows that the d values at the time that vertices are enqueued are monotonically increasing over time.

Corollary 22.4
Suppose that vertices v_i and v_j are enqueued during the execution of BFS, and that v_i is enqueued before v_j. Then $d[v_i] \le d[v_j]$ at the time that v_j is enqueued.

Proof Immediate from Lemma 22.3 and the property that each vertex receives a finite d value at most once during the course of BFS. ■

We can now prove that breadth-first search correctly finds shortest-path distances.

Theorem 22.5 (Correctness of breadth-first search)
Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source s, and upon termination, $d[v] = \delta(s, v)$ for all $v \in V$. Moreover, for any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $\pi[v]$ followed by the edge $(\pi[v], v)$.

Proof Assume, for the purpose of contradiction, that some vertex receives a d value not equal to its shortest path distance. Let v be the vertex with minimum $\delta(s, v)$ that receives such an incorrect d value; clearly $v \neq s$. By Lemma 22.2, $d[v] \geq \delta(s, v)$, and thus we have that $d[v] > \delta(s, v)$. Vertex v must be reachable from s, for if it is not, then $\delta(s, v) = \infty \geq d[v]$. Let u be the vertex immediately preceding v on a shortest path from s to v, so that $\delta(s, v) = \delta(s, u) + 1$. Because $\delta(s, u) < \delta(s, v)$, and because of how we chose v, we have $d[u] = \delta(s, u)$. Putting these properties together, we have

$$d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1 \tag{22.1}$$

Now consider the time when BFS chooses to dequeue vertex u from Q in line 11. At this time, vertex v is either white, gray, or black. We shall show that in each of these cases, we derive a contradiction to inequality (22.1). If v is white, then line 15 sets $d[v] = d[u] + 1$, contradicting inequality (22.1). If v is black, then it was already removed from the queue and, by Corollary 22.4, we have $d[v] \leq d[u]$, again contradicting inequality (22.1). If v is gray, then it was painted gray upon dequeuing some vertex w, which was removed from Q earlier than u and for which $d[v] = d[w] + 1$. By Corollary 22.4, however, $d[w] \leq d[u]$, and so we have $d[v] \leq d[u] + 1$, once again contradicting inequality (22.1).

Thus we conclude that $d[v] = \delta(s, v)$ for all $v \in V$. All vertices reachable from s must be discovered, for if they were not, they would have infinite d values. To conclude the proof of the theorem, observe that if $\pi[v] = u$, then $d[v] = d[u] + 1$. Thus, we can obtain a shortest path from s to v by taking a shortest path from s to $\pi[v]$ and then traversing the edge $(\pi[v], v)$. ■

Breadth-first trees

The procedure BFS builds a breadth-first tree as it searches the graph, as illustrated in Figure 22.3. The tree is represented by the π field in each vertex. More formally, for a graph $G = (V, E)$ with source s, we define the ***predecessor subgraph*** of G as $G_\pi = (V_\pi, E_\pi)$, where

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$$

and

$$E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\} .$$

The predecessor subgraph G_π is a ***breadth-first tree*** if V_π consists of the vertices reachable from s and, for all $v \in V_\pi$, there is a unique simple path from s to v in G_π that is also a shortest path from s to v in G. A breadth-first tree is in fact a tree, since it is connected and $|E_\pi| = |V_\pi| - 1$ (see Theorem B.2). The edges in E_π are called ***tree edges***.

After BFS has been run from a source s on a graph G, the following lemma shows that the predecessor subgraph is a breadth-first tree.

Lemma 22.6
When applied to a directed or undirected graph $G = (V, E)$, procedure BFS constructs π so that the predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ is a breadth-first tree.

Proof Line 16 of BFS sets $\pi[v] = u$ if and only if $(u, v) \in E$ and $\delta(s, v) < \infty$—that is, if v is reachable from s—and thus V_π consists of the vertices in V reachable from s. Since G_π forms a tree, by Theorem B.2, it contains a unique path from s to each vertex in V_π. By applying Theorem 22.5 inductively, we conclude that every such path is a shortest path. ■

The following procedure prints out the vertices on a shortest path from s to v, assuming that BFS has already been run to compute the shortest-path tree.

```
PRINT-PATH(G, s, v)
1  if v = s
2     then print s
3     else if π[v] = NIL
4              then print "no path from" s "to" v "exists"
5              else PRINT-PATH(G, s, π[v])
6                   print v
```

This procedure runs in time linear in the number of vertices in the path printed, since each recursive call is for a path one vertex shorter.

Exercises

22.2-1
Show the d and π values that result from running breadth-first search on the directed graph of Figure 22.2(a), using vertex 3 as the source.

22.2-2
Show the d and π values that result from running breadth-first search on the undirected graph of Figure 22.3, using vertex u as the source.

22.2-3

What is the running time of BFS if its input graph is represented by an adjacency matrix and the algorithm is modified to handle this form of input?

22.2-4

Argue that in a breadth-first search, the value $d[u]$ assigned to a vertex u is independent of the order in which the vertices in each adjacency list are given. Using Figure 22.3 as an example, show that the breadth-first tree computed by BFS can depend on the ordering within adjacency lists.

22.2-5

Give an example of a directed graph $G = (V, E)$, a source vertex $s \in V$, and a set of tree edges $E_\pi \subseteq E$ such that for each vertex $v \in V$, the unique path in the graph (V, E_π) from s to v is a shortest path in G, yet the set of edges E_π cannot be produced by running BFS on G, no matter how the vertices are ordered in each adjacency list.

22.2-6

There are two types of professional wrestlers: "good guys" and "bad guys." Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n professional wrestlers and we have a list of r pairs of wrestlers for which there are rivalries. Give an $O(n + r)$-time algorithm that determines whether it is possible to designate some of the wrestlers as good guys and the remainder as bad guys such that each rivalry is between a good guy and a bad guy. If it is possible to perform such a designation, your algorithm should produce it.

22.2-7 ⋆

The ***diameter*** of a tree $T = (V, E)$ is given by

$$\max_{u,v \in V} \delta(u, v) ;$$

that is, the diameter is the largest of all shortest-path distances in the tree. Give an efficient algorithm to compute the diameter of a tree, and analyze the running time of your algorithm.

22.2-8

Let $G = (V, E)$ be a connected, undirected graph. Give an $O(V + E)$-time algorithm to compute a path in G that traverses each edge in E exactly once in each direction. Describe how you can find your way out of a maze if you are given a large supply of pennies.

22.3 Depth-first search

The strategy followed by depth-first search is, as its name implies, to search "deeper" in the graph whenever possible. In depth-first search, edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it. When all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source. This entire process is repeated until all vertices are discovered.

As in breadth-first search, whenever a vertex v is discovered during a scan of the adjacency list of an already discovered vertex u, depth-first search records this event by setting v's predecessor field $\pi[v]$ to u. Unlike breadth-first search, whose predecessor subgraph forms a tree, the predecessor subgraph produced by a depth-first search may be composed of several trees, because the search may be repeated from multiple sources.[2] The ***predecessor subgraph*** of a depth-first search is therefore defined slightly differently from that of a breadth-first search: we let $G_\pi = (V, E_\pi)$, where

$$E_\pi = \{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq \text{NIL}\} .$$

The predecessor subgraph of a depth-first search forms a ***depth-first forest*** composed of several ***depth-first trees***. The edges in E_π are called ***tree edges***.

As in breadth-first search, vertices are colored during the search to indicate their state. Each vertex is initially white, is grayed when it is ***discovered*** in the search, and is blackened when it is ***finished***, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

Besides creating a depth-first forest, depth-first search also ***timestamps*** each vertex. Each vertex v has two timestamps: the first timestamp $d[v]$ records when v is first discovered (and grayed), and the second timestamp $f[v]$ records when the search finishes examining v's adjacency list (and blackens v). These timestamps

[2]It may seem arbitrary that breadth-first search is limited to only one source whereas depth-first search may search from multiple sources. Although conceptually, breadth-first search could proceed from multiple sources and depth-first search could be limited to one source, our approach reflects how the results of these searches are typically used. Breadth-first search is usually employed to find shortest-path distances (and the associated predecessor subgraph) from a given source. Depth-first search is often a subroutine in another algorithm, as we shall see later in this chapter.

are used in many graph algorithms and are generally helpful in reasoning about the behavior of depth-first search.

The procedure DFS below records when it discovers vertex u in the variable $d[u]$ and when it finishes vertex u in the variable $f[u]$. These timestamps are integers between 1 and $2|V|$, since there is one discovery event and one finishing event for each of the $|V|$ vertices. For every vertex u,

$$d[u] < f[u] . \qquad (22.2)$$

Vertex u is WHITE before time $d[u]$, GRAY between time $d[u]$ and time $f[u]$, and BLACK thereafter.

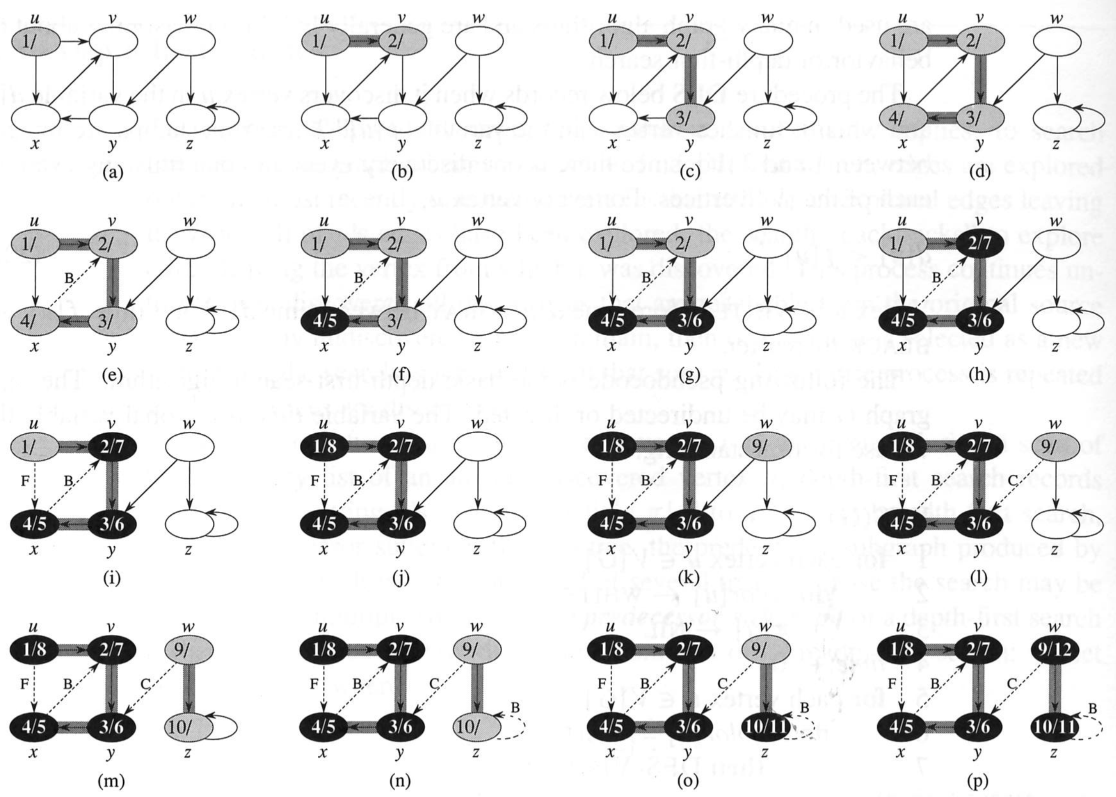
The following pseudocode is the basic depth-first-search algorithm. The input graph G may be undirected or directed. The variable *time* is a global variable that we use for timestamping.

```
DFS(G)
1  for each vertex $u \in V[G]$
2       do $color[u] \leftarrow$ WHITE
3          $\pi[u] \leftarrow$ NIL
4  $time \leftarrow 0$
5  for each vertex $u \in V[G]$
6       do if $color[u] =$ WHITE
7             then DFS-VISIT($u$)
```

```
DFS-VISIT(u)
1  $color[u] \leftarrow$ GRAY            ▷ White vertex $u$ has just been discovered.
2  $time \leftarrow time + 1$
3  $d[u] \leftarrow time$
4  for each $v \in Adj[u]$           ▷ Explore edge $(u, v)$.
5       do if $color[v] =$ WHITE
6             then $\pi[v] \leftarrow u$
7                  DFS-VISIT($v$)
8  $color[u] \leftarrow$ BLACK          ▷ Blacken $u$; it is finished.
9  $f[u] \leftarrow time \leftarrow time + 1$
```

Figure 22.4 illustrates the progress of DFS on the graph shown in Figure 22.2.

Procedure DFS works as follows. Lines 1–3 paint all vertices white and initialize their π fields to NIL. Line 4 resets the global time counter. Lines 5–7 check each vertex in V in turn and, when a white vertex is found, visit it using DFS-VISIT. Every time DFS-VISIT(u) is called in line 7, vertex u becomes the root of a new tree in the depth-first forest. When DFS returns, every vertex u has been assigned a ***discovery time*** $d[u]$ and a ***finishing time*** $f[u]$.



u v w x y z
(a) (b) (c) (d) (e) (f) (g) (h) (i) (j) (k) (l) (m) (n) (o) (p)
1/ 2/ 3/ 4/ 4/5 3/6 2/7 1/8 9/ 10/ 10/11 9/12
B F C

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time.

In each call DFS-VISIT(u), vertex u is initially white. Line 1 paints u gray, line 2 increments the global variable *time*, and line 3 records the new value of *time* as the discovery time $d[u]$. Lines 4–7 examine each vertex v adjacent to u and recursively visit v if it is white. As each vertex $v \in Adj[u]$ is considered in line 4, we say that edge (u, v) is ***explored*** by the depth-first search. Finally, after every edge leaving u has been explored, lines 8–9 paint u black and record the finishing time in $f[u]$.

Note that the results of depth-first search may depend upon the order in which the vertices are examined in line 5 of DFS, and upon the order in which the neighbors of a vertex are visited in line 4 of DFS-VISIT. These different visitation orders tend not to cause problems in practice, as *any* depth-first search result can usually be used effectively, with essentially equivalent results.

What is the running time of DFS? The loops on lines 1–3 and lines 5–7 of DFS take time $\Theta(V)$, exclusive of the time to execute the calls to DFS-VISIT. As we did for breadth-first search, we use aggregate analysis. The procedure DFS-VISIT is called exactly once for each vertex $v \in V$, since DFS-VISIT is invoked only on white vertices and the first thing it does is paint the vertex gray. During an execution of DFS-VISIT(v), the loop on lines 4–7 is executed $|Adj[v]|$ times. Since

$$\sum_{v \in V} |Adj[v]| = \Theta(E) ,$$

the total cost of executing lines 4–7 of DFS-VISIT is $\Theta(E)$. The running time of DFS is therefore $\Theta(V + E)$.

Properties of depth-first search

Depth-first search yields valuable information about the structure of a graph. Perhaps the most basic property of depth-first search is that the predecessor subgraph G_π does indeed form a forest of trees, since the structure of the depth-first trees exactly mirrors the structure of recursive calls of DFS-VISIT. That is, $u = \pi[v]$ if and only if DFS-VISIT(v) was called during a search of u's adjacency list. Additionally, vertex v is a descendant of vertex u in the depth-first forest if and only if v is discovered during the time in which u is gray.

Another important property of depth-first search is that discovery and finishing times have ***parenthesis structure***. If we represent the discovery of vertex u with a left parenthesis "$(u$" and represent its finishing by a right parenthesis "$u)$", then the history of discoveries and finishings makes a well-formed expression in the sense that the parentheses are properly nested. For example, the depth-first search of Figure 22.5(a) corresponds to the parenthesization shown in Figure 22.5(b). Another way of stating the condition of parenthesis structure is given in the following theorem.

Theorem 22.7 (Parenthesis theorem)

In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

- the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,
- the interval $[d[u], f[u]]$ is contained entirely within the interval $[d[v], f[v]]$, and u is a descendant of v in a depth-first tree, or
- the interval $[d[v], f[v]]$ is contained entirely within the interval $[d[u], f[u]]$, and v is a descendant of u in a depth-first tree.

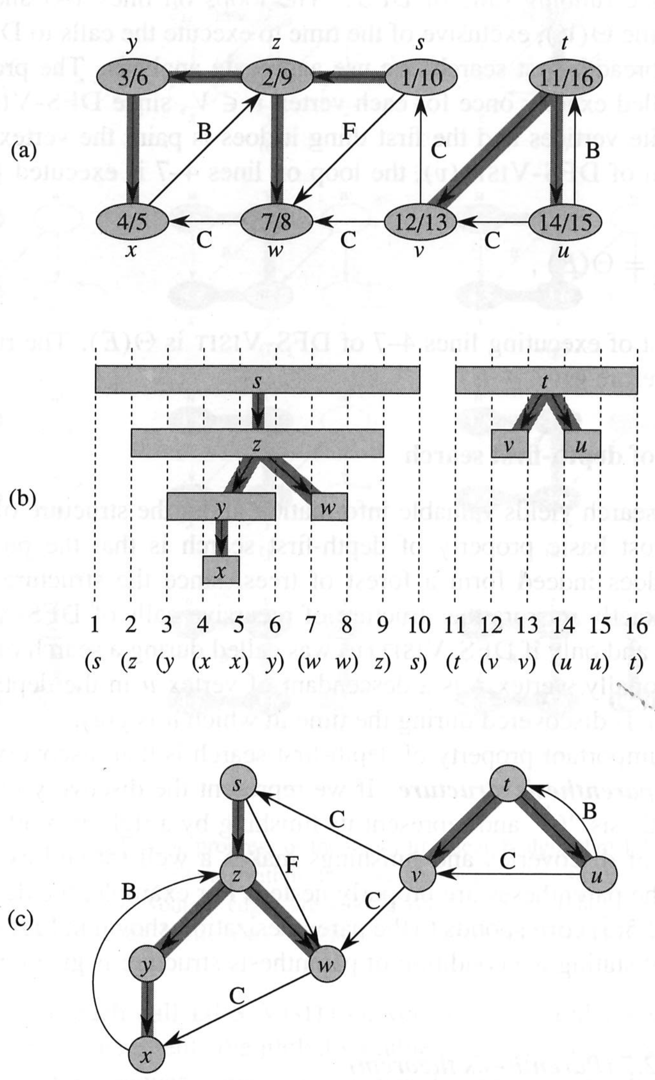


Figure 22.5 Properties of depth-first search. **(a)** The result of a depth-first search of a directed graph. Vertices are timestamped and edge types are indicated as in Figure 22.4. **(b)** Intervals for the discovery time and finishing time of each vertex correspond to the parenthesization shown. Each rectangle spans the interval given by the discovery and finishing times of the corresponding vertex. Tree edges are shown. If two intervals overlap, then one is nested within the other, and the vertex corresponding to the smaller interval is a descendant of the vertex corresponding to the larger. **(c)** The graph of part (a) redrawn with all tree and forward edges going down within a depth-first tree and all back edges going up from a descendant to an ancestor.

Proof We begin with the case in which $d[u] < d[v]$. There are two subcases to consider, according to whether $d[v] < f[u]$ or not. The first subcase occurs when $d[v] < f[u]$, so v was discovered while u was still gray. This implies that v is a descendant of u. Moreover, since v was discovered more recently than u, all of its outgoing edges are explored, and v is finished, before the search returns to and finishes u. In this case, therefore, the interval $[d[v], f[v]]$ is entirely contained within the interval $[d[u], f[u]]$. In the other subcase, $f[u] < d[v]$, and inequality (22.2) implies that the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are disjoint. Because the intervals are disjoint, neither vertex was discovered while the other was gray, and so neither vertex is a descendant of the other.

The case in which $d[v] < d[u]$ is similar, with the roles of u and v reversed in the above argument. ■

Corollary 22.8 (Nesting of descendants' intervals)
Vertex v is a proper descendant of vertex u in the depth-first forest for a (directed or undirected) graph G if and only if $d[u] < d[v] < f[v] < f[u]$.

Proof Immediate from Theorem 22.7. ■

The next theorem gives another important characterization of when one vertex is a descendant of another in the depth-first forest.

Theorem 22.9 (White-path theorem)
In a depth-first forest of a (directed or undirected) graph $G = (V, E)$, vertex v is a descendant of vertex u if and only if at the time $d[u]$ that the search discovers u, vertex v can be reached from u along a path consisting entirely of white vertices.

Proof $\Rightarrow$: Assume that v is a descendant of u. Let w be any vertex on the path between u and v in the depth-first tree, so that w is a descendant of u. By Corollary 22.8, $d[u] < d[w]$, and so w is white at time $d[u]$.

$\Leftarrow$: Suppose that vertex v is reachable from u along a path of white vertices at time $d[u]$, but v does not become a descendant of u in the depth-first tree. Without loss of generality, assume that every other vertex along the path becomes a descendant of u. (Otherwise, let v be the closest vertex to u along the path that doesn't become a descendant of u.) Let w be the predecessor of v in the path, so that w is a descendant of u (w and u may in fact be the same vertex) and, by Corollary 22.8, $f[w] \leq f[u]$. Note that v must be discovered after u is discovered, but before w is finished. Therefore, $d[u] < d[v] < f[w] \leq f[u]$. Theorem 22.7 then implies that the interval $[d[v], f[v]]$ is contained entirely within the interval $[d[u], f[u]]$. By Corollary 22.8, v must after all be a descendant of u. ■

Classification of edges

Another interesting property of depth-first search is that the search can be used to classify the edges of the input graph $G = (V, E)$. This edge classification can be used to glean important information about a graph. For example, in the next section, we shall see that a directed graph is acyclic if and only if a depth-first search yields no "back" edges (Lemma 22.11).

We can define four edge types in terms of the depth-first forest G_π produced by a depth-first search on G.

1. ***Tree edges*** are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
2. ***Back edges*** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. Self-loops, which may occur in directed graphs, are considered to be back edges.
3. ***Forward edges*** are those nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
4. ***Cross edges*** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

In Figures 22.4 and 22.5, edges are labeled to indicate their type. Figure 22.5(c) also shows how the graph of Figure 22.5(a) can be redrawn so that all tree and forward edges head downward in a depth-first tree and all back edges go up. Any graph can be redrawn in this fashion.

The DFS algorithm can be modified to classify edges as it encounters them. The key idea is that each edge (u, v) can be classified by the color of the vertex v that is reached when the edge is first explored (except that forward and cross edges are not distinguished):

1. WHITE indicates a tree edge,
2. GRAY indicates a back edge, and
3. BLACK indicates a forward or cross edge.

The first case is immediate from the specification of the algorithm. For the second case, observe that the gray vertices always form a linear chain of descendants corresponding to the stack of active DFS-VISIT invocations; the number of gray vertices is one more than the depth in the depth-first forest of the vertex most recently discovered. Exploration always proceeds from the deepest gray vertex, so an edge that reaches another gray vertex reaches an ancestor. The third case handles the remaining possibility; it can be shown that such an edge (u, v) is a forward edge if $d[u] < d[v]$ and a cross edge if $d[u] > d[v]$. (See Exercise 22.3-4.)

In an undirected graph, there may be some ambiguity in the type classification, since (u, v) and (v, u) are really the same edge. In such a case, the edge is classified as the *first* type in the classification list that applies. Equivalently (see Exercise 22.3-5), the edge is classified according to whichever of (u, v) or (v, u) is encountered first during the execution of the algorithm.

We now show that forward and cross edges never occur in a depth-first search of an undirected graph.

Theorem 22.10
In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge.

Proof Let (u, v) be an arbitrary edge of G, and suppose without loss of generality that $d[u] < d[v]$. Then, v must be discovered and finished before we finish u (while u is gray), since v is on u's adjacency list. If the edge (u, v) is explored first in the direction from u to v, then v is undiscovered (white) until that time, for otherwise we would have explored this edge already in the direction from v to u. Thus, (u, v) becomes a tree edge. If (u, v) is explored first in the direction from v to u, then (u, v) is a back edge, since u is still gray at the time the edge is first explored. ■

We shall see several applications of these theorems in the following sections.

Exercises

22.3-1
Make a 3-by-3 chart with row and column labels WHITE, GRAY, and BLACK. In each cell (i, j), indicate whether, at any point during a depth-first search of a directed graph, there can be an edge from a vertex of color i to a vertex of color j. For each possible edge, indicate what edge types it can be. Make a second such chart for depth-first search of an undirected graph.

22.3-2
Show how depth-first search works on the graph of Figure 22.6. Assume that the **for** loop of lines 5–7 of the DFS procedure considers the vertices in alphabetical order, and assume that each adjacency list is ordered alphabetically. Show the discovery and finishing times for each vertex, and show the classification of each edge.

22.3-3
Show the parenthesis structure of the depth-first search shown in Figure 22.4.

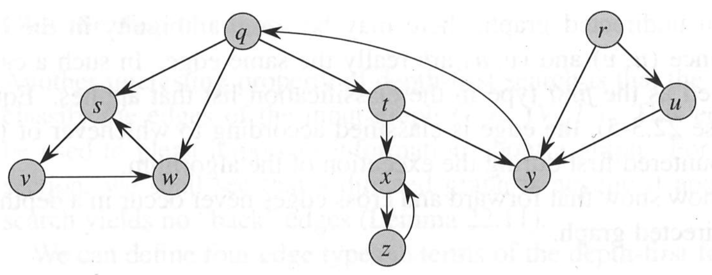


Figure 22.6 A directed graph for use in Exercises 22.3-2 and 22.5-2.

22.3-4
Show that edge (u, v) is

a. a tree edge or forward edge if and only if $d[u] < d[v] < f[v] < f[u]$,

b. a back edge if and only if $d[v] < d[u] < f[u] < f[v]$, and

c. a cross edge if and only if $d[v] < f[v] < d[u] < f[u]$.

22.3-5
Show that in an undirected graph, classifying an edge (u, v) as a tree edge or a back edge according to whether (u, v) or (v, u) is encountered first during the depth-first search is equivalent to classifying it according to the priority of types in the classification scheme.

22.3-6
Rewrite the procedure DFS, using a stack to eliminate recursion.

22.3-7
Give a counterexample to the conjecture that if there is a path from u to v in a directed graph G, and if $d[u] < d[v]$ in a depth-first search of G, then v is a descendant of u in the depth-first forest produced.

22.3-8
Give a counterexample to the conjecture that if there is a path from u to v in a directed graph G, then any depth-first search must result in $d[v] \leq f[u]$.

22.3-9
Modify the pseudocode for depth-first search so that it prints out every edge in the directed graph G, together with its type. Show what modifications, if any, must be made if G is undirected.