20 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. Several other graph algorithms elaborate on basic graph searching. Techniques for searching a graph lie at the heart of the field of graph algorithms.

Section 20.1 discusses the two most common computational representations of graphs: as adjacency lists and as adjacency matrices. Section 20.2 presents a simple graph-searching algorithm called breadth-first search and shows how to create a breadth-first tree. Section 20.3 presents depth-first search and proves some standard results about the order in which depth-first search visits vertices. Section 20.4 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 the topic of Section 20.5.

20.1 Representations of graphs

You can choose between two standard ways to represent a graph $G = (V, E)$: as a collection of adjacency lists or as an adjacency matrix. Either way applies to both directed and undirected graphs. Because the adjacency-list representation provides a compact way to represent *sparse* graphs—those for which $|E|$ is much less than $|V|^2$—it is usually the method of choice. Most of the graph algorithms presented in this book assume that an input graph is represented in adjacency-list form. You might prefer an adjacency-matrix representation, however, when the graph is *dense*—$|E|$ is close to $|V|^2$—or when you need to be able to tell quickly whether there is an edge connecting two given vertices. For example, two of the

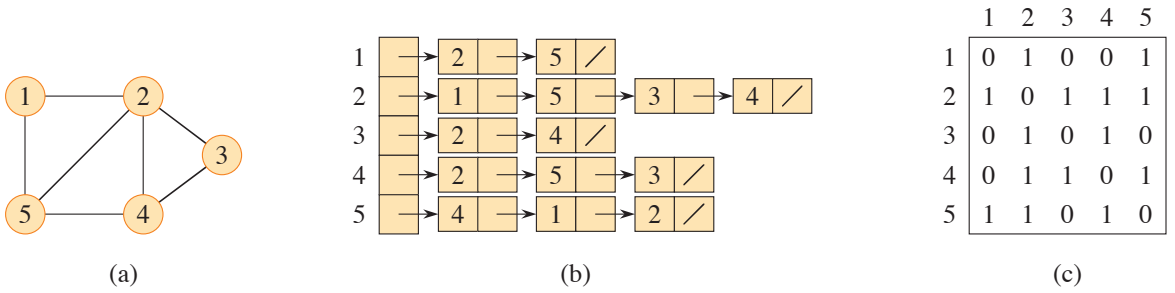

Figure 20.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

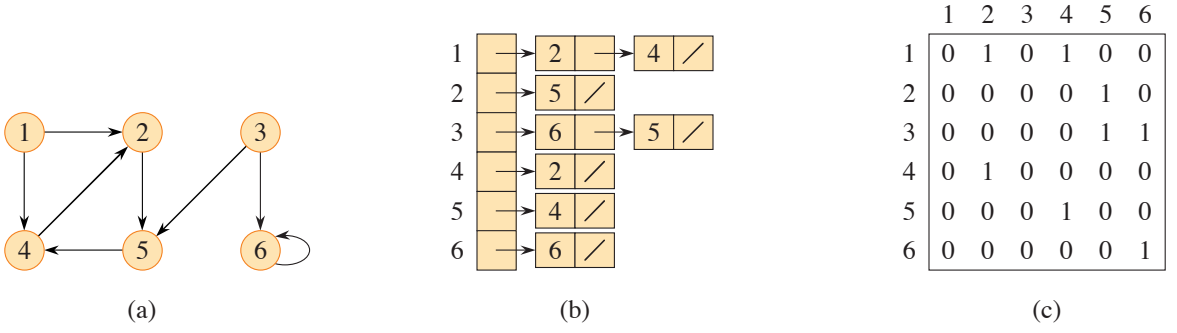

Figure 20.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

all-pairs shortest-paths algorithms presented in Chapter 23 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 can contain pointers to these vertices.) Since the adjacency lists represent the edges of a graph, our pseudocode treats the array Adj as an attribute of the graph, just like the edge set E. In pseudocode, therefore, you will see notation such as $G.Adj[u]$. Figure 20.1(b) is an adjacency-list representation of the undirected graph in Figure 20.1(a). Similarly, Figure 20.2(b) is an adjacency-list representation of the directed graph in Figure 20.2(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 $\Theta(V + E)$. Finding each edge in the graph also takes $\Theta(V + E)$ time, rather than just $\Theta(E)$, since each of the $|V|$ adjacency lists must be examined. Of course, if $|E| = \Omega(V)$ —such as in a connected, undirected graph or a strongly connected, directed graph—we can say that finding each edge takes $\Theta(E)$ time.

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

A potential disadvantage of the adjacency-list representation is that it provides no quicker way to determine whether a given edge (u, v) is present in the graph than to search for v in the adjacency list $Adj[u]$. An adjacency-matrix representation of the graph remedies this disadvantage, but at the cost of using asymptotically more memory. (See Exercise 20.1-8 for suggestions of variations on adjacency lists that permit faster edge lookup.)

The ***adjacency-matrix representation*** of a graph $G = (V, E)$ assumes 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 20.1(c) and 20.2(c) are the adjacency matrices of the undirected and directed graphs in Figures 20.1(a) and 20.2(a), respectively. The adjacency matrix of a graph requires $\Theta(V^2)$ memory, independent of the number of edges in the graph. Because finding each edge in the graph requires examining the entire adjacency matrix, doing so takes $\Theta(V^2)$ time.

Observe the symmetry along the main diagonal of the adjacency matrix in Figure 20.1(c). 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, an adjacency matrix can represent a weighted graph. For example, if $G = (V, E)$ is a weighted graph with edge-weight function w, you can store the weight $w(u, v)$ of the edge $(u, v) \in E$

as the entry in row u and column v of the adjacency matrix. If an edge does not exist, you can store a NIL value 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 space-efficient as the adjacency-matrix representation, adjacency matrices are simpler, and so you might prefer them when graphs are reasonably small. Moreover, adjacency matrices carry a further advantage for unweighted graphs: they require only one bit per entry.

Representing attributes

Most algorithms that operate on graphs need to maintain attributes for vertices and/or edges. We indicate these attributes using our usual notation, such as $v.d$ for an attribute d of a vertex v. When we indicate edges as pairs of vertices, we use the same style of notation. For example, if edges have an attribute f, then we denote this attribute for edge (u, v) by $(u, v).f$. For the purpose of presenting and understanding algorithms, our attribute notation suffices.

Implementing vertex and edge attributes in real programs can be another story entirely. There is no one best way to store and access vertex and edge attributes. For a given situation, your decision will likely depend on the programming language you are using, the algorithm you are implementing, and how the rest of your program uses the graph. If you represent a graph using adjacency lists, one design choice is to represent vertex attributes in additional arrays, such as an array $d[1 : |V|]$ that parallels the Adj array. If the vertices adjacent to u belong to $Adj[u]$, then the attribute $u.d$ can actually be stored in the array entry $d[u]$. Many other ways of implementing attributes are possible. For example, in an object-oriented programming language, vertex attributes might be represented as instance variables within a subclass of a `Vertex` class.

Exercises

20.1-1

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?

20.1-2

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

20.1-3

The ***transpose*** of a directed graph $G = (V, E)$ is the graph $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, where $E^{\mathrm{T}} = \{(v, u) \in V \times V : (u, v) \in E\}$. That is, 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.

20.1-4

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.

20.1-5

The ***square*** of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$ such that $(u, v) \in E^2$ if and only if G contains a path with at most two edges between u and v. 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.

20.1-6

Most graph algorithms that take an adjacency-matrix representation as input require $\Omega(V^2)$ time, but there are some exceptions. Show how to determine whether a directed graph G contains a ***universal sink***—a vertex with in-degree $|V| - 1$ and out-degree 0—in $O(V)$ time, given an adjacency matrix for G.

20.1-7

The ***incidence matrix*** of a directed graph $G = (V, E)$ with no self-loops 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.

20.1-8

Suppose that instead of a linked list, each array entry $Adj[u]$ is a hash table containing the vertices v for which $(u, v) \in E$, with collisions resolved by chaining. Under the assumption of uniform independent hashing, 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 with the hash table?

20.2 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 21.2) and Dijkstra's single-source shortest-paths algorithm (Section 22.3) 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 from s to each reachable vertex, where the distance to a vertex v equals the smallest number of edges needed to go from s to v. Breadth-first search also produces a "breadth-first tree" with root s that contains all reachable vertices. For any vertex v reachable from s, the simple 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. You can think of it as discovering vertices in waves emanating from the source vertex. That is, starting from s, the algorithm first discovers all neighbors of s, which have distance 1. Then it discovers all vertices with distance 2, then all vertices with distance 3, and so on, until it has discovered every vertex reachable from s.

In order to keep track of the waves of vertices, breadth-first search could maintain separate arrays or lists of the vertices at each distance from the source vertex. Instead, it uses a single first-in, first-out queue (see Section 10.1.3) containing some vertices at a distance k, possibly followed by some vertices at distance $k + 1$. The queue, therefore, contains portions of two consecutive waves at any time.

To keep track of progress, breadth-first search colors each vertex white, gray, or black. All vertices start out white, and vertices not reachable from the source vertex s stay white the entire time. A vertex that is reachable from s is ***discovered*** the first time it is encountered during the search, at which time it becomes gray, indicating that is now on the frontier of the search: the boundary between discovered and undiscovered vertices. The queue contains all the gray vertices. Eventually, all the edges of a gray vertex will be explored, so that all of its neighbors will be

discovered. Once all of a vertex's edges have been explored, the vertex is behind the frontier of the search, and it goes from gray to black.[1]

Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex s. Whenever the search discovers a white vertex v in the course of scanning the adjacency list of a gray 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 every vertex reachable from s is discovered at most once, each vertex reachable from s has exactly one parent. (There is one exception: because s is the root of the breadth-first tree, it has no parent.) Ancestor and descendant relationships in the breadth-first tree are defined relative to the root s as usual: if u is on the simple 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 on the following page assumes that the graph $G = (V, E)$ is represented using adjacency lists. It denotes the queue by Q, and it attaches three additional attributes to each vertex v in the graph:

- $v.color$ is the color of v: WHITE, GRAY, or BLACK.
- $v.d$ holds the distance from the source vertex s to v, as computed by the algorithm.
- $v.\pi$ is v's predecessor in the breadth-first tree. If v has no predecessor because it is the source vertex or is undiscovered, then $v.\pi = \text{NIL}$.

Figure 20.3 illustrates the progress of BFS on an undirected graph.

The procedure BFS works as follows. With the exception of the source vertex s, lines 1–4 paint every vertex white, set $u.d = \infty$ for each vertex u, and set the parent of every vertex to be NIL. Because the source vertex s is always the first vertex discovered, lines 5–7 paint s gray, set $s.d$ to 0, and set the predecessor of s to NIL. Lines 8–9 create the queue Q, initially containing just the source vertex.

The **while** loop of lines 10–18 iterates as long as there remain gray vertices, which are on the frontier: 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

[1] We distinguish between gray and black vertices to help us understand how breadth-first search operates. In fact, as Exercise 20.2-3 shows, we get the same result even if we do not distinguish between gray and black vertices.

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each vertex v in G.Adj[u]    // search the neighbors of u
13          if v.color == WHITE          // is v being discovered now?
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)            // v is now on the frontier
18      u.color = BLACK                  // u is now behind the frontier
```

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 procedure discovers it by executing lines 14–17. These lines paint vertex v gray, set v's distance $v.d$ to $u.d + 1$, record u as v's parent $v.\pi$, and place v at the tail of the queue Q. Once the procedure has examined all the vertices on u's adjacency list, it blackens u in line 18, indicating that u is now behind the frontier. 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 do not. (See Exercise 20.2-5.)

A simple change allows the BFS procedure to terminate in many cases before the queue Q becomes empty. Because each vertex is discovered at most once and receives a finite d value only when it is discovered, the algorithm can terminate once every vertex has a finite d value. If BFS keeps count of how many vertices have been discovered, it can terminate once either the queue Q is empty or all $|V|$ vertices are discovered.

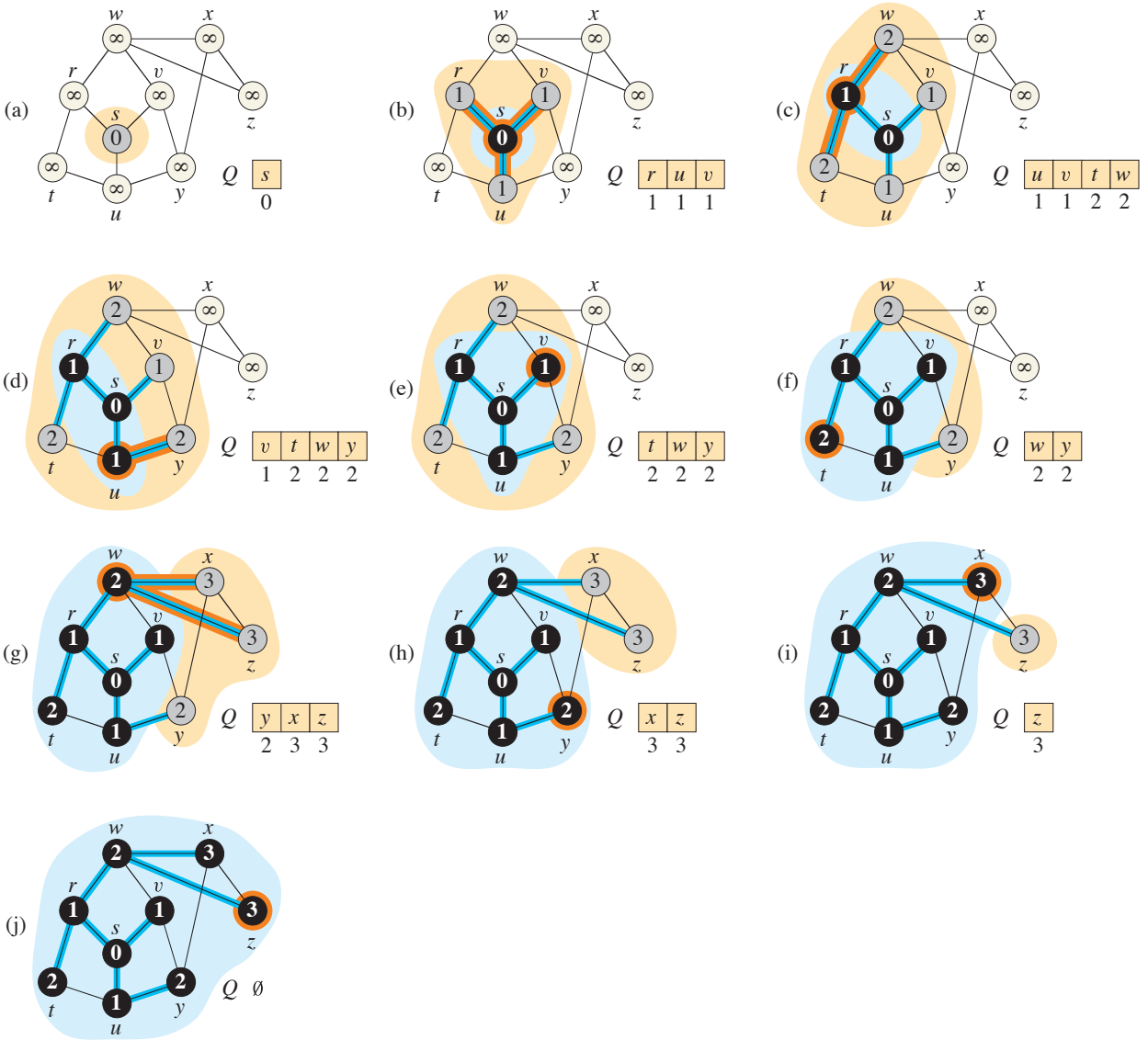

Figure 20.3 The operation of BFS on an undirected graph. Each part shows the graph and the queue Q at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances appear within each vertex and below vertices in the queue. The tan region surrounds the frontier of the search, consisting of the vertices in the queue. The light blue region surrounds the vertices behind the frontier, which have been dequeued. Each part highlights in orange the vertex dequeued and the breadth-first tree edges added, if any, in the previous iteration. Blue edges belong to the breadth-first tree constructed so far.

Analysis

Before proving the various properties of breadth-first search, let's take on the easier job of analyzing its running time on an input graph $G = (V, E)$. We use aggregate analysis, as we saw in Section 16.1. After initialization, breadth-first search never whitens a vertex, 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, and so the total time devoted to queue operations is $O(V)$. Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all $|V|$ adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(V + E)$. The overhead for initialization is $O(V)$, and thus the total running time of the BFS procedure is $O(V + E)$. Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G.

Shortest paths

Now, let's see why breadth-first search finds the shortest distance from a given source vertex s to each vertex in a graph. 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 $\delta(s, v) = \infty$. We call a path of length $\delta(s, v)$ from s to v a ***shortest path***[2] from s to v. Before showing that breadth-first search correctly computes shortest-path distances, we investigate an important property of shortest-path distances.

Lemma 20.1

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

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

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 again, the inequality holds. ■

Our goal is to show that the BFS procedure properly computes $v.d = \delta(s, v)$ for each vertex $v \in V$. We first show that $v.d$ bounds $\delta(s, v)$ from above.

---

[2] Chapters 22 and 23 generalize 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.

Lemma 20.2

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, for each vertex $v \in V$, the value $v.d$ computed by BFS satisfies $v.d \geq \delta(s, v)$ at all times, including at termination.

Proof The lemma is true intuitively, because any finite value assigned to $v.d$ equals the number of edges on some path from s to v. The formal proof is by induction on the number of ENQUEUE operations. The inductive hypothesis is that $v.d \geq \delta(s, v)$ for all $v \in V$.

The base case of the induction is the situation immediately after enqueuing s in line 9 of BFS. The inductive hypothesis holds here, because $s.d = 0 = \delta(s, s)$ and $v.d = \infty \geq \delta(s, v)$ 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 $u.d \geq \delta(s, u)$. The assignment performed by line 15 and Lemma 20.1 give

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

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

To prove that $v.d = \delta(s, v)$, we first show more precisely how the queue Q operates during the course of BFS. The next lemma shows that at all times, the d values of vertices in the queue either are all the same or form a sequence $\langle k, k, \ldots, k, k+1, k+1, \ldots, k+1 \rangle$ for some integer $k \geq 0$.

Lemma 20.3

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, $v_r.d \leq v_1.d + 1$ and $v_i.d \leq v_{i+1}.d$ 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 trivially holds.

For the inductive step, we must prove that the lemma holds after both dequeuing and enqueuing a vertex. First, we examine dequeuing. When 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, $v_1.d \leq v_2.d$. But then we have $v_r.d \leq v_1.d + 1 \leq v_2.d + 1$, and the remaining inequalities are unaffected. Thus, the lemma follows with v_2 as the new head.

Now, we examine enqueuing. When line 17 of BFS enqueues a vertex v onto a queue containing vertices $\langle v_1, v_2, \ldots, v_r \rangle$, the enqueued vertex becomes v_{r+1}. If the queue was empty before v was enqueued, then after enqueuing v, we have $r = 1$ and the lemma trivially holds. Now suppose that the queue was nonempty when v was enqueued. At that time, the procedure has most recently removed vertex u, whose adjacency list is currently being scanned, from the queue Q. Just before u was removed, we had $u = v_1$ and the inductive hypothesis held, so that $u.d \leq v_2.d$ and $v_r.d \leq u.d + 1$. After u is removed from the queue, the vertex that had been v_2 becomes the new head v_1 of the queue, so that now $u.d \leq v_1.d$. Thus, $v_{r+1}.d = v.d = u.d + 1 \leq v_1.d + 1$. Since $v_r.d \leq u.d + 1$, we have $v_r.d \leq u.d + 1 = v.d = v_{r+1}.d$, 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 monotonically increase over time.

Corollary 20.4
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 $v_i.d \leq v_j.d$ at the time that v_j is enqueued.

Proof Immediate from Lemma 20.3 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 20.5 (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, $v.d = \delta(s, v)$ 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 $v.\pi$ followed by the edge $(v.\pi, v)$.

Proof Assume for the purpose of contradiction that some vertex receives a d value not equal to its shortest-path distance. Of all such vertices, let v be a vertex that has the minimum $\delta(s, v)$. By Lemma 20.2, we have $v.d \geq \delta(s, v)$, and thus $v.d > \delta(s, v)$. We cannot have $v = s$, because $s.d = 0$ and $\delta(s, s) = 0$. Vertex v must be reachable from s, for otherwise we would have $\delta(s, v) = \infty \geq v.d$. Let u be the vertex immediately preceding v on some shortest path from s to v (since $v \neq s$, vertex u must exist), 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 $u.d = \delta(s,u)$. Putting these properties together gives

$$v.d > \delta(s,v) = \delta(s,u) + 1 = u.d + 1 \,. \tag{20.1}$$

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 each of these cases leads to a contradiction of inequality (20.1). If v is white, then line 15 sets $v.d = u.d + 1$, contradicting inequality (20.1). If v is black, then it was already removed from the queue and, by Corollary 20.4, we have $v.d \le u.d$, again contradicting inequality (20.1). 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 $v.d = w.d + 1$. By Corollary 20.4, however, $w.d \le u.d$, and so $v.d = w.d + 1 \le u.d + 1$, once again contradicting inequality (20.1).

Thus we conclude that $v.d = \delta(s,v)$ for all $v \in V$. All vertices v reachable from s must be discovered, for otherwise they would have $\infty = v.d > \delta(s,v)$. To conclude the proof of the theorem, observe from lines 15–16 that if $v.\pi = u$, then $v.d = u.d + 1$. Thus, to form a shortest path from s to v, take a shortest path from s to $v.\pi$ and then traverse the edge $(v.\pi, v)$. ■

Breadth-first trees

The blue edges in Figure 20.3 show the breadth-first tree built by the BFS procedure as it searches the graph. The tree corresponds to the π attributes. 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

$$V_\pi = \{v \in V : v.\pi \ne \text{NIL}\} \cup \{s\} \tag{20.2}$$

and

$$E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\} \,. \tag{20.3}$$

The predecessor subgraph G_π is a ***breadth-first tree*** if V_π consists of the vertices reachable from s and, for all $v \in V_\pi$, the subgraph G_π contains a unique simple path from s to v 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 on page 1169). We call the edges in E_π ***tree edges***.

The following lemma shows that the predecessor subgraph produced by the BFS procedure is a breadth-first tree.

Lemma 20.6

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 $v.\pi = u$ 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 the predecessor subgraph G_π forms a tree, by Theorem B.2, it contains a unique simple path from s to each vertex in V_π. Applying Theorem 20.5 inductively yields that every such path is a shortest path in G. ■

The PRINT-PATH procedure prints out the vertices on a shortest path from s to v, assuming that BFS has already computed a breadth-first tree. 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.

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

Exercises

20.2-1
Show the d and π values that result from running breadth-first search on the directed graph of Figure 20.2(a), using vertex 3 as the source.

20.2-2
Show the d and π values that result from running breadth-first search on the undirected graph of Figure 20.3, using vertex u as the source. Assume that neighbors of a vertex are visited in alphabetical order.

20.2-3
Show that using a single bit to store each vertex color suffices by arguing that the BFS procedure produces the same result if line 18 is removed. Then show how to obviate the need for vertex colors altogether.

20.2-4
What is the running time of BFS if we represent its input graph by an adjacency matrix and modify the algorithm to handle this form of input?

20.2-5

Argue that in a breadth-first search, the value $u.d$ assigned to a vertex u is independent of the order in which the vertices appear in each adjacency list. Using Figure 20.3 as an example, show that the breadth-first tree computed by BFS can depend on the ordering within adjacency lists.

20.2-6

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 simple 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.

20.2-7

There are two types of professional wrestlers: "faces" (short for "babyfaces," i.e., "good guys") and "heels" ("bad guys"). Between any pair of professional wrestlers, there may or may not be a rivalry. You are given the names of n professional wrestlers and 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 faces and the remainder as heels such that each rivalry is between a face and a heel. If it is possible to perform such a designation, your algorithm should produce it.

★ ***20.2-8***

The ***diameter*** of a tree $T = (V, E)$ is defined as $\max\{\delta(u, v) : u, v \in V\}$, that 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.

20.3 Depth-first search

As its name implies, depth-first search searches "deeper" in the graph whenever possible. Depth-first search explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it. Once 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 all vertices that are reachable from the original source vertex have been discovered. If any undiscovered vertices remain, then depth-first search selects one of them as a new source, repeating the search

from that source. The algorithm repeats this entire process until it has discovered every vertex.[3]

As in breadth-first search, whenever depth-first search discovers a vertex v during a scan of the adjacency list of an already discovered vertex u, it records this event by setting v's predecessor attribute $v.\pi$ to u. Unlike breadth-first search, whose predecessor subgraph forms a tree, depth-first search produces a predecessor subgraph that might contain several trees, because the search may repeat from multiple sources. Therefore, we define the ***predecessor subgraph*** of a depth-first search slightly differently from that of a breadth-first search: it always includes all vertices, and it accounts for multiple sources. Specifically, for a depth-first search the predecessor subgraph is $G_\pi = (V, E_\pi)$, where

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

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

Like breadth-first search, depth-first search colors vertices 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 $v.d$ records when v is first discovered (and grayed), and the second timestamp $v.f$ records when the search finishes examining v's adjacency list (and blackens v). These timestamps provide important information about the structure of the graph and are generally helpful in reasoning about the behavior of depth-first search.

The procedure DFS on the facing page records when it discovers vertex u in the attribute $u.d$ and when it finishes vertex u in the attribute $u.f$. 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,

$$u.d < u.f . \tag{20.4}$$

Vertex u is WHITE before time $u.d$, GRAY between time $u.d$ and time $u.f$, and BLACK thereafter. In the DFS procedure, the input graph G may be undirected or

[3] 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 usually serves 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'll see later in this chapter.

directed. The variable *time* is a global variable used for timestamping. Figure 20.4 illustrates the progress of DFS on the graph shown in Figure 20.2 (but with vertices labeled by letters rather than numbers).

```
DFS(G)
 1  for each vertex u ∈ G.V
 2      u.color = WHITE
 3      u.π = NIL
 4  time = 0
 5  for each vertex u ∈ G.V
 6      if u.color == WHITE
 7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
 1  time = time + 1                  // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each vertex v in G.Adj[u]    // explore each edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  time = time + 1
 9  u.f = time
10  u.color = BLACK                  // blacken u; it is finished
```

The DFS procedure works as follows. Lines 1–3 paint all vertices white and initialize their π attributes 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 by calling DFS-VISIT. Upon every call of DFS-VISIT(G, u) 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*** $u.d$ and a ***finish time*** $u.f$.

In each call DFS-VISIT(G, u), vertex u is initially white. Lines 1–3 increment the global variable *time*, record the new value of *time* as the discovery time $u.d$, and paint u gray. Lines 4–7 examine each vertex v adjacent to u and recursively visit v if it is white. As line 4 considers each vertex $v \in Adj[u]$, the depth-first search ***explores*** edge (u, v). Finally, after every edge leaving u has been explored, lines 8–10 increment *time*, record the finish time in $u.f$, and paint u black.

The results of depth-first search may depend upon the order in which line 5 of DFS examines the vertices and upon the order in which line 4 of DFS-VISIT visits the neighbors of a vertex. These different visitation orders tend not to cause

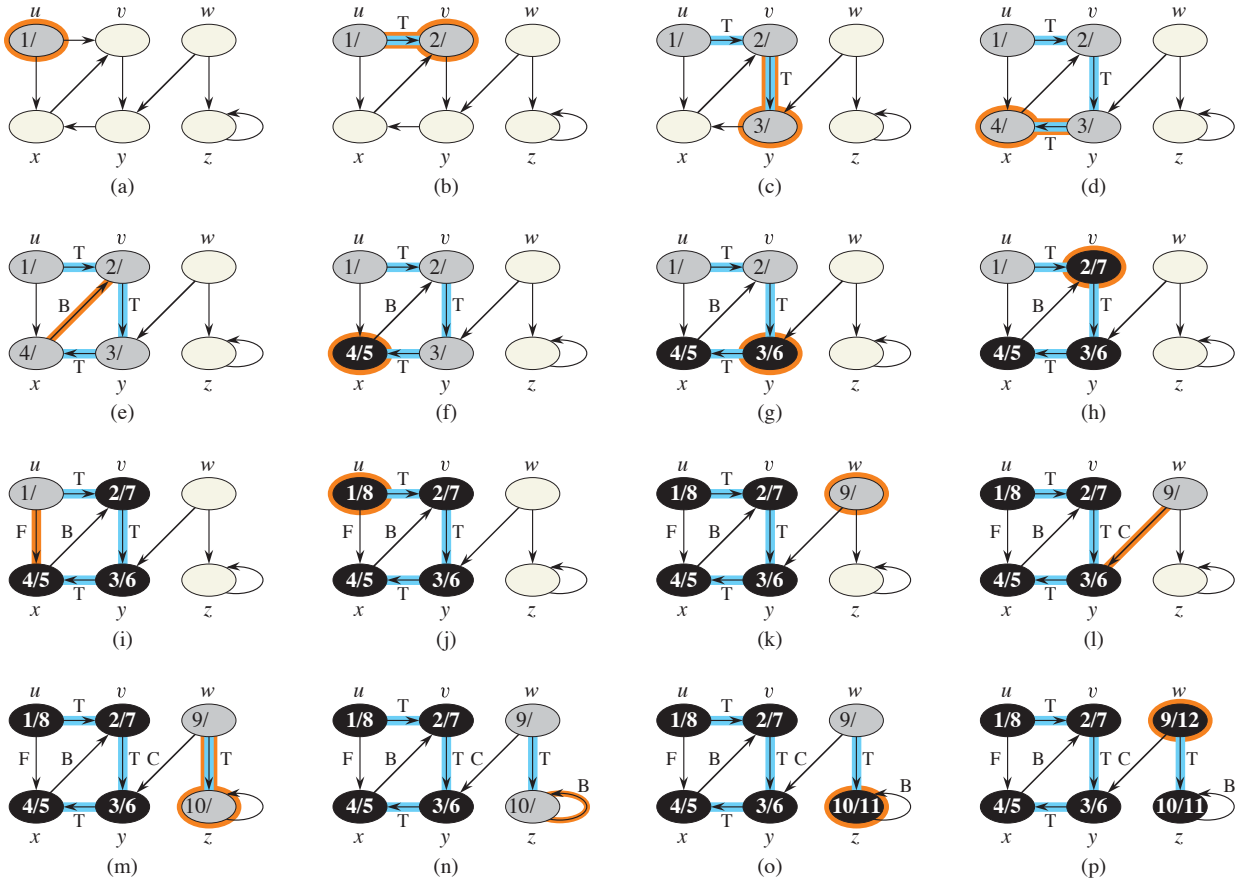

Figure 20.4 The progress of the depth-first-search algorithm DFS on a directed graph. Edges are classified as they are explored: tree edges are labeled T, back edges B, forward edges F, and cross edges C. Timestamps within vertices indicate discovery time/finish times. Tree edges are highlighted in blue. Orange highlights indicate vertices whose discovery or finish times change and edges that are explored in each step.

problems in practice, because many applications of depth-first search can use the result from any depth-first search.

What is the running time of DFS? The loops on lines 1–3 and lines 5–7 of DFS take $\Theta(V)$ time, 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 the vertex u on which DFS-VISIT is invoked must be white and the first thing DFS-VISIT does is paint vertex u gray. During an execution of DFS-VISIT(G, v), the loop in lines 4–7 executes $|Adj[v]|$ times. Since $\sum_{v \in V} |Adj[v]| = \Theta(E)$ and DFS-VISIT is called once per vertex, the

total cost of executing lines 4–7 of DFS-VISIT is $\Theta(V + E)$. 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, $u = v.\pi$ if and only if DFS-VISIT(G, 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 finish times have ***parenthesis structure***. If the DFS-VISIT procedure were to print a left parenthesis "(u" when it discovers vertex u and to print a right parenthesis "u)" when it finishes u, then the printed expression would be well formed in the sense that the parentheses are properly nested. For example, the depth-first search of Figure 20.5(a) corresponds to the parenthesization shown in Figure 20.5(b). The following theorem provides another way to characterize the parenthesis structure.

Theorem 20.7 (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 $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,
- the interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$, and u is a descendant of v in a depth-first tree, or
- the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$, and v is a descendant of u in a depth-first tree.

Proof We begin with the case in which $u.d < v.d$. We consider two subcases, according to whether $v.d < u.f$. The first subcase occurs when $v.d < u.f$, so that v was discovered while u was still gray, which implies that v is a descendant of u. Moreover, since v was discovered after 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 $[v.d, v.f]$ is entirely contained within the interval $[u.d, u.f]$. In the other subcase, $u.f < v.\ d$, and by inequality (20.4), $u.d < u.\ f < v.\ d < v.\ f$, and thus the intervals $[u.d, u.f]$ and $[v.d, v.f]$ 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.

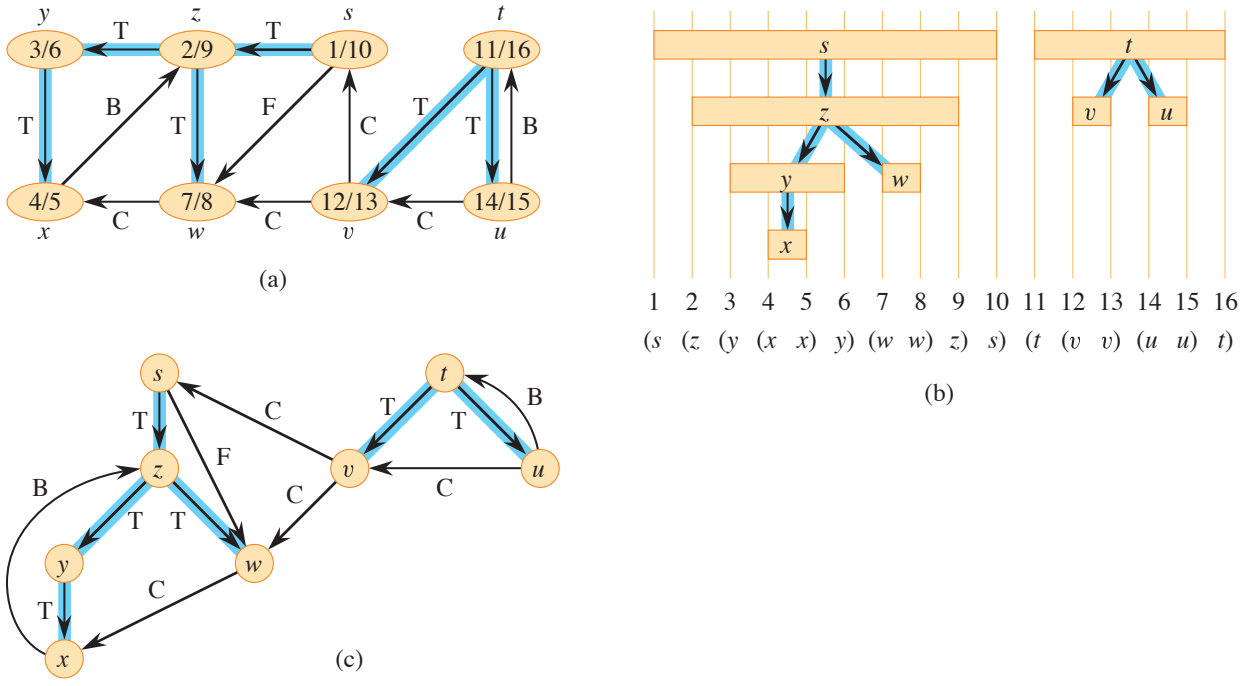

Figure 20.5 Properties of depth-first search. **(a)** The result of a depth-first search of a directed graph. Vertices are timestamped and edge types are indicated as in Figure 20.4. **(b)** Intervals for the discovery time and finish time of each vertex correspond to the parenthesization shown. Each rectangle spans the interval given by the discovery and finish times of the corresponding vertex. Only 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.

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

Corollary 20.8 (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 $u.d < v.d < v.f < u.f$.

Proof Immediate from Theorem 20.7. ■

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

Theorem 20.9 (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 $u.d$ that the search discovers u, there is a path from u to v consisting entirely of white vertices.

Proof $\Rightarrow$: If $v = u$, then the path from u to v contains just vertex u, which is still white when $u.d$ receives a value. Now, suppose that v is a proper descendant of u in the depth-first forest. By Corollary 20.8, $u.d < v.d$, and so v is white at time $u.d$. Since v can be any descendant of u, all vertices on the unique simple path from u to v in the depth-first forest are white at time $u.d$.

$\Leftarrow$: Suppose that there is a path of white vertices from u to v at time $u.d$, but v does not become a descendant of u in the depth-first tree. Without loss of generality, assume that every vertex other than v 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). By Corollary 20.8, $w.f \leq u.f$. Because v must be discovered after u is discovered, but before w is finished, $u.d < v.d < w.f \leq u.f$. Theorem 20.7 then implies that the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$. By Corollary 20.8, v must after all be a descendant of u. ■

Classification of edges

You can obtain important information about a graph by classifying its edges during a depth-first search. For example, Section 20.4 will show that a directed graph is acyclic if and only if a depth-first search yields no "back" edges (Lemma 20.11).

The depth-first forest G_π produced by a depth-first search on graph G can contain four types of edges:

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. We consider self-loops, which may occur in directed graphs, to be back edges.
3. ***Forward edges*** are those nontree edges (u, v) connecting a vertex u to a proper 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 20.4 and 20.5, edge labels indicate edge types. Figure 20.5(c) also shows how to redraw the graph of Figure 20.5(a) so that all tree and forward edges head downward in a depth-first tree and all back edges go up. You can redraw any graph in this fashion.

The DFS algorithm has enough information to classify some edges as it encounters them. The key idea is that when an edge (u, v) is first explored, the color of vertex v says something about the edge:

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 1 more than the depth in the depth-first forest of the vertex most recently discovered. Depth-first search always explores from the deepest gray vertex, so that an edge that reaches another gray vertex has reached an ancestor. The third case handles the remaining possibility. Exercise 20.3-5 asks you to show that such an edge (u, v) is a forward edge if $u.d < v.d$ and a cross edge if $u.d > v.d$.

According to the following theorem, forward and cross edges never occur in a depth-first search of an undirected graph.

Theorem 20.10

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 $u.d < v.d$. Then, while u is gray, the search must discover and finish v before it finishes u, since v is on u's adjacency list. If the first time that the search explores edge (u, v), it is in the direction from u to v, then v is undiscovered (white) until that time, for otherwise the search would have explored this edge already in the direction from v to u. Thus, (u, v) becomes a tree edge. If the search explores (u, v) first in the direction from v to u, then (u, v) is a back edge, since there must be a path of tree edges from u to v. ■

Since (u, v) and (v, u) are really the same edge in an undirected graph, the proof of Theorem 20.10 says how to classify the edge. When searching from a vertex, which must be gray, if the adjacent vertex is white, then the edge is a tree edge. Otherwise, the edge is a back edge.

The next two sections apply the above theorems about depth-first search.

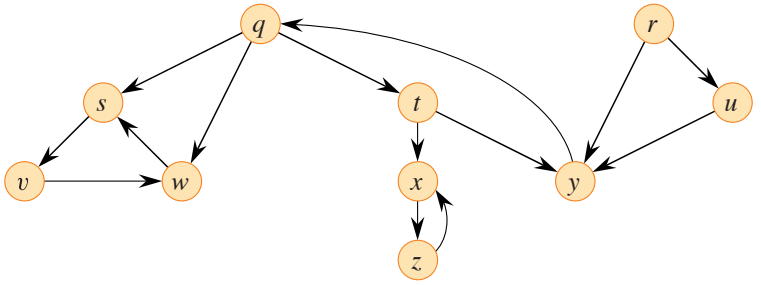

Figure 20.6 A directed graph for use in Exercises 20.3-2 and 20.5-2.

Exercises

20.3-1

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.

20.3-2

Show how depth-first search works on the graph of Figure 20.6. 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 finish times for each vertex, and show the classification of each edge.

20.3-3

Show the parenthesis structure of the depth-first search of Figure 20.4.

20.3-4

Show that using a single bit to store each vertex color suffices by arguing that the DFS procedure produces the same result if line 10 of DFS-VISIT is removed.

20.3-5

Show that in a directed graph, edge (u, v) is

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

b. a back edge if and only if $v.d \leq u.d < u.f \leq v.f$, and

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

20.3-6
Rewrite the procedure DFS, using a stack to eliminate recursion.

20.3-7
Give a counterexample to the conjecture that if a directed graph G contains a path from u to v, and if $u.d < v.d$ in a depth-first search of G, then v is a descendant of u in the depth-first forest produced.

20.3-8
Give a counterexample to the conjecture that if a directed graph G contains a path from u to v, then any depth-first search must result in $v.d \leq u.f$.

20.3-9
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, you need to make if G is undirected.

20.3-10
Explain how a vertex u of a directed graph can end up in a depth-first tree containing only u, even though u has both incoming and outgoing edges in G.

20.3-11
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.

20.3-12
Show how to use a depth-first search of an undirected graph G to identify the connected components of G, so that the depth-first forest contains as many trees as G has connected components. More precisely, show how to modify depth-first search so that it assigns to each vertex v an integer label $v.cc$ between 1 and k, where k is the number of connected components of G, such that $u.cc = v.cc$ if and only if u and v belong to the same connected component.

★ ***20.3-13***
A directed graph $G = (V, E)$ is ***singly connected*** if $u \rightsquigarrow v$ implies that G contains at most one simple path from u to v for all vertices $u, v \in V$. Give an efficient algorithm to determine whether a directed graph is singly connected.

20.4 Topological sort

This section shows how to use depth-first search to perform a topological sort of a directed acyclic graph, or a "dag" as it is sometimes called. A ***topological sort*** of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering. Topological sorting is defined only on directed graphs that are acyclic; no linear ordering is possible when a directed graph contains a cycle. Think of a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right. Topological sorting is thus different from the usual kind of "sorting" studied in Part II.

Many applications use directed acyclic graphs to indicate precedences among events. Figure 20.7 gives an example that arises when Professor Bumstead gets dressed in the morning. The professor must don certain garments before others (e.g., socks before shoes). Other items may be put on in any order (e.g., socks and pants). A directed edge (u, v) in the dag of Figure 20.7(a) indicates that garment u must be donned before garment v. A topological sort of this dag therefore gives a possible order for getting dressed. Figure 20.7(b) shows the topologically sorted dag as an ordering of vertices along a horizontal line such that all directed edges go from left to right.

The procedure TOPOLOGICAL-SORT topologically sorts a dag. Figure 20.7(b) shows how the topologically sorted vertices appear in reverse order of their finish times.

```
TOPOLOGICAL-SORT(G)
1  call DFS(G) to compute finish times v.f for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

The TOPOLOGICAL-SORT procedure runs in $\Theta(V + E)$ time, since depth-first search takes $\Theta(V + E)$ time and it takes $O(1)$ time to insert each of the $|V|$ vertices onto the front of the linked list.

To prove the correctness of this remarkably simple and efficient algorithm, we start with the following key lemma characterizing directed acyclic graphs.

Lemma 20.11

A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.

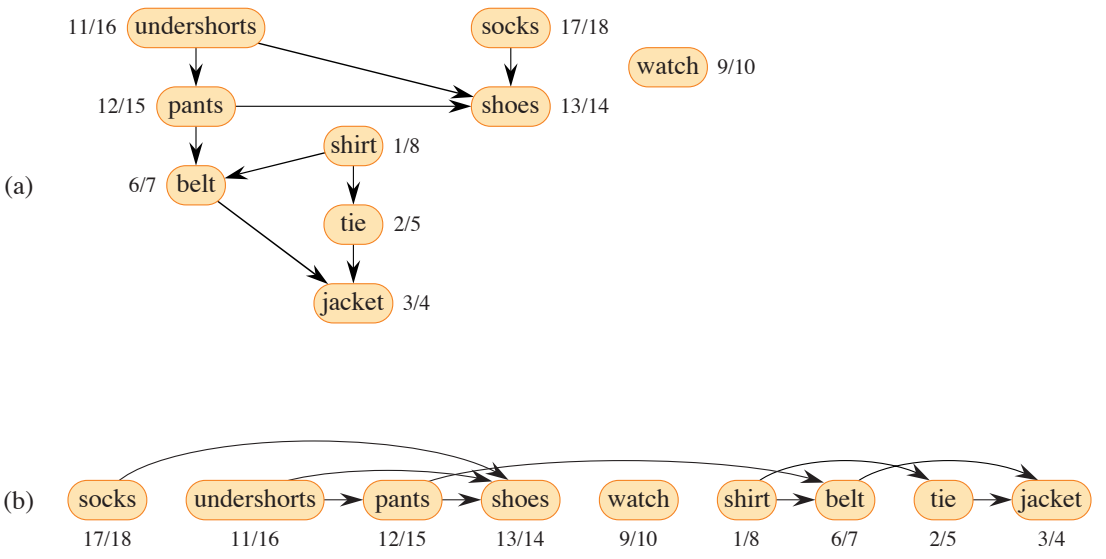

Figure 20.7 **(a)** Professor Bumstead topologically sorts his clothing when getting dressed. Each directed edge (u, v) means that garment u must be put on before garment v. The discovery and finish times from a depth-first search are shown next to each vertex. **(b)** The same graph shown topologically sorted, with its vertices arranged from left to right in order of decreasing finish time. All directed edges go from left to right.

Proof $\Rightarrow$: Suppose that a depth-first search produces a back edge (u, v). Then vertex v is an ancestor of vertex u in the depth-first forest. Thus, G contains a path from v to u, and the back edge (u, v) completes a cycle.

$\Leftarrow$: Suppose that G contains a cycle c. We show that a depth-first search of G yields a back edge. Let v be the first vertex to be discovered in c, and let (u, v) be the preceding edge in c. At time $v.d$, the vertices of c form a path of white vertices from v to u. By the white-path theorem, vertex u becomes a descendant of v in the depth-first forest. Therefore, (u, v) is a back edge. ■

Theorem 20.12

TOPOLOGICAL-SORT produces a topological sort of the directed acyclic graph provided as its input.

Proof Suppose that DFS is run on a given dag $G = (V, E)$ to determine finish times for its vertices. It suffices to show that for any pair of distinct vertices $u, v \in V$, if G contains an edge from u to v, then $v.f < u.f$. Consider any edge (u, v) explored by DFS(G). When this edge is explored, v cannot be gray, since then v would be an ancestor of u and (u, v) would be a back edge, contradicting Lemma 20.11. Therefore, v must be either white or black. If v is

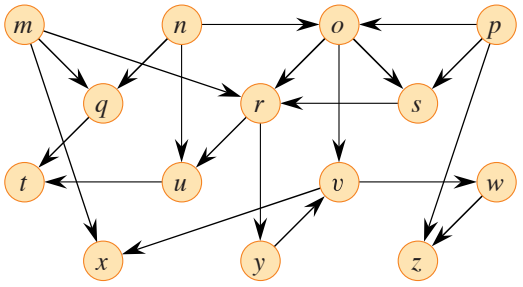

Figure 20.8 A dag for topological sorting.

white, it becomes a descendant of u, and so $v.f < u.f$. If v is black, it has already been finished, so that $v.f$ has already been set. Because the search is still exploring from u, it has yet to assign a timestamp to $u.f$, so that the timestamp eventually assigned to $u.f$ is greater than $v.f$. Thus, $v.f < u.f$ for any edge (u, v) in the dag, proving the theorem. ■

Exercises

20.4-1

Show the ordering of vertices produced by TOPOLOGICAL-SORT when it is run on the dag of Figure 20.8. 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.

20.4-2

Give a linear-time algorithm that, given a directed acyclic graph $G = (V, E)$ and two vertices $a, b \in V$, returns the number of simple paths from a to b in G. For example, the directed acyclic graph of Figure 20.8 contains exactly four simple paths from vertex p to vertex v: $\langle p, o, v\rangle$, $\langle p, o, r, y, v\rangle$, $\langle p, o, s, r, y, v\rangle$, and $\langle p, s, r, y, v\rangle$. Your algorithm needs only to count the simple paths, not list them.

20.4-3

Give an algorithm that determines whether an undirected graph $G = (V, E)$ contains a simple cycle. Your algorithm should run in $O(V)$ time, independent of $|E|$.

20.4-4

Prove or disprove: If a directed graph G contains cycles, then the vertex ordering produced by TOPOLOGICAL-SORT(G) minimizes the number of "bad" edges that are inconsistent with the ordering produced.

20.4-5

Another way to topologically sort a directed acyclic graph $G = (V, E)$ is to repeatedly find a vertex of in-degree 0, output it, and remove it and all of its outgoing edges from the graph. Explain how to implement this idea so that it runs in time $O(V + E)$. What happens to this algorithm if G has cycles?

20.5 Strongly connected components

We now consider a classic application of depth-first search: decomposing a directed graph into its strongly connected components. This section shows how to do so using two depth-first searches. Many algorithms that work with directed graphs begin with such a decomposition. After decomposing the graph into strongly connected components, such algorithms run separately on each one and then combine the solutions according to the structure of connections among components.

Recall from Appendix B that a strongly connected component of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$, that is, vertices u and v are reachable from each other. Figure 20.9 shows an example.

The algorithm for finding the strongly connected components of a directed graph $G = (V, E)$ uses the transpose of G, which we defined in Exercise 20.1-3 to be the graph $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, where $E^{\mathrm{T}} = \{(u, v) : (v, u) \in E\}$. That is, E^{T} consists of the edges of G with their directions reversed. Given an adjacency-list representation of G, the time to create G^{T} is $\Theta(V + E)$. The graphs G and G^{T} have exactly the same strongly connected components: u and v are reachable from each other in G if and only if they are reachable from each other in G^{T}. Figure 20.9(b) shows the transpose of the graph in Figure 20.9(a), with the strongly connected components shaded blue in both parts.

The linear-time (i.e., $\Theta(V + E)$-time) procedure STRONGLY-CONNECTED-COMPONENTS on the next page computes the strongly connected components of a directed graph $G = (V, E)$ using two depth-first searches, one on G and one on G^{T}.

The idea behind this algorithm comes from a key property of the ***component graph*** $G^{\mathrm{SCC}} = (V^{\mathrm{SCC}}, E^{\mathrm{SCC}})$, defined as follows. Suppose that G has strongly connected components $C_1, C_2, \ldots, C_k$. The vertex set V^{SCC} is $\{v_1, v_2, \ldots, v_k\}$, and it contains one vertex v_i for each strongly connected component C_i of G. There is an edge $(v_i, v_j) \in E^{\mathrm{SCC}}$ if G contains a directed edge (x, y) for some $x \in C_i$ and some $y \in C_j$. Looked at another way, if we contract all edges whose incident vertices are within the same strongly connected component of G so that

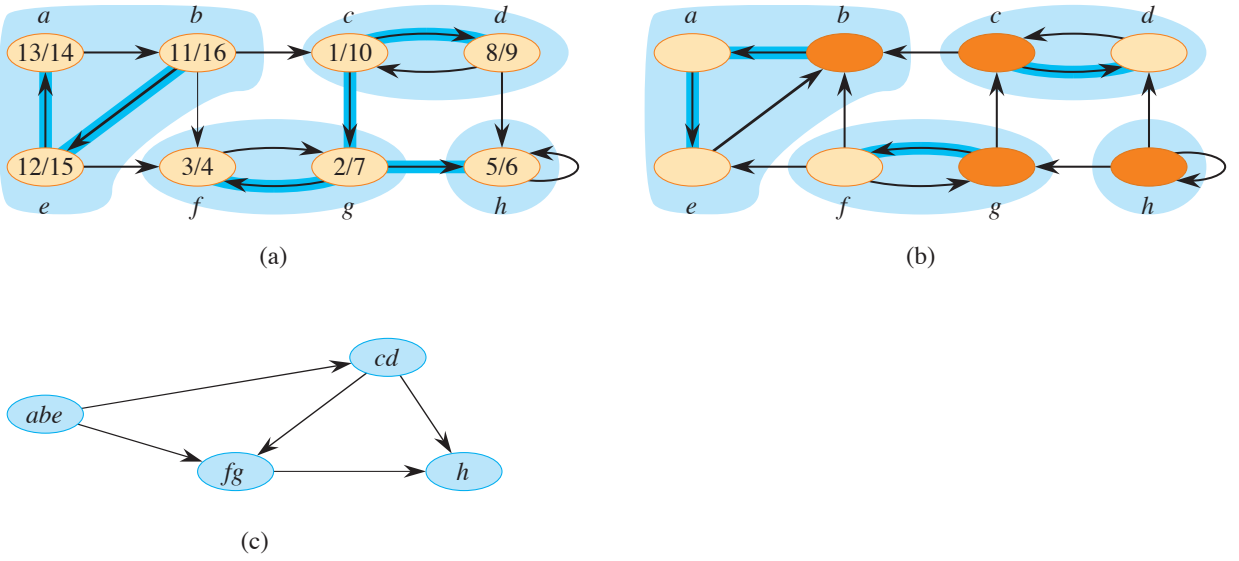

Figure 20.9 **(a)** A directed graph G. Each region shaded light blue is a strongly connected component of G. Each vertex is labeled with its discovery and finish times in a depth-first search, and tree edges are dark blue. **(b)** The graph G^{T}, the transpose of G, with the depth-first forest computed in line 3 of STRONGLY-CONNECTED-COMPONENTS shown and tree edges shaded dark blue. Each strongly connected component corresponds to one depth-first tree. Orange vertices b, c, g, and h are the roots of the depth-first trees produced by the depth-first search of G^{T}. **(c)** The acyclic component graph G^{SCC} obtained by contracting all edges within each strongly connected component of G so that only a single vertex remains in each component.

STRONGLY-CONNECTED-COMPONENTS(G)

1 call DFS(G) to compute finish times $u.f$ for each vertex u
2 create G^{T}
3 call DFS(G^{T}), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

only a single vertex remains, the resulting graph is G^{SCC}. Figure 20.9(c) shows the component graph of the graph in Figure 20.9(a).

The following lemma gives the key property that the component graph is acyclic. We'll see that the algorithm uses this property to visit the vertices of the component graph in topologically sorted order, by considering vertices in the second depth-first search in decreasing order of the finish times that were computed in the first depth-first search.

Lemma 20.13

Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$, let $u, v \in C$, let $u', v' \in C'$, and suppose that G contains a path $u \rightsquigarrow u'$. Then G cannot also contain a path $v' \rightsquigarrow v$.

Proof If G contains a path $v' \rightsquigarrow v$, then it contains paths $u \rightsquigarrow u' \rightsquigarrow v'$ and $v' \rightsquigarrow v \rightsquigarrow u$. Thus, u and v' are reachable from each other, thereby contradicting the assumption that C and C' are distinct strongly connected components. ■

Because the STRONGLY-CONNECTED-COMPONENTS procedure performs two depth-first searches, there are two distinct sets of discovery and finish times. In this section, discovery and finish times always refer to those computed by the *first* call of DFS, in line 1.

The notation for discovery and finish times extends to sets of vertices. For a subset U of vertices, $d(U)$ and $f(U)$ are the earliest discovery time and latest finish time, respectively, of any vertex in U: $d(U) = \min\{u.d : u \in U\}$ and $f(U) = \max\{u.f : u \in U\}$.

The following lemma and its corollary give a key property relating strongly connected components and finish times in the first depth-first search.

Lemma 20.14

Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E$, where $u \in C'$ and $v \in C$. Then $f(C') > f(C)$.

Proof We consider two cases, depending on which strongly connected component, C or C', had the first discovered vertex during the first depth-first search.

If $d(C') < d(C)$, let x be the first vertex discovered in C'. At time $x.d$, all vertices in C and C' are white. At that time, G contains a path from x to each vertex in C' consisting only of white vertices. Because $(u, v) \in E$, for any vertex $w \in C$, there is also a path in G at time $x.d$ from x to w consisting only of white vertices: $x \rightsquigarrow u \rightarrow v \rightsquigarrow w$. By the white-path theorem, all vertices in C and C' become descendants of x in the depth-first tree. By Corollary 20.8, x has the latest finish time of any of its descendants, and so $x.f = f(C') > f(C)$.

Otherwise, $d(C') > d(C)$. Let y be the first vertex discovered in C, so that $y.d = d(C)$. At time $y.d$, all vertices in C are white and G contains a path from y to each vertex in C consisting only of white vertices. By the white-path theorem, all vertices in C become descendants of y in the depth-first tree, and by Corollary 20.8, $y.f = f(C)$. Because $d(C') > d(C) = y.d$, all vertices in C' are white at time $y.d$. Since there is an edge (u, v) from C' to C, Lemma 20.13 implies that there cannot be a path from C to C'. Hence, no vertex in C' is reachable

from y. At time $y.f$, therefore, all vertices in C' are still white. Thus, for any vertex $w \in C'$, we have $w.f > y.f$, which implies that $f(C') > f(C)$. ■

Corollary 20.15

Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$, and suppose that $f(C) > f(C')$. Then E^T contains no edge (v, u) such that $u \in C'$ and $v \in C$.

Proof The contrapositive of Lemma 20.14 says that if $f(C') < f(C)$, then there is no edge $(u, v) \in E$ such that $u \in C'$ and $v \in C$. Because the strongly connected components of G and G^T are the same, if there is no such edge $(u, v) \in E$, then there is no edge $(v, u) \in E^T$ such that $u \in C'$ and $v \in C$. ■

Corollary 20.15 provides the key to understanding why the strongly connected components algorithm works. Let's examine what happens during the second depth-first search, which is on G^T. The search starts from the vertex x whose finish time from the first depth-first search is maximum. This vertex belongs to some strongly connected component C, and since $x.f$ is maximum, $f(C)$ is maximum over all strongly connected components. When the search starts from x, it visits all vertices in C. By Corollary 20.15, G^T contains no edges from C to any other strongly connected component, and so the search from x never visits vertices in any other component. Thus, the tree rooted at x contains exactly the vertices of C. Having completed visiting all vertices in C, the second depth-first search selects as a new root a vertex from some other strongly connected component C' whose finish time $f(C')$ is maximum over all components other than C. Again, the search visits all vertices in C'. But by Corollary 20.15, if any edges in G^T go from C' to any other component, they must go to C, which the second depth-first search has already visited. In general, when the depth-first search of G^T in line 3 visits any strongly connected component, any edges out of that component must be to components that the search has already visited. Each depth-first tree, therefore, corresponds to exactly one strongly connected component. The following theorem formalizes this argument.

Theorem 20.16

The STRONGLY-CONNECTED-COMPONENTS procedure correctly computes the strongly connected components of the directed graph G provided as its input.

Proof We argue by induction on the number of depth-first trees found in the depth-first search of G^T in line 3 that the vertices of each tree form a strongly connected component. The inductive hypothesis is that the first k trees produced

in line 3 are strongly connected components. The basis for the induction, when $k = 0$, is trivial.

In the inductive step, we assume that each of the first k depth-first trees produced in line 3 is a strongly connected component, and we consider the $(k+1)$st tree produced. Let the root of this tree be vertex u, and let u be in strongly connected component C. Because of how the depth-first search chooses roots in line 3, $u.f = f(C) > f(C')$ for any strongly connected component C' other than C that has yet to be visited. By the inductive hypothesis, at the time that the search visits u, all other vertices of C are white. By the white-path theorem, therefore, all other vertices of C are descendants of u in its depth-first tree. Moreover, by the inductive hypothesis and by Corollary 20.15, any edges in G^{T} that leave C must be to strongly connected components that have already been visited. Thus, no vertex in any strongly connected component other than C is a descendant of u during the depth-first search of G^{T}. The vertices of the depth-first tree in G^{T} that is rooted at u form exactly one strongly connected component, which completes the inductive step and the proof. ■

Here is another way to look at how the second depth-first search operates. Consider the component graph $(G^{\mathrm{T}})^{\mathrm{SCC}}$ of G^{T}. If you map each strongly connected component visited in the second depth-first search to a vertex of $(G^{\mathrm{T}})^{\mathrm{SCC}}$, the second depth-first search visits vertices of $(G^{\mathrm{T}})^{\mathrm{SCC}}$ in the reverse of a topologically sorted order. If you reverse the edges of $(G^{\mathrm{T}})^{\mathrm{SCC}}$, you get the graph $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}}$. Because $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}} = G^{\mathrm{SCC}}$ (see Exercise 20.5-4), the second depth-first search visits the vertices of G^{SCC} in topologically sorted order.

Exercises

20.5-1

How can the number of strongly connected components of a graph change if a new edge is added?

20.5-2

Show how the procedure STRONGLY-CONNECTED-COMPONENTS works on the graph of Figure 20.6. Specifically, show the finish times computed in line 1 and the forest produced in line 3. Assume that the loop of lines 5–7 of DFS considers vertices in alphabetical order and that the adjacency lists are in alphabetical order.

20.5-3

Professor Bacon rewrites the algorithm for strongly connected components to use the original (instead of the transpose) graph in the second depth-first search and

scan the vertices in order of *increasing* finish times. Does this modified algorithm always produce correct results?

20.5-4
Prove that for any directed graph G, the transpose of the component graph of G^{T} is the same as the component graph of G. That is, $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}} = G^{\mathrm{SCC}}$.

20.5-5
Give an $O(V + E)$-time algorithm to compute the component graph of a directed graph $G = (V, E)$. Make sure that there is at most one edge between two vertices in the component graph your algorithm produces.

20.5-6
Give an $O(V + E)$-time algorithm that, given a directed graph $G = (V, E)$, constructs another graph $G' = (V, E')$ such that G and G' have the same strongly connected components, G' has the same component graph as G, and $|E'|$ is as small as possible.

20.5-7
A directed graph $G = (V, E)$ is ***semiconnected*** if, for all pairs of vertices $u, v \in V$, we have $u \rightsquigarrow v$ or $v \rightsquigarrow u$. Give an efficient algorithm to determine whether G is semiconnected. Prove that your algorithm is correct, and analyze its running time.

20.5-8
Let $G = (V, E)$ be a directed graph, and let $l : V \to \mathbb{R}$ be a function that assigns a real-valued label l to each vertex. For vertices $s, t \in V$, define

$$\Delta l(s,t) = \begin{cases} l(t) - l(s) & \text{if there is a path from } s \text{ to } t \text{ in } G\ , \\ -\infty & \text{otherwise}\ . \end{cases}$$

Give an $O(V + E)$-time algorithm to find vertices s and t such that $\Delta l(s, t)$ is maximum over all pairs of vertices. (*Hint:* Use Exercise 20.5-5.)

Problems

20-1 Classifying edges by breadth-first search
A depth-first forest classifies the edges of a graph into tree, back, forward, and cross edges. A breadth-first tree can also be used to classify the edges reachable from the source of the search into the same four categories.

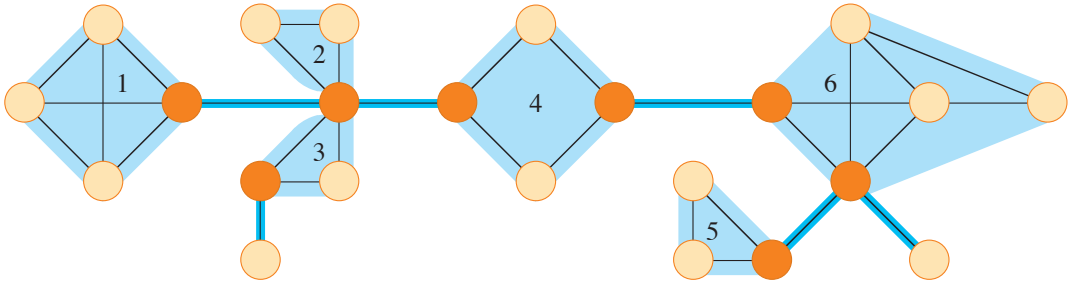

Figure 20.10 The articulation points, bridges, and biconnected components of a connected, undirected graph for use in Problem 20-2. The articulation points are the orange vertices, the bridges are the dark blue edges, and the biconnected components are the edges in the light blue regions, with a *bcc* numbering shown.

a. Prove that in a breadth-first search of an undirected graph, the following properties hold:

1. There are no back edges and no forward edges.
2. If (u, v) is a tree edge, then $v.d = u.d + 1$.
3. If (u, v) is a cross edge, then $v.d = u.d$ or $v.d = u.d + 1$.

b. Prove that in a breadth-first search of a directed graph, the following properties hold:

1. There are no forward edges.
2. If (u, v) is a tree edge, then $v.d = u.d + 1$.
3. If (u, v) is a cross edge, then $v.d \le u.d + 1$.
4. If (u, v) is a back edge, then $0 \le v.d \le u.d$.

20-2 Articulation points, bridges, and biconnected components

Let $G = (V, E)$ be a connected, undirected graph. An ***articulation point*** of G is a vertex whose removal disconnects G. A ***bridge*** of G is an edge whose removal disconnects G. A ***biconnected component*** of G is a maximal set of edges such that any two edges in the set lie on a common simple cycle. Figure 20.10 illustrates these definitions. You can determine articulation points, bridges, and biconnected components using depth-first search. Let $G_\pi = (V, E_\pi)$ be a depth-first tree of G.

a. Prove that the root of G_π is an articulation point of G if and only if it has at least two children in G_π.

b. Let v be a nonroot vertex of G_π. Prove that v is an articulation point of G if and only if v has a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v.

c. Let

$$v.low = \min \begin{Bmatrix} v.d\,, \\ w.d : (u, w) \text{ is a back edge for some descendant } u \text{ of } v\,. \end{Bmatrix}$$

Show how to compute $v.low$ for all vertices $v \in V$ in $O(E)$ time.

d. Show how to compute all articulation points in $O(E)$ time.

e. Prove that an edge of G is a bridge if and only if it does not lie on any simple cycle of G.

f. Show how to compute all the bridges of G in $O(E)$ time.

g. Prove that the biconnected components of G partition the nonbridge edges of G.

h. Give an $O(E)$-time algorithm to label each edge e of G with a positive integer $e.bcc$ such that $e.bcc = e'.bcc$ if and only if e and e' belong to the same biconnected component.

20-3 Euler tour

An ***Euler tour*** of a strongly connected, directed graph $G = (V, E)$ is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once.

a. Show that G has an Euler tour if and only if in-degree(v) = out-degree(v) for each vertex $v \in V$.

b. Describe an $O(E)$-time algorithm to find an Euler tour of G if one exists. (*Hint:* Merge edge-disjoint cycles.)

20-4 Reachability

Let $G = (V, E)$ be a directed graph in which each vertex $u \in V$ is labeled with a unique integer $L(u)$ from the set $\{1, 2, \ldots, |V|\}$. For each vertex $u \in V$, let $R(u) = \{v \in V : u \rightsquigarrow v\}$ be the set of vertices that are reachable from u. Define $\min(u)$ to be the vertex in $R(u)$ whose label is minimum, that is, $\min(u)$ is the vertex v such that $L(v) = \min\{L(w) : w \in R(u)\}$. Give an $O(V + E)$-time algorithm that computes $\min(u)$ for all vertices $u \in V$.

20-5 Inserting and querying vertices in planar graphs

A ***planar*** graph is an undirected graph that can be drawn in the plane with no edges crossing. Euler proved that every planar graph has $|E| < 3|V|$.

Consider the following two operations on a planar graph G:

- INSERT$(G, v, \mathit{neighbors})$ inserts a new vertex v into G, where *neighbors* is an array (possibly empty) of vertices that have already been inserted into G and will become all the neighbors of v in G when v is inserted.
- NEWEST-NEIGHBOR(G, v) returns the neighbor of vertex v that was most recently inserted into G, or NIL if v has no neighbors.

Design a data structure that supports these two operations such that NEWEST-NEIGHBOR takes $O(1)$ worst-case time and INSERT takes $O(1)$ amortized time. Note that the length of the array *neighbors* given to INSERT may vary. (*Hint:* Use a potential function for the amortized analysis.)

Chapter notes

Even [137] and Tarjan [429] are excellent references for graph algorithms.

Breadth-first search was discovered by Moore [334] in the context of finding paths through mazes. Lee [280] independently discovered the same algorithm in the context of routing wires on circuit boards.

Hopcroft and Tarjan [226] advocated the use of the adjacency-list representation over the adjacency-matrix representation for sparse graphs and were the first to recognize the algorithmic importance of depth-first search. Depth-first search has been widely used since the late 1950s, especially in artificial intelligence programs.

Tarjan [426] gave a linear-time algorithm for finding strongly connected components. The algorithm for strongly connected components in Section 20.5 is adapted from Aho, Hopcroft, and Ullman [6], who credit it to S. R. Kosaraju (unpublished) and Sharir [408]. Dijkstra [117, Chapter 25] also developed an algorithm for strongly connected components that is based on contracting cycles. Subsequently, Gabow [163] rediscovered this algorithm. Knuth [259] was the first to give a linear-time algorithm for topological sorting.