## Introduction

Graphs are a pervasive data structure in computer science, and algorithms for working with them are fundamental to the field. There are hundreds of interesting computational problems defined in terms of graphs. In this part, we touch on a few of the more significant ones.

Chapter 22 shows how we can represent a graph on a computer and then discusses algorithms based on searching a graph using either breadth-first search or depth-first search. Two applications of depth-first search are given: topologically sorting a directed acyclic graph and decomposing a directed graph into its strongly connected components.

Chapter 23 describes how to compute a minimum-weight spanning tree of a graph. Such a tree is defined as the least-weight way of connecting all of the vertices together when each edge has an associated weight. The algorithms for computing minimum spanning trees are good examples of greedy algorithms (see Chapter 16).

Chapters 24 and 25 consider the problem of computing shortest paths between vertices when each edge has an associated length or "weight." Chapter 24 considers the computation of shortest paths from a given source vertex to all other vertices, and Chapter 25 considers the computation of shortest paths between every pair of vertices.

Finally, Chapter 26 shows how to compute a maximum flow of material in a network (directed graph) having a specified source of material, a specified sink, and specified capacities for the amount of material that can traverse each directed edge. This general problem arises in many forms, and a good algorithm for computing maximum flows can be used to solve a variety of related problems efficiently.

In describing the running time of a graph algorithm on a given graph $G = (V, E)$, we usually measure the size of the input in terms of the number of vertices $|V|$ and the number of edges $|E|$ of the graph. That is, there are two relevant parameters describing the size of the input, not just one. We adopt a common notational convention for these parameters. Inside asymptotic notation (such as $O$-notation or $\Theta$-notation), and *only* inside such notation, the symbol $V$ denotes $|V|$ and the symbol $E$ denotes $|E|$. For example, we might say, "the algorithm runs in time $O(VE)$," meaning that the algorithm runs in time $O(|V||E|)$. This convention makes the running-time formulas easier to read, without risk of ambiguity.

Another convention we adopt appears in pseudocode. We denote the vertex set of a graph $G$ by $V[G]$ and its edge set by $E[G]$. That is, the pseudocode views vertex and edge sets as attributes of a graph.

# 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-

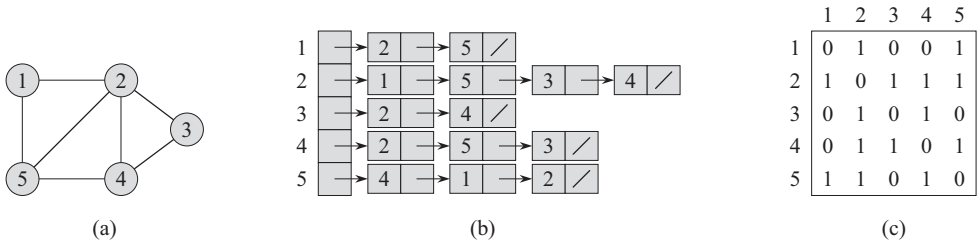



**Figure 22.1** Two representations of an undirected graph. **(a)** An undirected graph $G$ having five vertices and seven edges. **(b)** An adjacency-list representation of $G$. **(c)** The adjacency-matrix representation of $G$.

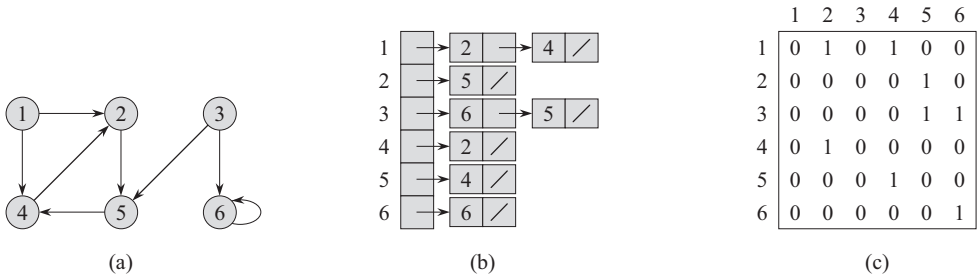



**Figure 22.2** Two representations of a directed graph. **(a)** A directed graph $G$ having six vertices and eight edges. **(b)** An adjacency-list representation of $G$. **(c)** The adjacency-matrix representation of $G$.

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 $A^{\mathrm{T}} = (a_{ij}^{\mathrm{T}})$ given by $a_{ij}^{\mathrm{T}} = a_{ji}$. 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 $\infty$.

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, $G^{\mathrm{T}}$ is $G$ with all its edges reversed. Describe efficient algorithms for computing $G^{\mathrm{T}}$ 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

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

Describe what the entries of the matrix product $BB^{\mathrm{T}}$ represent, where $B^{\mathrm{T}}$ 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 $\pi[u] = \text{NIL}$. 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)
 1  for each vertex u ∈ V[G] − {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       do u ← DEQUEUE(Q)
12          for each v ∈ Adj[u]
13               do if color[v] = WHITE
14                     then color[v] ← GRAY
15                          d[v] ← d[u] + 1
16                          π[v] ← u
17                          ENQUEUE(Q, v)
18          color[u] ← BLACK
```

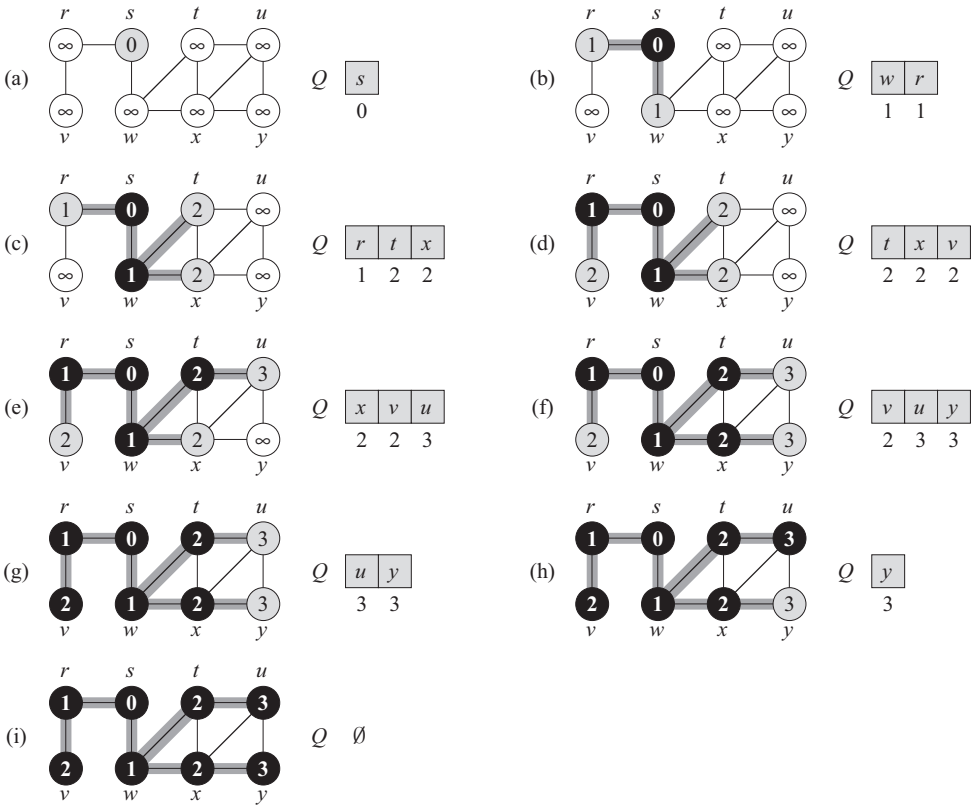



**Figure 22.3** The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex $u$ is shown $d[u]$. The queue $Q$ is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in 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 lines 11–18. 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 $(u, v) \in E$,

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

***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 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 $d[v] \geq \delta(s, v)$.

***Proof*** We use induction on the number of ENQUEUE operations. Our inductive hypothesis is that $d[v] \geq \delta(s, v)$ 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 $d[v] = \infty \geq \delta(s, v)$ 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 $d[u] \geq \delta(s, u)$. From the assignment performed by line 15 and from Lemma 22.1, we obtain

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

---

[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, $d[v_r] \leq d[v_1] + 1$ and $d[v_i] \leq d[v_{i+1}]$ 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, $d[v_1] \leq d[v_2]$. But then we have $d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$, 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 $d[v_1] \geq d[u]$. Thus, $d[v_{r+1}] = d[v] = d[u] + 1 \leq d[v_1] + 1$. From the inductive hypothesis, we also have $d[v_r] \leq d[u] + 1$, and so $d[v_r] \leq d[u] + 1 = d[v] = d[v_{r+1}]$, 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 $d[v_i] \leq d[v_j]$ 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 $\pi$ 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 $G_\pi$ is a ***breadth-first tree*** if $V_\pi$ 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 $G_\pi$ 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 $E_\pi$ 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 $\pi$ 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 $V_\pi$ consists of the vertices in $V$ reachable from $s$. Since $G_\pi$ forms a tree, by Theorem B.2, it contains a unique path from $s$ to each vertex in $V_\pi$. 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 $\pi$ 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 $\pi$ 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 $(V, E_\pi)$ from $s$ to $v$ is a shortest path in $G$, yet the set of edges $E_\pi$ 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 is it 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 $E_\pi$ 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 $u$,

$$d[u] < f[u] . \tag{22.2}$$

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 ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)

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

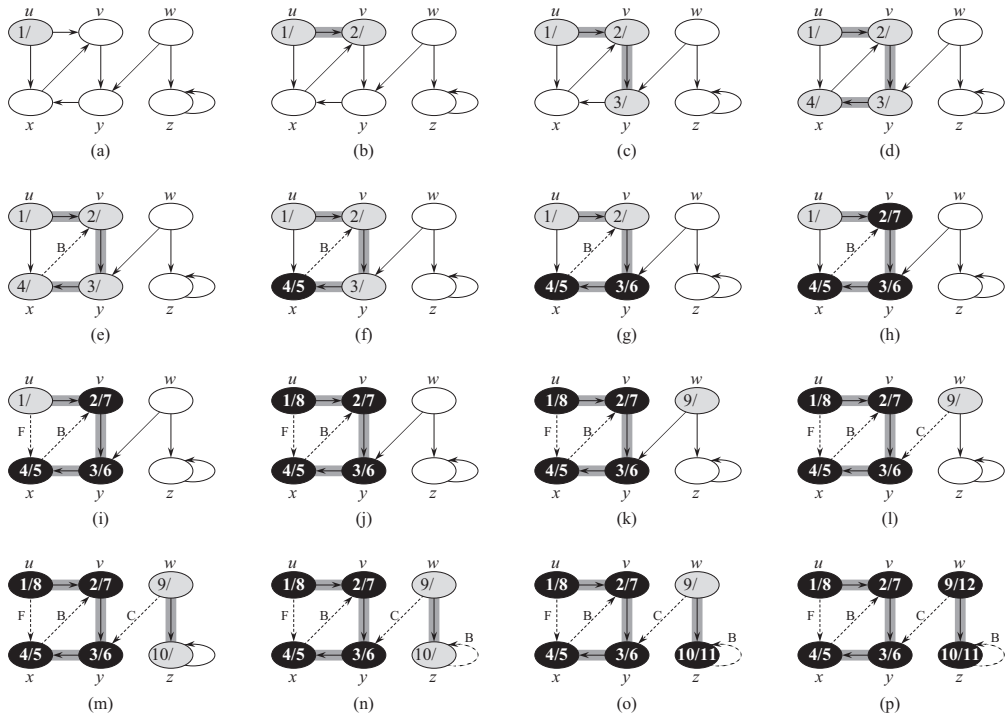



**Figure 22.4** The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward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

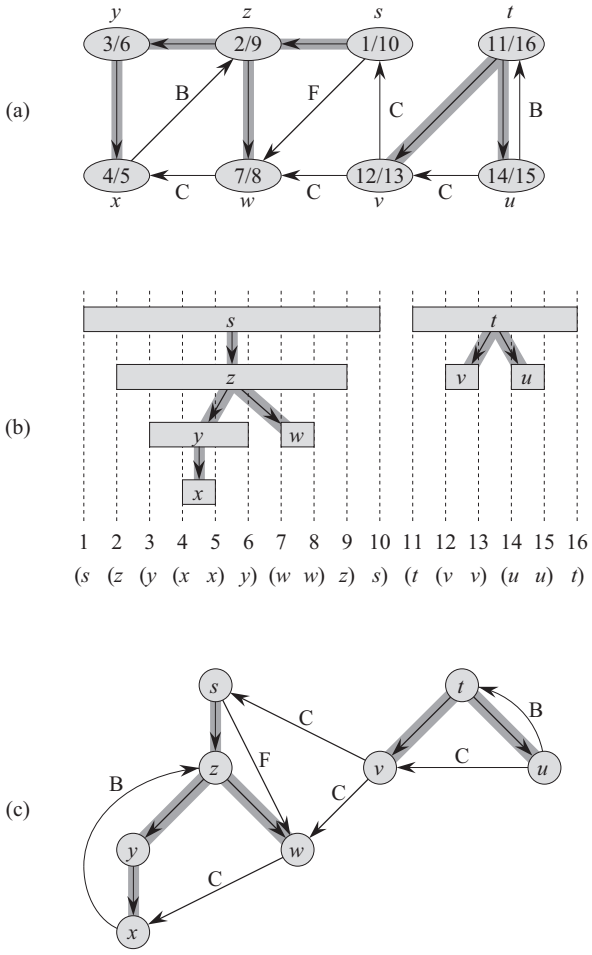



**Figure 22.5** Properties of depth-first search. **(a)** The result of a depth-first search of a directed graph. Vertices are timestamped and edge types are indicated as in Figure 22.4. **(b)** Intervals for the discovery time and finishing time of each vertex correspond to the parenthesization shown. 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 $G_\pi$ produced by a depth-first search on $G$.

1. ***Tree edges*** are edges in the depth-first forest $G_\pi$. 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.

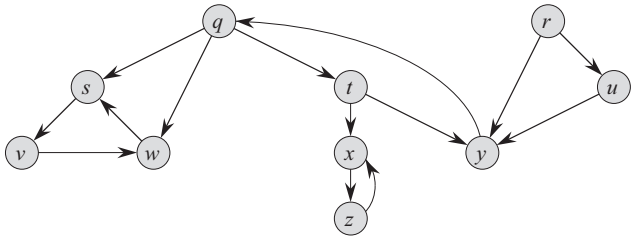



**Figure 22.6** A directed graph for use in Exercises 22.3-2 and 22.5-2.

***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.

***22.3-10***
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 $G$.

***22.3-11***
Show that a depth-first search of an undirected graph $G$ can be used to identify the connected components of $G$, and 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 each vertex $v$ is assigned an integer label $cc[v]$ between 1 and $k$, where $k$ is the number of connected components of $G$, such that $cc[u] = cc[v]$ if and only if $u$ and $v$ are in the same connected component.

***22.3-12*** ⋆
A directed graph $G = (V, E)$ is ***singly connected*** if $u \rightsquigarrow v$ implies that there is at most one simple path from $u$ to $v$ for all vertices $u, v \in V$. Give an efficient algorithm to determine whether or not a directed graph is singly connected.

## 22.4 Topological sort

This section shows how depth-first search can be used 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. (If the graph is not acyclic, then no linear ordering is possible.) A topological sort of a graph can be viewed 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.

Directed acyclic graphs are used in many applications to indicate precedences among events. Figure 22.7 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 22.7(a) indicates that garment $u$ must be donned before garment $v$. A topological sort of this dag therefore gives an order for getting dressed. Figure 22.7(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 following simple algorithm topologically sorts a dag.

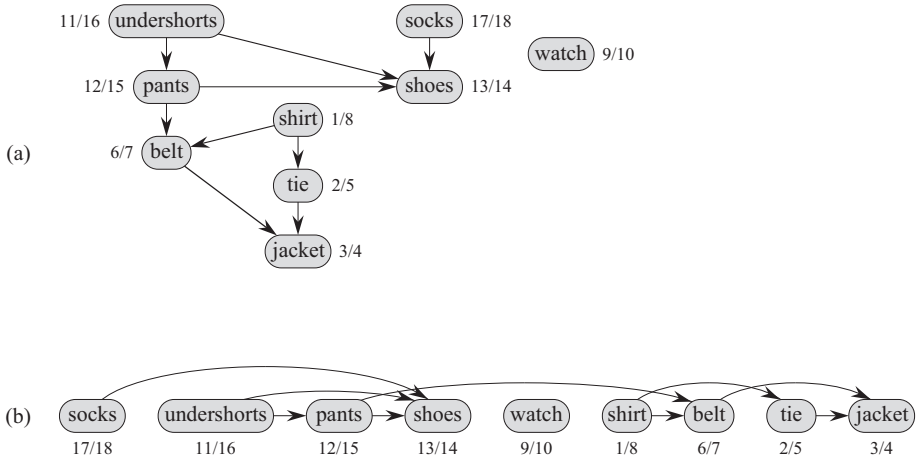



**Figure 22.7** **(a)** Professor Bumstead topologically sorts his clothing when getting dressed. Each directed edge $(u, v)$ means that garment $u$ must be put on before garment $v$. The discovery and finishing times from a depth-first search are shown next to each vertex. **(b)** The same graph shown topologically sorted. Its vertices are arranged from left to right in order of decreasing finishing time. Note that all directed edges go from left to right.

TOPOLOGICAL-SORT($G$)

1 call DFS($G$) to compute finishing times $f[v]$ 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

Figure 22.7(b) shows how the topologically sorted vertices appear in reverse order of their finishing times.

We can perform a topological sort in time $\Theta(V + E)$, 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.

We prove the correctness of this algorithm using the following key lemma characterizing directed acyclic graphs.

***Lemma 22.11***

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

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

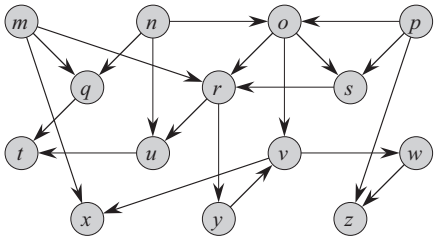



**Figure 22.8** A dag for topological sorting.

$\Leftarrow$: 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 $d[v]$, 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 22.12***
TOPOLOGICAL-SORT$(G)$ produces a topological sort of a directed acyclic graph $G$.

***Proof*** Suppose that DFS is run on a given dag $G = (V, E)$ to determine finishing times for its vertices. It suffices to show that for any pair of distinct vertices $u, v \in V$, if there is an edge in $G$ from $u$ to $v$, then $f[v] < f[u]$. 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 22.11. Therefore, $v$ must be either white or black. If $v$ is white, it becomes a descendant of $u$, and so $f[v] < f[u]$. If $v$ is black, it has already been finished, so that $f[v]$ has already been set. Because we are still exploring from $u$, we have yet to assign a timestamp to $f[u]$, and so once we do, we will have $f[v] < f[u]$ as well. Thus, for any edge $(u, v)$ in the dag, we have $f[v] < f[u]$, proving the theorem. ■

## Exercises

***22.4-1***
Show the ordering of vertices produced by TOPOLOGICAL-SORT when it is run on the dag of Figure 22.8, under the assumption of Exercise 22.3-2.

***22.4-2***
Give a linear-time algorithm that takes as input a directed acyclic graph $G = (V, E)$ and two vertices $s$ and $t$, and returns the number of paths from $s$ to $t$ in $G$. For example, in the directed acyclic graph of Figure 22.8, there are exactly four paths from vertex $p$ to vertex $v$: $pov$, $poryv$, $posryv$, and $psryv$. (Your algorithm only needs to count the paths, not list them.)

***22.4-3***
Give an algorithm that determines whether or not a given undirected graph $G = (V, E)$ contains a cycle. Your algorithm should run in $O(V)$ time, independent of $|E|$.

***22.4-4***
Prove or disprove: If a directed graph $G$ contains cycles, then TOPOLOGICAL-SORT$(G)$ produces a vertex ordering that minimizes the number of "bad" edges that are inconsistent with the ordering produced.

***22.4-5***
Another way to perform topological sorting on 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?

## 22.5 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 this decomposition using two depth-first searches. Many algorithms that work with directed graphs begin with such a decomposition. After decomposition, the algorithm is run separately on each strongly connected component. The solutions are then combined according to the structure of connections between 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 $u$ and $v$ in $C$, we have both $u \rightsquigarrow v$ and $v \rightsquigarrow u$; that is, vertices $u$ and $v$ are reachable from each other. Figure 22.9 shows an example.

Our algorithm for finding strongly connected components of a graph $G = (V, E)$ uses the transpose of $G$, which is defined in Exercise 22.1-3 to be the graph $G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, where $E^{\mathrm{T}} = \{(u, v) : (v, u) \in E\}$. That is, $E^{\mathrm{T}}$ consists of the edges of $G$ with their directions reversed. Given an adjacency-list representation of $G$, the time to create $G^{\mathrm{T}}$ is $O(V + E)$. It is interesting to observe that $G$ and $G^{\mathrm{T}}$ have

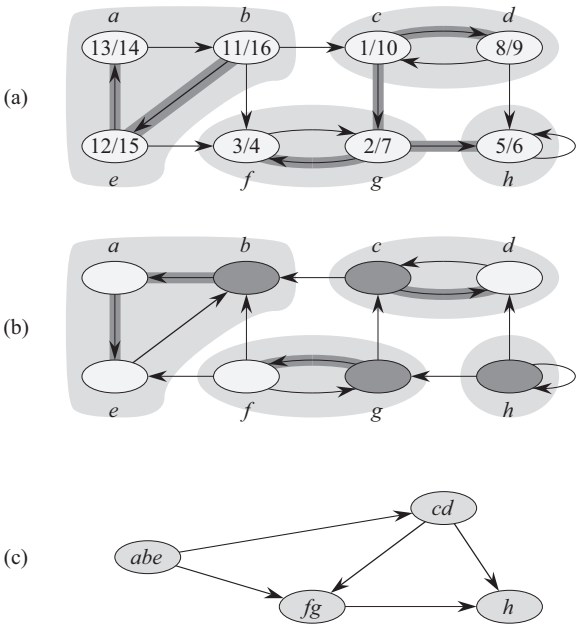



**Figure 22.9** **(a)** A directed graph $G$. The strongly connected components of $G$ are shown as shaded regions. Each vertex is labeled with its discovery and finishing times. Tree edges are shaded. **(b)** The graph $G^{\mathrm{T}}$, the transpose of $G$. The depth-first forest computed in line 3 of STRONGLY-CONNECTED-COMPONENTS is shown, with tree edges shaded. Each strongly connected component corresponds to one depth-first tree. Vertices $b$, $c$, $g$, and $h$, which are heavily shaded, are the roots of the depth-first trees produced by the depth-first search of $G^{\mathrm{T}}$. **(c)** The acyclic component graph $G^{\mathrm{SCC}}$ obtained by contracting all edges within each strongly connected component of $G$ so that only a single vertex remains in each component.

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 $G^{\mathrm{T}}$. Figure 22.9(b) shows the transpose of the graph in Figure 22.9(a), with the strongly connected components shaded.

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

STRONGLY-CONNECTED-COMPONENTS($G$)
1 call DFS($G$) to compute finishing times $f[u]$ for each vertex $u$
2 compute $G^{\mathrm{T}}$
3 call DFS($G^{\mathrm{T}}$), but in the main loop of DFS, consider the vertices in order of decreasing $f[u]$ (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

The idea behind this algorithm comes from a key property of the ***component graph*** $G^{\mathrm{SCC}} = (V^{\mathrm{SCC}}, E^{\mathrm{SCC}})$, which we define as follows. Suppose that $G$ has strongly connected components $C_1, C_2, \ldots, C_k$. The vertex set $V^{\mathrm{SCC}}$ is $\{v_1, v_2, \ldots, v_k\}$, and it contains a 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, by contracting all edges whose incident vertices are within the same strongly connected component of $G$, the resulting graph is $G^{\mathrm{SCC}}$. Figure 22.9(c) shows the component graph of the graph in Figure 22.9(a).

The key property is that the component graph is a dag, which the following lemma implies.

***Lemma 22.13***
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 there is a path $u \rightsquigarrow u'$ in $G$. Then there cannot also be a path $v' \rightsquigarrow v$ in $G$.

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

We shall see that by considering vertices in the second depth-first search in decreasing order of the finishing times that were computed in the first depth-first search, we are, in essence, visiting the vertices of the component graph (each of which corresponds to a strongly connected component of $G$) in topologically sorted order.

Because STRONGLY-CONNECTED-COMPONENTS performs two depth-first searches, there is the potential for ambiguity when we discuss $d[u]$ or $f[u]$. In this section, these values always refer to the discovery and finishing times as computed by the first call of DFS, in line 1.

We extend the notation for discovery and finishing times to sets of vertices. If $U \subseteq V$, then we define $d(U) = \min_{u \in U} \{d[u]\}$ and $f(U) = \max_{u \in U} \{f[u]\}$. That is, $d(U)$ and $f(U)$ are the earliest discovery time and latest finishing time, respectively, of any vertex in $U$.

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

***Lemma 22.14***
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*** There are two cases, depending on which strongly connected component, $C$ or $C'$, had the first discovered vertex during the depth-first search.

If $d(C) < d(C')$, let $x$ be the first vertex discovered in $C$. At time $d[x]$, all vertices in $C$ and $C'$ are white. There is a path in $G$ 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 at time $d[x]$ from $x$ to $w$ in $G$ 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 22.8, $f[x] = f(C) > f(C')$.

If instead we have $d(C) > d(C')$, let $y$ be the first vertex discovered in $C'$. At time $d[y]$, all vertices in $C'$ are white and there is a path in $G$ 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 22.8, $f[y] = f(C')$. At time $d[y]$, all vertices in $C$ are white. Since there is an edge $(u, v)$ from $C$ to $C'$, Lemma 22.13 implies that there cannot be a path from $C'$ to $C$. Hence, no vertex in $C$ is reachable from $y$. At time $f[y]$, therefore, all vertices in $C$ are still white. Thus, for any vertex $w \in C$, we have $f[w] > f[y]$, which implies that $f(C) > f(C')$. ■

The following corollary tells us that each edge in $G^T$ that goes between different strongly connected components goes from a component with an earlier finishing time (in the first depth-first search) to a component with a later finishing time.

***Corollary 22.15***
Let $C$ and $C'$ be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.

***Proof*** Since $(u, v) \in E^T$, we have $(v, u) \in E$. Since the strongly connected components of $G$ and $G^T$ are the same, Lemma 22.14 implies that $f(C) < f(C')$. ■

Corollary 22.15 provides the key to understanding why the STRONGLY-CONNECTED-COMPONENTS procedure works. Let us examine what happens when we perform the second depth-first search, which is on $G^T$. We start with the strongly connected component $C$ whose finishing time $f(C)$ is maximum. The

search starts from some vertex $x \in C$, and it visits all vertices in $C$. By Corollary 22.15, there are no edges in $G^T$ from $C$ to any other strongly connected component, and so the search from $x$ will not visit 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 search in line 3 selects as a root a vertex from some other strongly connected component $C'$ whose finishing time $f(C')$ is maximum over all components other than $C$. Again, the search will visit all vertices in $C'$, but by Corollary 22.15, the only edges in $G^T$ from $C'$ to any other component must be to $C$, which we have 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 were already visited. Each depth-first tree, therefore, will be exactly one strongly connected component. The following theorem formalizes this argument.

***Theorem 22.16***

STRONGLY-CONNECTED-COMPONENTS($G$) correctly computes the strongly connected components of a directed graph $G$.

***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 we choose roots in the depth-first search in line 3, $f[u] = f(C) > f(C')$ 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 22.15, 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 $C$ will be a descendant of $u$ during the depth-first search of $G^T$. Thus, 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 $(G^T)^{SCC}$ of $G^T$. If we map each strongly connected component visited in the second depth-first search to a vertex of $(G^T)^{SCC}$, the vertices of $(G^T)^{SCC}$ are visited in the reverse of a topologically sorted order. If we re-

verse the edges of $(G^{\mathrm{T}})^{\mathrm{SCC}}$, we get the graph $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}}$. Because $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}} = G^{\mathrm{SCC}}$ (see Exercise 22.5-4), the second depth-first search visits the vertices of $G^{\mathrm{SCC}}$ in topologically sorted order.

## Exercises

***22.5-1***
How can the number of strongly connected components of a graph change if a new edge is added?

***22.5-2***
Show how the procedure STRONGLY-CONNECTED-COMPONENTS works on the graph of Figure 22.6. Specifically, show the finishing 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.

***22.5-3***
Professor Deaver claims that the algorithm for strongly connected components can be simplified by using the original (instead of the transpose) graph in the second depth-first search and scanning the vertices in order of *increasing* finishing times. Is the professor correct?

***22.5-4***
Prove that for any directed graph $G$, we have $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}} = G^{\mathrm{SCC}}$. That is, the transpose of the component graph of $G^{\mathrm{T}}$ is the same as the component graph of $G$.

***22.5-5***
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.

***22.5-6***
Given a directed graph $G = (V, E)$, explain how to create another graph $G' = (V, E')$ such that (a) $G'$ has the same strongly connected components as $G$, (b) $G'$ has the same component graph as $G$, and (c) $E'$ is as small as possible. Describe a fast algorithm to compute $G'$.

***22.5-7***
A directed graph $G = (V, E)$ is said to be ***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 or not $G$ is semiconnected. Prove that your algorithm is correct, and analyze its running time.

## Problems

### *22-1 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.

***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. For each tree edge $(u, v)$, we have $d[v] = d[u] + 1$.
3. For each cross edge $(u, v)$, we have $d[v] = d[u]$ or $d[v] = d[u] + 1$.

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

1. There are no forward edges.
2. For each tree edge $(u, v)$, we have $d[v] = d[u] + 1$.
3. For each cross edge $(u, v)$, we have $d[v] \leq d[u] + 1$.
4. For each back edge $(u, v)$, we have $0 \leq d[v] \leq d[u]$.

### *22-2 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 22.10 illustrates these definitions. We 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 $G_\pi$ is an articulation point of $G$ if and only if it has at least two children in $G_\pi$.

***b.*** Let $v$ be a nonroot vertex of $G_\pi$. 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

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

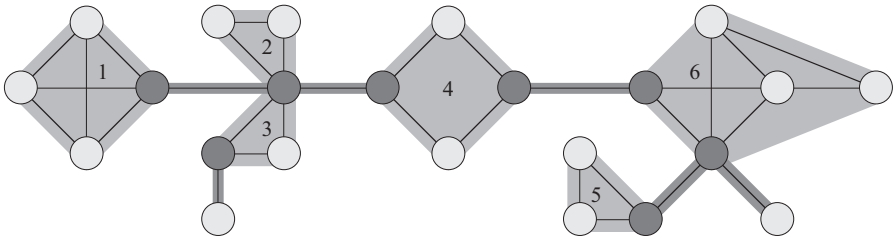



**Figure 22.10** The articulation points, bridges, and biconnected components of a connected, undirected graph for use in Problem 22-2. The articulation points are the heavily shaded vertices, the bridges are the heavily shaded edges, and the biconnected components are the edges in the shaded regions, with a *bcc* numbering shown.

Show how to compute $low[v]$ 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 $bcc[e]$ such that $bcc[e] = bcc[e']$ if and only if $e$ and $e'$ are in the same biconnected component.

### *22-3 Euler tour*

An ***Euler tour*** of a 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.)

### *22-4 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, i.e., $\min(u)$ 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 $u \in V$.

## Chapter notes

Even [87] and Tarjan [292] are excellent references for graph algorithms.

Breadth-first search was discovered by Moore [226] in the context of finding paths through mazes. Lee [198] independently discovered the same algorithm in the context of routing wires on circuit boards.

Hopcroft and Tarjan [154] 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 1950's, especially in artificial intelligence programs.

Tarjan [289] gave a linear-time algorithm for finding strongly connected components. The algorithm for strongly connected components in Section 22.5 is adapted from Aho, Hopcroft, and Ullman [6], who credit it to S. R. Kosaraju (unpublished) and M. Sharir [276]. Gabow [101] also developed an algorithm for strongly connected components that is based on contracting cycles and uses two stacks to make it run in linear time. Knuth [182] was the first to give a linear-time algorithm for topological sorting.

# 23 Minimum Spanning Trees

In the design of electronic circuitry, it is often necessary to make the pins of several components electrically equivalent by wiring them together. To interconnect a set of $n$ pins, we can use an arrangement of $n - 1$ wires, each connecting two pins. Of all such arrangements, the one that uses the least amount of wire is usually the most desirable.

We can model this wiring problem with a connected, undirected graph $G = (V, E)$, where $V$ is the set of pins, $E$ is the set of possible interconnections between pairs of pins, and for each edge $(u, v) \in E$, we have a weight $w(u, v)$ specifying the cost (amount of wire needed) to connect $u$ and $v$. We then wish to find an acyclic subset $T \subseteq E$ that connects all of the vertices and whose total weight

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

is minimized. Since $T$ is acyclic and connects all of the vertices, it must form a tree, which we call a ***spanning tree*** since it "spans" the graph $G$. We call the problem of determining the tree $T$ the ***minimum-spanning-tree problem***.[1] Figure 23.1 shows an example of a connected graph and its minimum spanning tree.

In this chapter, we shall examine two algorithms for solving the minimum-spanning-tree problem: Kruskal's algorithm and Prim's algorithm. Each can easily be made to run in time $O(E \lg V)$ using ordinary binary heaps. By using Fibonacci heaps, Prim's algorithm can be sped up to run in time $O(E + V \lg V)$, which is an improvement if $|V|$ is much smaller than $|E|$.

The two algorithms are greedy algorithms, as described in Chapter 16. At each step of an algorithm, one of several possible choices must be made. The greedy strategy advocates making the choice that is the best at the moment. Such a strategy is not generally guaranteed to find globally optimal solutions to problems. For

[1] The phrase "minimum spanning tree" is a shortened form of the phrase "minimum-weight spanning tree." We are not, for example, minimizing the number of edges in $T$, since all spanning trees have exactly $|V| - 1$ edges by Theorem B.2.

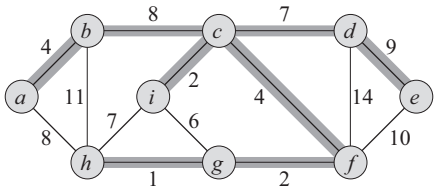



**Figure 23.1** A minimum spanning tree for a connected graph. The weights on edges are shown, and the edges in a minimum spanning tree are shaded. The total weight of the tree shown is 37. This minimum spanning tree is not unique: removing the edge $(b, c)$ and replacing it with the edge $(a, h)$ yields another spanning tree with weight 37.

the minimum-spanning-tree problem, however, we can prove that certain greedy strategies do yield a spanning tree with minimum weight. Although the present chapter can be read independently of Chapter 16, the greedy methods presented here are a classic application of the theoretical notions introduced there.

Section 23.1 introduces a "generic" minimum-spanning-tree algorithm that grows a spanning tree by adding one edge at a time. Section 23.2 gives two ways to implement the generic algorithm. The first algorithm, due to Kruskal, is similar to the connected-components algorithm from Section 21.1. The second, due to Prim, is similar to Dijkstra's shortest-paths algorithm (Section 24.3).

## 23.1 Growing a minimum spanning tree

Assume that we have a connected, undirected graph $G = (V, E)$ with a weight function $w : E \to \mathbf{R}$, and we wish to find a minimum spanning tree for $G$. The two algorithms we consider in this chapter use a greedy approach to the problem, although they differ in how they apply this approach.

This greedy strategy is captured by the following "generic" algorithm, which grows the minimum spanning tree one edge at a time. The algorithm manages a set of edges $A$, maintaining the following loop invariant:

> Prior to each iteration, $A$ is a subset of some minimum spanning tree.

At each step, we determine an edge $(u, v)$ that can be added to $A$ without violating this invariant, in the sense that $A \cup \{(u, v)\}$ is also a subset of a minimum spanning tree. We call such an edge a ***safe edge*** for $A$, since it can be safely added to $A$ while maintaining the invariant.

```
GENERIC-MST(G, w)
1  A ← ∅
2  while A does not form a spanning tree
3      do find an edge (u, v) that is safe for A
4         A ← A ∪ {(u, v)}
5  return A
```

We use the loop invariant as follows:

**Initialization:** After line 1, the set $A$ trivially satisfies the loop invariant.

**Maintenance:** The loop in lines 2–4 maintains the invariant by adding only safe edges.

**Termination:** All edges added to $A$ are in a minimum spanning tree, and so the set $A$ is returned in line 5 must be a minimum spanning tree.

The tricky part is, of course, finding a safe edge in line 3. One must exist, since when line 3 is executed, the invariant dictates that there is a spanning tree $T$ such that $A \subseteq T$. Within the **while** loop body, $A$ must be a proper subset of $T$, and therefore there must be an edge $(u, v) \in T$ such that $(u, v) \notin A$ and $(u, v)$ is safe for $A$.

In the remainder of this section, we provide a rule (Theorem 23.1) for recognizing safe edges. The next section describes two algorithms that use this rule to find safe edges efficiently.

We first need some definitions. A ***cut*** $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of $V$. Figure 23.2 illustrates this notion. We say that an edge $(u, v) \in E$ ***crosses*** the cut $(S, V - S)$ if one of its endpoints is in $S$ and the other is in $V - S$. We say that a cut ***respects*** a set $A$ of edges if no edge in $A$ crosses the cut. An edge is a ***light edge*** crossing a cut if its weight is the minimum of any edge crossing the cut. Note that there can be more than one light edge crossing a cut in the case of ties. More generally, we say that an edge is a ***light edge*** satisfying a given property if its weight is the minimum of any edge satisfying the property.

Our rule for recognizing safe edges is given by the following theorem.

***Theorem 23.1***
Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function $w$ defined on $E$. Let $A$ be a subset of $E$ that is included in some minimum spanning tree for $G$, let $(S, V - S)$ be any cut of $G$ that respects $A$, and let $(u, v)$ be a light edge crossing $(S, V - S)$. Then, edge $(u, v)$ is safe for $A$.

***Proof*** Let $T$ be a minimum spanning tree that includes $A$, and assume that $T$ does not contain the light edge $(u, v)$, since if it does, we are done. We shall

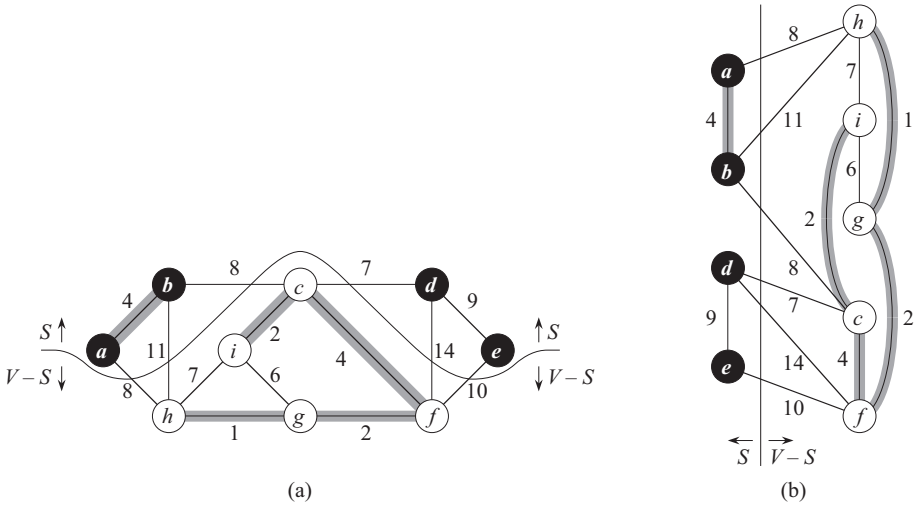



**Figure 23.2** Two ways of viewing a cut $(S, V - S)$ of the graph from Figure 23.1. **(a)** The vertices in the set $S$ are shown in black, and those in $V - S$ are shown in white. The edges crossing the cut are those connecting white vertices with black vertices. The edge $(d, c)$ is the unique light edge crossing the cut. A subset $A$ of the edges is shaded; note that the cut $(S, V - S)$ respects $A$, since no edge of $A$ crosses the cut. **(b)** The same graph with the vertices in the set $S$ on the left and the vertices in the set $V - S$ on the right. An edge crosses the cut if it connects a vertex on the left with a vertex on the right.

construct another minimum spanning tree $T'$ that includes $A \cup \{(u, v)\}$ by using a cut-and-paste technique, thereby showing that $(u, v)$ is a safe edge for $A$.

The edge $(u, v)$ forms a cycle with the edges on the path $p$ from $u$ to $v$ in $T$, as illustrated in Figure 23.3. Since $u$ and $v$ are on opposite sides of the cut $(S, V - S)$, there is at least one edge in $T$ on the path $p$ that also crosses the cut. Let $(x, y)$ be any such edge. The edge $(x, y)$ is not in $A$, because the cut respects $A$. Since $(x, y)$ is on the unique path from $u$ to $v$ in $T$, removing $(x, y)$ breaks $T$ into two components. Adding $(u, v)$ reconnects them to form a new spanning tree $T' = T - \{(x, y)\} \cup \{(u, v)\}$.

We next show that $T'$ is a minimum spanning tree. Since $(u, v)$ is a light edge crossing $(S, V - S)$ and $(x, y)$ also crosses this cut, $w(u, v) \leq w(x, y)$. Therefore,

$$\begin{aligned} w(T') &= w(T) - w(x, y) + w(u, v) \\ &\leq w(T) . \end{aligned}$$

But $T$ is a minimum spanning tree, so that $w(T) \leq w(T')$; thus, $T'$ must be a minimum spanning tree also.

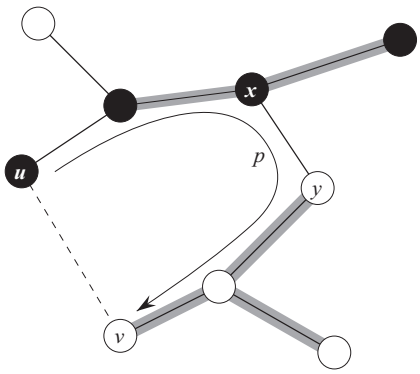



**Figure 23.3** The proof of Theorem 23.1. The vertices in $S$ are black, and the vertices in $V - S$ are white. The edges in the minimum spanning tree $T$ are shown, but the edges in the graph $G$ are not. The edges in $A$ are shaded, and $(u, v)$ is a light edge crossing the cut $(S, V - S)$. The edge $(x, y)$ is an edge on the unique path $p$ from $u$ to $v$ in $T$. A minimum spanning tree $T'$ that contains $(u, v)$ is formed by removing the edge $(x, y)$ from $T$ and adding the edge $(u, v)$.

It remains to show that $(u, v)$ is actually a safe edge for $A$. We have $A \subseteq T'$, since $A \subseteq T$ and $(x, y) \notin A$; thus, $A \cup \{(u, v)\} \subseteq T'$. Consequently, since $T'$ is a minimum spanning tree, $(u, v)$ is safe for $A$. ■

Theorem 23.1 gives us a better understanding of the workings of the GENERIC-MST algorithm on a connected graph $G = (V, E)$. As the algorithm proceeds, the set $A$ is always acyclic; otherwise, a minimum spanning tree including $A$ would contain a cycle, which is a contradiction. At any point in the execution of the algorithm, the graph $G_A = (V, A)$ is a forest, and each of the connected components of $G_A$ is a tree. (Some of the trees may contain just one vertex, as is the case, for example, when the algorithm begins: $A$ is empty and the forest contains $|V|$ trees, one for each vertex.) Moreover, any safe edge $(u, v)$ for $A$ connects distinct components of $G_A$, since $A \cup \{(u, v)\}$ must be acyclic.

The loop in lines 2–4 of GENERIC-MST is executed $|V| - 1$ times as each of the $|V| - 1$ edges of a minimum spanning tree is successively determined. Initially, when $A = \emptyset$, there are $|V|$ trees in $G_A$, and each iteration reduces that number by 1. When the forest contains only a single tree, the algorithm terminates.

The two algorithms in Section 23.2 use the following corollary to Theorem 23.1.

***Corollary 23.2***

Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function $w$ defined on $E$. Let $A$ be a subset of $E$ that is included in some minimum spanning tree for $G$, and let $C = (V_C, E_C)$ be a connected component (tree) in

the forest $G_A = (V, A)$. If $(u, v)$ is a light edge connecting $C$ to some other component in $G_A$, then $(u, v)$ is safe for $A$.

***Proof*** The cut $(V_C, V - V_C)$ respects $A$, and $(u, v)$ is a light edge for this cut. Therefore, $(u, v)$ is safe for $A$. ■

## Exercises

***23.1-1***
Let $(u, v)$ be a minimum-weight edge in a graph $G$. Show that $(u, v)$ belongs to some minimum spanning tree of $G$.

***23.1-2***
Professor Sabatier conjectures the following converse of Theorem 23.1. Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function $w$ defined on $E$. Let $A$ be a subset of $E$ that is included in some minimum spanning tree for $G$, let $(S, V - S)$ be any cut of $G$ that respects $A$, and let $(u, v)$ be a safe edge for $A$ crossing $(S, V - S)$. Then, $(u, v)$ is a light edge for the cut. Show that the professor's conjecture is incorrect by giving a counterexample.

***23.1-3***
Show that if an edge $(u, v)$ is contained in some minimum spanning tree, then it is a light edge crossing some cut of the graph.

***23.1-4***
Give a simple example of a graph such that the set of edges $\{(u, v) :$ there exists a cut $(S, V - S)$ such that $(u, v)$ is a light edge crossing $(S, V - S)\}$ does not form a minimum spanning tree.

***23.1-5***
Let $e$ be a maximum-weight edge on some cycle of $G = (V, E)$. Prove that there is a minimum spanning tree of $G' = (V, E - \{e\})$ that is also a minimum spanning tree of $G$. That is, there is a minimum spanning tree of $G$ that does not include $e$.

***23.1-6***
Show that a graph has a unique minimum spanning tree if, for every cut of the graph, there is a unique light edge crossing the cut. Show that the converse is not true by giving a counterexample.

***23.1-7***
Argue that if all edge weights of a graph are positive, then any subset of edges that connects all vertices and has minimum total weight must be a tree. Give an example

to show that the same conclusion does not follow if we allow some weights to be nonpositive.

***23.1-8***
Let $T$ be a minimum spanning tree of a graph $G$, and let $L$ be the sorted list of the edge weights of $T$. Show that for any other minimum spanning tree $T'$ of $G$, the list $L$ is also the sorted list of edge weights of $T'$.

***23.1-9***
Let $T$ be a minimum spanning tree of a graph $G = (V, E)$, and let $V'$ be a subset of $V$. Let $T'$ be the subgraph of $T$ induced by $V'$, and let $G'$ be the subgraph of $G$ induced by $V'$. Show that if $T'$ is connected, then $T'$ is a minimum spanning tree of $G'$.

***23.1-10***
Given a graph $G$ and a minimum spanning tree $T$, suppose that we decrease the weight of one of the edges in $T$. Show that $T$ is still a minimum spanning tree for $G$. More formally, let $T$ be a minimum spanning tree for $G$ with edge weights given by weight function $w$. Choose one edge $(x, y) \in T$ and a positive number $k$, and define the weight function $w'$ by

$$w'(u, v) = \begin{cases} w(u, v) & \text{if } (u, v) \neq (x, y) , \\ w(x, y) - k & \text{if } (u, v) = (x, y) . \end{cases}$$

Show that $T$ is a minimum spanning tree for $G$ with edge weights given by $w'$.

***23.1-11*** ⋆
Given a graph $G$ and a minimum spanning tree $T$, suppose that we decrease the weight of one of the edges not in $T$. Give an algorithm for finding the minimum spanning tree in the modified graph.

## 23.2 The algorithms of Kruskal and Prim

The two minimum-spanning-tree algorithms described in this section are elaborations of the generic algorithm. They each use a specific rule to determine a safe edge in line 3 of GENERIC-MST. In Kruskal's algorithm, the set $A$ is a forest. The safe edge added to $A$ is always a least-weight edge in the graph that connects two distinct components. In Prim's algorithm, the set $A$ forms a single tree. The safe edge added to $A$ is always a least-weight edge connecting the tree to a vertex not in the tree.

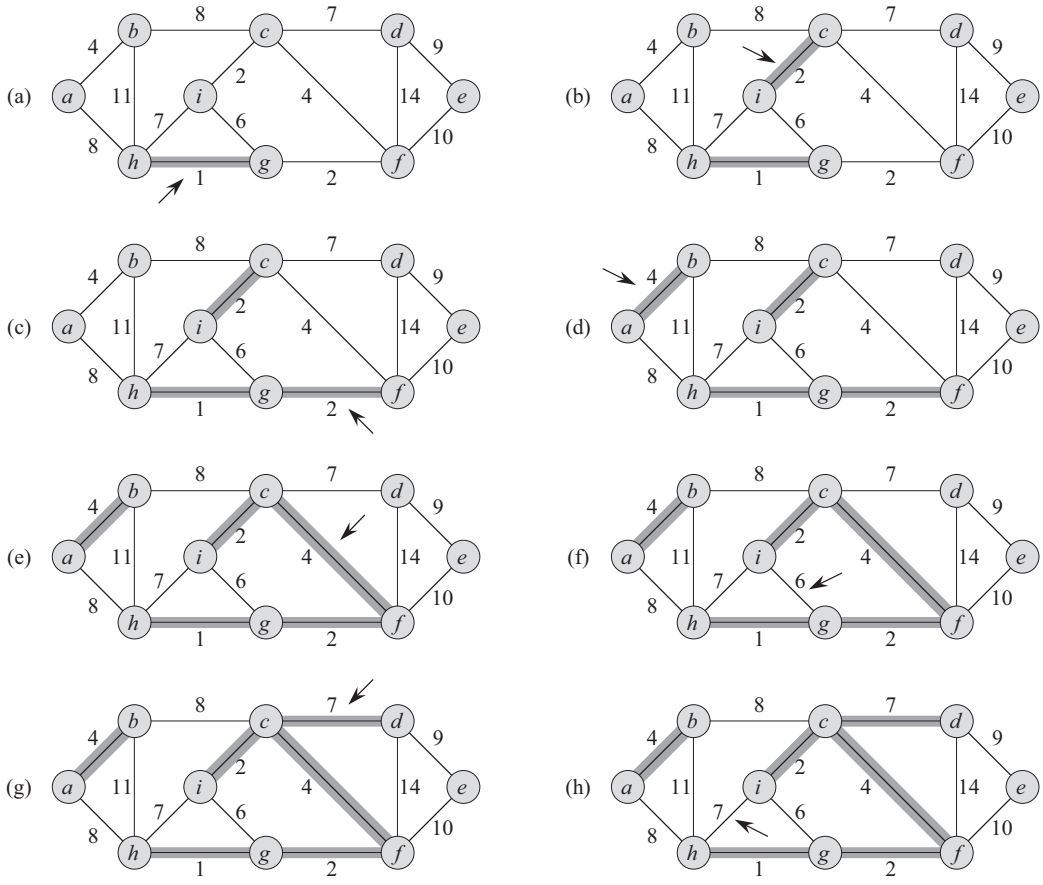
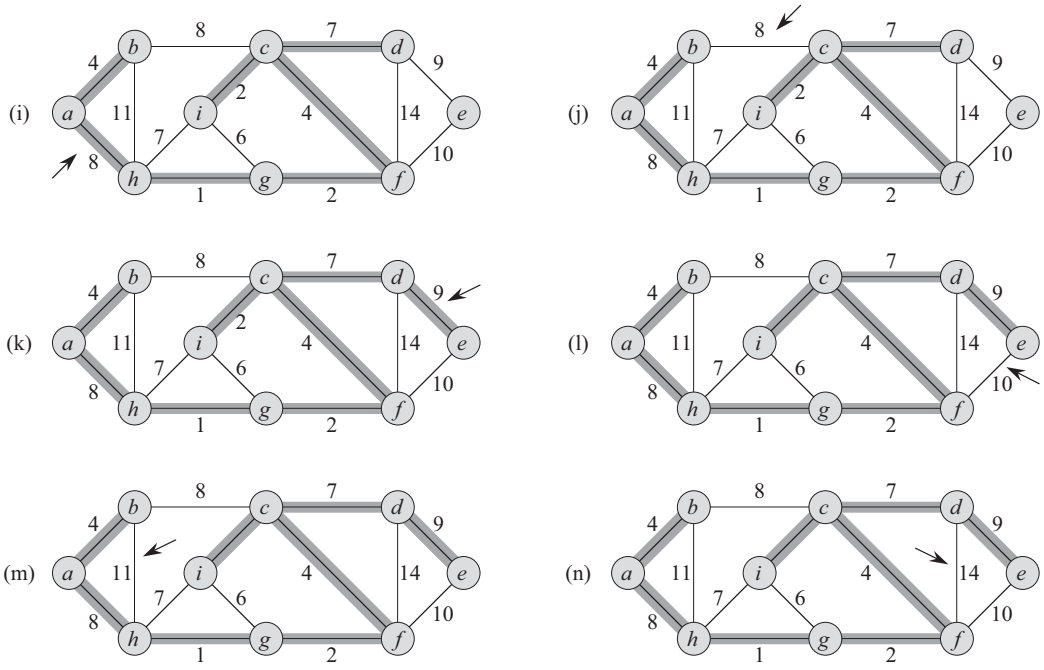



**Figure 23.4** The execution of Kruskal's algorithm on the graph from Figure 23.1. Shaded edges belong to the forest $A$ being grown. The edges are considered by the algorithm in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

### Kruskal's algorithm

Kruskal's algorithm is based directly on the generic minimum-spanning-tree algorithm given in Section 23.1. It finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge $(u, v)$ of least weight. Let $C_1$ and $C_2$ denote the two trees that are connected by $(u, v)$. Since $(u, v)$ must be a light edge connecting $C_1$ to some other tree, Corollary 23.2




implies that $(u, v)$ is a safe edge for $C_1$. Kruskal's algorithm is a greedy algorithm, because at each step it adds to the forest an edge of least possible weight.

Our implementation of Kruskal's algorithm is like the algorithm to compute connected components from Section 21.1. It uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in a tree of the current forest. The operation FIND-SET$(u)$ returns a representative element from the set that contains $u$. Thus, we can determine whether two vertices $u$ and $v$ belong to the same tree by testing whether FIND-SET$(u)$ equals FIND-SET$(v)$. The combining of trees is accomplished by the UNION procedure.

```
MST-KRUSKAL(G, w)
1  A ← ∅
2  for each vertex v ∈ V[G]
3       do MAKE-SET(v)
4  sort the edges of E into nondecreasing order by weight w
5  for each edge (u, v) ∈ E, taken in nondecreasing order by weight
6       do if FIND-SET(u) ≠ FIND-SET(v)
7            then A ← A ∪ {(u, v)}
8                 UNION(u, v)
9  return A
```

Kruskal's algorithm works as shown in Figure 23.4. Lines 1–3 initialize the set $A$ to the empty set and create $|V|$ trees, one containing each vertex. The edges in $E$ are sorted into nondecreasing order by weight in line 4. The **for** loop in lines 5–8 checks, for each edge $(u, v)$, whether the endpoints $u$ and $v$ belong to the same tree. If they do, then the edge $(u, v)$ cannot be added to the forest without creating a cycle, and the edge is discarded. Otherwise, the two vertices belong to different trees. In this case, the edge $(u, v)$ is added to $A$ in line 7, and the vertices in the two trees are merged in line 8.

The running time of Kruskal's algorithm for a graph $G = (V, E)$ depends on the implementation of the disjoint-set data structure. We shall assume the disjoint-set-forest implementation of Section 21.3 with the union-by-rank and path-compression heuristics, since it is the asymptotically fastest implementation known. Initializing the set $A$ in line 1 takes $O(1)$ time, and the time to sort the edges in line 4 is $O(E \lg E)$. (We will account for the cost of the $|V|$ MAKE-SET operations in the **for** loop of lines 2–3 in a moment.) The **for** loop of lines 5–8 performs $O(E)$ FIND-SET and UNION operations on the disjoint-set forest. Along with the $|V|$ MAKE-SET operations, these take a total of $O((V + E)\,\alpha(V))$ time, where $\alpha$ is the very slowly growing function defined in Section 21.4. Because $G$ is assumed to be connected, we have $|E| \geq |V| - 1$, and so the disjoint-set operations take $O(E\,\alpha(V))$ time. Moreover, since $\alpha(|V|) = O(\lg V) = O(\lg E)$, the total running time of Kruskal's algorithm is $O(E \lg E)$. Observing that $|E| < |V|^2$, we have $\lg |E| = O(\lg V)$, and so we can restate the running time of Kruskal's algorithm as $O(E \lg V)$.

### Prim's algorithm

Like Kruskal's algorithm, Prim's algorithm is a special case of the generic minimum-spanning-tree algorithm from Section 23.1. Prim's algorithm operates much like Dijkstra's algorithm for finding shortest paths in a graph, which we shall see in Section 24.3. Prim's algorithm has the property that the edges in the set $A$ always form a single tree. As is illustrated in Figure 23.5, the tree starts from an arbitrary root vertex $r$ and grows until the tree spans all the vertices in $V$. At each step, a light edge is added to the tree $A$ that connects $A$ to an isolated vertex of $G_A = (V, A)$. By Corollary 23.2, this rule adds only edges that are safe for $A$; therefore, when the algorithm terminates, the edges in $A$ form a minimum spanning tree. This strategy is greedy since the tree is augmented at each step with an edge that contributes the minimum amount possible to the tree's weight.

The key to implementing Prim's algorithm efficiently is to make it easy to select a new edge to be added to the tree formed by the edges in $A$. In the pseudocode below, the connected graph $G$ and the root $r$ of the minimum spanning tree to be grown are inputs to the algorithm. During execution of the algorithm, all vertices

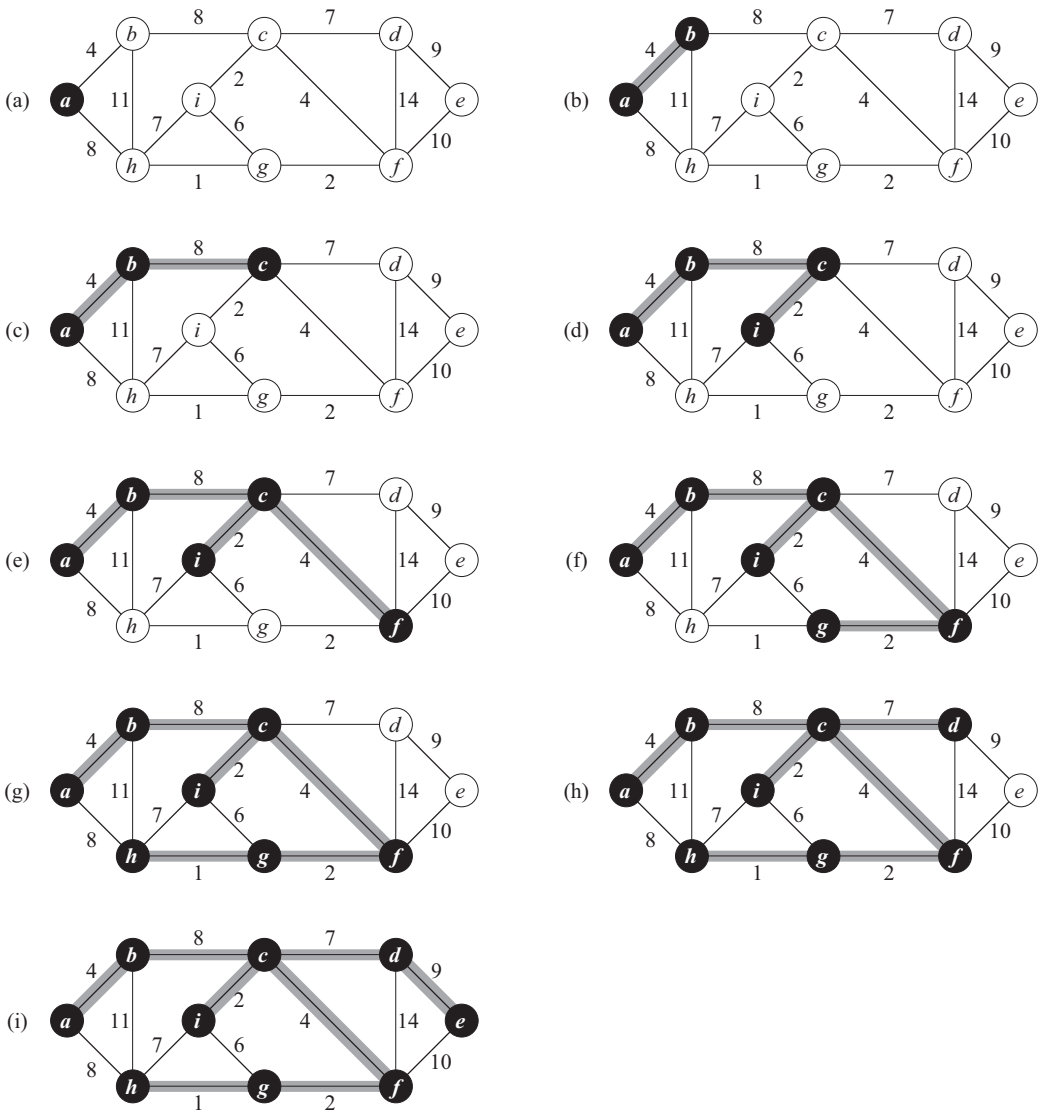



**Figure 23.5** The execution of Prim's algorithm on the graph from Figure 23.1. The root vertex is $a$. Shaded edges are in the tree being grown, and the vertices in the tree are shown in black. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree. In the second step, for example, the algorithm has a choice of adding either edge $(b, c)$ or edge $(a, h)$ to the tree since both are light edges crossing the cut.

that are *not* in the tree reside in a min-priority queue $Q$ based on a *key* field. For each vertex $v$, $key[v]$ is the minimum weight of any edge connecting $v$ to a vertex in the tree; by convention, $key[v] = \infty$ if there is no such edge. The field $\pi[v]$ names the parent of $v$ in the tree. During the algorithm, the set $A$ from GENERIC-MST is kept implicitly as

$$A = \{(v, \pi[v]) : v \in V - \{r\} - Q\} \ .$$

When the algorithm terminates, the min-priority queue $Q$ is empty; the minimum spanning tree $A$ for $G$ is thus

$$A = \{(v, \pi[v]) : v \in V - \{r\}\} \ .$$

```
MST-PRIM(G, w, r)
 1  for each u ∈ V[G]
 2       do key[u] ← ∞
 3          π[u] ← NIL
 4  key[r] ← 0
 5  Q ← V[G]
 6  while Q ≠ ∅
 7       do u ← EXTRACT-MIN(Q)
 8          for each v ∈ Adj[u]
 9               do if v ∈ Q and w(u, v) < key[v]
10                    then π[v] ← u
11                         key[v] ← w(u, v)
```

Prim's algorithm works as shown in Figure 23.5. Lines 1–5 set the key of each vertex to $\infty$ (except for the root $r$, whose key is set to 0 so that it will be the first vertex processed), set the parent of each vertex to NIL, and initialize the min-priority queue $Q$ to contain all the vertices. The algorithm maintains the following three-part loop invariant:

Prior to each iteration of the **while** loop of lines 6–11,

1. $A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$.
2. The vertices already placed into the minimum spanning tree are those in $V - Q$.
3. For all vertices $v \in Q$, if $\pi[v] \neq$ NIL, then $key[v] < \infty$ and $key[v]$ is the weight of a light edge $(v, \pi[v])$ connecting $v$ to some vertex already placed into the minimum spanning tree.

Line 7 identifies a vertex $u \in Q$ incident on a light edge crossing the cut $(V - Q, Q)$ (with the exception of the first iteration, in which $u = r$ due to line 4). Removing $u$

from the set $Q$ adds it to the set $V - Q$ of vertices in the tree, thus adding $(u, \pi[u])$ to $A$. The **for** loop of lines 8–11 update the *key* and $\pi$ fields of every vertex $v$ adjacent to $u$ but not in the tree. The updating maintains the third part of the loop invariant.

The performance of Prim's algorithm depends on how we implement the min-priority queue $Q$. If $Q$ is implemented as a binary min-heap (see Chapter 6), we can use the BUILD-MIN-HEAP procedure to perform the initialization in lines 1–5 in $O(V)$ time. The body of the **while** loop is executed $|V|$ times, and since each EXTRACT-MIN operation takes $O(\lg V)$ time, the total time for all calls to EXTRACT-MIN is $O(V \lg V)$. The **for** loop in lines 8–11 is executed $O(E)$ times altogether, since the sum of the lengths of all adjacency lists is $2|E|$. Within the **for** loop, the test for membership in $Q$ in line 9 can be implemented in constant time by keeping a bit for each vertex that tells whether or not it is in $Q$, and updating the bit when the vertex is removed from $Q$. The assignment in line 11 involves an implicit DECREASE-KEY operation on the min-heap, which can be implemented in a binary min-heap in $O(\lg V)$ time. Thus, the total time for Prim's algorithm is $O(V \lg V + E \lg V) = O(E \lg V)$, which is asymptotically the same as for our implementation of Kruskal's algorithm.

The asymptotic running time of Prim's algorithm can be improved, however, by using Fibonacci heaps. Chapter 20 shows that if $|V|$ elements are organized into a Fibonacci heap, we can perform an EXTRACT-MIN operation in $O(\lg V)$ amortized time and a DECREASE-KEY operation (to implement line 11) in $O(1)$ amortized time. Therefore, if we use a Fibonacci heap to implement the min-priority queue $Q$, the running time of Prim's algorithm improves to $O(E + V \lg V)$.

## Exercises

### *23.2-1*

Kruskal's algorithm can return different spanning trees for the same input graph $G$, depending on how ties are broken when the edges are sorted into order. Show that for each minimum spanning tree $T$ of $G$, there is a way to sort the edges of $G$ in Kruskal's algorithm so that the algorithm returns $T$.

### *23.2-2*

Suppose that the graph $G = (V, E)$ is represented as an adjacency matrix. Give a simple implementation of Prim's algorithm for this case that runs in $O(V^2)$ time.

### *23.2-3*

Is the Fibonacci-heap implementation of Prim's algorithm asymptotically faster than the binary-heap implementation for a sparse graph $G = (V, E)$, where $|E| = \Theta(V)$? What about for a dense graph, where $|E| = \Theta(V^2)$? How must

$|E|$ and $|V|$ be related for the Fibonacci-heap implementation to be asymptotically faster than the binary-heap implementation?

***23.2-4***

Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Kruskal's algorithm run? What if the edge weights are integers in the range from 1 to $W$ for some constant $W$?

***23.2-5***

Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Prim's algorithm run? What if the edge weights are integers in the range from 1 to $W$ for some constant $W$?

***23.2-6*** ⋆

Suppose that the edge weights in a graph are uniformly distributed over the half-open interval $[0, 1)$. Which algorithm, Kruskal's or Prim's, can you make run faster?

***23.2-7*** ⋆

Suppose that a graph $G$ has a minimum spanning tree already computed. How quickly can the minimum spanning tree be updated if a new vertex and incident edges are added to $G$?

***23.2-8***

Professor Toole proposes a new divide-and-conquer algorithm for computing minimum spanning trees, which goes as follows. Given a graph $G = (V, E)$, partition the set $V$ of vertices into two sets $V_1$ and $V_2$ such that $|V_1|$ and $|V_2|$ differ by at most 1. Let $E_1$ be the set of edges that are incident only on vertices in $V_1$, and let $E_2$ be the set of edges that are incident only on vertices in $V_2$. Recursively solve a minimum-spanning-tree problem on each of the two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Finally, select the minimum-weight edge in $E$ that crosses the cut $(V_1, V_2)$, and use this edge to unite the resulting two minimum spanning trees into a single spanning tree.

Either argue that the algorithm correctly computes a minimum spanning tree of $G$, or provide an example for which the algorithm fails.

## Problems

***23-1 Second-best minimum spanning tree***

Let $G = (V, E)$ be an undirected, connected graph with weight function $w : E \to \mathbf{R}$, and suppose that $|E| \geq |V|$ and all edge weights are distinct.

A second-best minimum spanning tree is defined as follows. Let $\mathcal{T}$ be the set of all spanning trees of $G$, and let $T'$ be a minimum spanning tree of $G$. Then a ***second-best minimum spanning tree*** is a spanning tree $T$ such that $w(T) = \min_{T'' \in \mathcal{T} - \{T'\}} \{w(T'')\}$.

***a.*** Show that the minimum spanning tree is unique, but that the second-best minimum spanning tree need not be unique.

***b.*** Let $T$ be a minimum spanning tree of $G$. Prove that there exist edges $(u, v) \in T$ and $(x, y) \notin T$ such that $T - \{(u, v)\} \cup \{(x, y)\}$ is a second-best minimum spanning tree of $G$.

***c.*** Let $T$ be a spanning tree of $G$ and, for any two vertices $u, v \in V$, let $max[u, v]$ be an edge of maximum weight on the unique path between $u$ and $v$ in $T$. Describe an $O(V^2)$-time algorithm that, given $T$, computes $max[u, v]$ for all $u, v \in V$.

***d.*** Give an efficient algorithm to compute the second-best minimum spanning tree of $G$.

***23-2 Minimum spanning tree in sparse graphs***

For a very sparse connected graph $G = (V, E)$, we can further improve upon the $O(E + V \lg V)$ running time of Prim's algorithm with Fibonacci heaps by "preprocessing" $G$ to decrease the number of vertices before running Prim's algorithm. In particular, we choose, for each vertex $u$, the minimum-weight edge $(u, v)$ incident on $u$, and we put $(u, v)$ into the minimum spanning tree under construction. We then contract all chosen edges (see Section B.4). Rather than contracting these edges one at a time, we first identify sets of vertices that are united into the same new vertex. Then, we create the graph that would have resulted from contracting these edges one at a time, but we do so by "renaming" edges according to the sets into which their endpoints were placed. Several edges from the original graph may be renamed the same as each other. In such a case, only one edge results, and its weight is the minimum of the weights of the corresponding original edges.

Initially, we set the minimum spanning tree $T$ being constructed to be empty, and for each edge $(u, v) \in E$, we set $orig[u, v] = (u, v)$ and $c[u, v] = w(u, v)$. We use the *orig* attribute to reference the edge from the initial graph that is associated with

an edge in the contracted graph. The $c$ attribute holds the weight of an edge, and as edges are contracted, it is updated according to the above scheme for choosing edge weights. The procedure MST-REDUCE takes inputs $G$, $orig$, $c$, and $T$, and it returns a contracted graph $G'$ and updated attributes $orig'$ and $c'$ for graph $G'$. The procedure also accumulates edges of $G$ into the minimum spanning tree $T$.

```
MST-REDUCE(G, orig, c, T)
 1  for each v ∈ V[G]
 2       do mark[v] ← FALSE
 3          MAKE-SET(v)
 4  for each u ∈ V[G]
 5       do if mark[u] = FALSE
 6             then choose v ∈ Adj[u] such that c[u, v] is minimized
 7                  UNION(u, v)
 8                  T ← T ∪ {orig[u, v]}
 9                  mark[u] ← mark[v] ← TRUE
10  V[G'] ← {FIND-SET(v) : v ∈ V[G]}
11  E[G'] ← ∅
12  for each (x, y) ∈ E[G]
13       do u ← FIND-SET(x)
14          v ← FIND-SET(y)
15          if (u, v) ∉ E[G']
16            then E[G'] ← E[G'] ∪ {(u, v)}
17                 orig'[u, v] ← orig[x, y]
18                 c'[u, v] ← c[x, y]
19            else if c[x, y] < c'[u, v]
20                   then orig'[u, v] ← orig[x, y]
21                        c'[u, v] ← c[x, y]
22  construct adjacency lists Adj for G'
23  return G', orig', c', and T
```

***a.*** Let $T$ be the set of edges returned by MST-REDUCE, and let $A$ be the minimum spanning tree of the graph $G'$ formed by the call MST-PRIM$(G', c', r)$, where $r$ is any vertex in $V[G']$. Prove that $T \cup \{orig'[x, y] : (x, y) \in A\}$ is a minimum spanning tree of $G$.

***b.*** Argue that $|V[G']| \leq |V|/2$.

***c.*** Show how to implement MST-REDUCE so that it runs in $O(E)$ time. (*Hint:* Use simple data structures.)

***d.*** Suppose that we run $k$ phases of MST-REDUCE, using the outputs $G'$, $orig'$, and $c'$ produced by one phase as the inputs $G$, $orig$, and $c$ to the next phase and

accumulating edges in $T$. Argue that the overall running time of the $k$ phases is $O(kE)$.

***e.*** Suppose that after running $k$ phases of MST-REDUCE, as in part (d), we run Prim's algorithm by calling MST-PRIM$(G', c', r)$, where $G'$ and $c'$ are returned by the last phase and $r$ is any vertex in $V[G']$. Show how to pick $k$ so that the overall running time is $O(E \lg\lg V)$. Argue that your choice of $k$ minimizes the overall asymptotic running time.

***f.*** For what values of $|E|$ (in terms of $|V|$) does Prim's algorithm with preprocessing asymptotically beat Prim's algorithm without preprocessing?

### *23-3 Bottleneck spanning tree*

A ***bottleneck spanning tree*** $T$ of an undirected graph $G$ is a spanning tree of $G$ whose largest edge weight is minimum over all spanning trees of $G$. We say that the value of the bottleneck spanning tree is the weight of the maximum-weight edge in $T$.

***a.*** Argue that a minimum spanning tree is a bottleneck spanning tree.

Part (a) shows that finding a bottleneck spanning tree is no harder than finding a minimum spanning tree. In the remaining parts, we will show that one can be found in linear time.

***b.*** Give a linear-time algorithm that given a graph $G$ and an integer $b$, determines whether the value of the bottleneck spanning tree is at most $b$.

***c.*** Use your algorithm for part (b) as a subroutine in a linear-time algorithm for the bottleneck-spanning-tree problem. (*Hint:* You may want to use a subroutine that contracts sets of edges, as in the MST-REDUCE procedure described in Problem 23-2.)

### *23-4 Alternative minimum-spanning-tree algorithms*

In this problem, we give pseudocode for three different algorithms. Each one takes a graph as input and returns a set of edges $T$. For each algorithm, you must either prove that $T$ is a minimum spanning tree or prove that $T$ is not a minimum spanning tree. Also describe the most efficient implementation of each algorithm, whether or not it computes a minimum spanning tree.

***a.*** MAYBE-MST-A$(G, w)$

```
1  sort the edges into nonincreasing order of edge weights w
2  T ← E
3  for each edge e, taken in nonincreasing order by weight
4       do if T − {e} is a connected graph
5            then T ← T − e
6  return T
```

***b.*** MAYBE-MST-B$(G, w)$

```
1  T ← ∅
2  for each edge e, taken in arbitrary order
3       do if T ∪ {e} has no cycles
4            then T ← T ∪ e
5  return T
```

***c.*** MAYBE-MST-C$(G, w)$

```
1  T ← ∅
2  for each edge e, taken in arbitrary order
3       do T ← T ∪ {e}
4          if T has a cycle c
5            then let e′ be the maximum-weight edge on c
6                 T ← T − {e′}
7  return T
```

## Chapter notes

Tarjan [292] surveys the minimum-spanning-tree problem and provides excellent advanced material. A history of the minimum-spanning-tree problem has been written by Graham and Hell [131].

Tarjan attributes the first minimum-spanning-tree algorithm to a 1926 paper by O. Borůvka. Borůvka's algorithm consists of running $O(\lg V)$ iterations of the procedure MST-REDUCE described in Problem 23-2. Kruskal's algorithm was reported by Kruskal [195] in 1956. The algorithm commonly known as Prim's algorithm was indeed invented by Prim [250], but it was also invented earlier by V. Jarník in 1930.

The reason why greedy algorithms are effective at finding minimum spanning trees is that the set of forests of a graph forms a graphic matroid. (See Section 16.4.)

When $|E| = \Omega(V \lg V)$, Prim's algorithm, implemented with Fibonacci heaps runs in $O(E)$ time. For sparser graphs, using a combination of the ideas from Prim's algorithm, Kruskal's algorithm, and Borůvka's algorithm, together with advanced data structures, Fredman and Tarjan [98] give an algorithm that runs in $O(E \lg^* V)$ time. Gabow, Galil, Spencer, and Tarjan [102] improved this algorithm to run in $O(E \lg \lg^* V)$ time. Chazelle [53] gives an algorithm that runs in $O(E\, \widehat{\alpha}(E, V))$ time, where $\widehat{\alpha}(E, V)$ is the functional inverse of Ackermann's function. (See the chapter notes for Chapter 21 for a brief discussion of Ackermann's function and its inverse.) Unlike previous minimum-spanning-tree algorithms, Chazelle's algorithm does not follow the greedy method.

A related problem is ***spanning tree verification***, in which we are given a graph $G = (V, E)$ and a tree $T \subseteq E$, and we wish to determine whether $T$ is a minimum spanning tree of $G$. King [177] gives a linear-time algorithm for spanning tree verification, building on earlier work of Komlós [188] and Dixon, Rauch, and Tarjan [77].

The above algorithms are all deterministic and fall into the comparison-based model described in Chapter 8. Karger, Klein, and Tarjan [169] give a randomized minimum-spanning-tree algorithm that runs in $O(V + E)$ expected time. This algorithm uses recursion in a manner similar to the linear-time selection algorithm in Section 9.3: a recursive call on an auxiliary problem identifies a subset of the edges $E'$ that cannot be in any minimum spanning tree. Another recursive call on $E - E'$ then finds the minimum spanning tree. The algorithm also uses ideas from Borůvka's algorithm and King's algorithm for spanning tree verification.

Fredman and Willard [100] showed how to find a minimum spanning tree in $O(V+E)$ time using a deterministic algorithm that is not comparison based. Their algorithm assumes that the data are $b$-bit integers and that the computer memory consists of addressable $b$-bit words.

# 24 Single-Source Shortest Paths

A motorist wishes to find the shortest possible route from Chicago to Boston. Given a road map of the United States on which the distance between each pair of adjacent intersections is marked, how can we determine this shortest route?

One possible way is to enumerate all the routes from Chicago to Boston, add up the distances on each route, and select the shortest. It is easy to see, however, that even if we disallow routes that contain cycles, there are millions of possibilities, most of which are simply not worth considering. For example, a route from Chicago to Houston to Boston is obviously a poor choice, because Houston is about a thousand miles out of the way.

In this chapter and in Chapter 25, we show how to solve such problems efficiently. In a ***shortest-paths problem***, we are given a weighted, directed graph $G = (V, E)$, with weight function $w : E \to \mathbf{R}$ mapping edges to real-valued weights. The ***weight*** of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

We define the ***shortest-path weight*** from $u$ to $v$ by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from } u \text{ to } v , \\ \infty & \text{otherwise .} \end{cases}$$

A ***shortest path*** from vertex $u$ to vertex $v$ is then defined as any path $p$ with weight $w(p) = \delta(u, v)$.

In the Chicago-to-Boston example, we can model the road map as a graph: vertices represent intersections, edges represent road segments between intersections, and edge weights represent road distances. Our goal is to find a shortest path from a given intersection in Chicago (say, Clark St. and Addison Ave.) to a given intersection in Boston (say, Brookline Ave. and Yawkey Way).

Edge weights can be interpreted as metrics other than distances. They are often used to represent time, cost, penalties, loss, or any other quantity that accumulates linearly along a path and that one wishes to minimize.

The breadth-first-search algorithm from Section 22.2 is a shortest-paths algorithm that works on unweighted graphs, that is, graphs in which each edge can be considered to have unit weight. Because many of the concepts from breadth-first search arise in the study of shortest paths in weighted graphs, the reader is encouraged to review Section 22.2 before proceeding.

### Variants

In this chapter, we shall focus on the ***single-source shortest-paths problem***: given a graph $G = (V, E)$, we want to find a shortest path from a given ***source*** vertex $s \in V$ to each vertex $v \in V$. Many other problems can be solved by the algorithm for the single-source problem, including the following variants.

**Single-destination shortest-paths problem:** Find a shortest path to a given ***destination*** vertex $t$ from each vertex $v$. By reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem.

**Single-pair shortest-path problem:** Find a shortest path from $u$ to $v$ for given vertices $u$ and $v$. If we solve the single-source problem with source vertex $u$, we solve this problem also. Moreover, no algorithms for this problem are known that run asymptotically faster than the best single-source algorithms in the worst case.

**All-pairs shortest-paths problem:** Find a shortest path from $u$ to $v$ for every pair of vertices $u$ and $v$. Although this problem can be solved by running a single-source algorithm once from each vertex, it can usually be solved faster. Additionally, its structure is of interest in its own right. Chapter 25 addresses the all-pairs problem in detail.

### Optimal substructure of a shortest path

Shortest-paths algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it. (The Edmonds-Karp maximum-flow algorithm in Chapter 26 also relies on this property.) This optimal-substructure property is a hallmark of the applicability of both dynamic programming (Chapter 15) and the greedy method (Chapter 16). Dijkstra's algorithm, which we shall see in Section 24.3, is a greedy algorithm, and the Floyd-Warshall algorithm, which finds shortest paths between all pairs of vertices (see Chapter 25), is a dynamic-programming algorithm. The following lemma states the optimal-substructure property of shortest paths more precisely.

***Lemma 24.1 (Subpaths of shortest paths are shortest paths)***
Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbf{R}$, let $p = \langle v_1, v_2, \ldots, v_k \rangle$ be a shortest path from vertex $v_1$ to vertex $v_k$ and, for any $i$ and $j$ such that $1 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of $p$ from vertex $v_i$ to vertex $v_j$. Then, $p_{ij}$ is a shortest path from $v_i$ to $v_j$.

***Proof*** If we decompose path $p$ into $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$, then we have that $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$. Now, assume that there is a path $p'_{ij}$ from $v_i$ to $v_j$ with weight $w(p'_{ij}) < w(p_{ij})$. Then, $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ is a path from $v_1$ to $v_k$ whose weight $w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$, which contradicts the assumption that $p$ is a shortest path from $v_1$ to $v_k$. ■

### Negative-weight edges

In some instances of the single-source shortest-paths problem, there may be edges whose weights are negative. If the graph $G = (V, E)$ contains no negative-weight cycles reachable from the source $s$, then for all $v \in V$, the shortest-path weight $\delta(s, v)$ remains well defined, even if it has a negative value. If there is a negative-weight cycle reachable from $s$, however, shortest-path weights are not well defined. No path from $s$ to a vertex on the cycle can be a shortest path—a lesser-weight path can always be found that follows the proposed "shortest" path and then traverses the negative-weight cycle. If there is a negative-weight cycle on some path from $s$ to $v$, we define $\delta(s, v) = -\infty$.

Figure 24.1 illustrates the effect of negative weights and negative-weight cycles on shortest-path weights. Because there is only one path from $s$ to $a$ (the path $\langle s, a \rangle$), $\delta(s, a) = w(s, a) = 3$. Similarly, there is only one path from $s$ to $b$, and so $\delta(s, b) = w(s, a) + w(a, b) = 3 + (-4) = -1$. There are infinitely many paths from $s$ to $c$: $\langle s, c \rangle$, $\langle s, c, d, c \rangle$, $\langle s, c, d, c, d, c \rangle$, and so on. Because the cycle $\langle c, d, c \rangle$ has weight $6 + (-3) = 3 > 0$, the shortest path from $s$ to $c$ is $\langle s, c \rangle$, with weight $\delta(s, c) = 5$. Similarly, the shortest path from $s$ to $d$ is $\langle s, c, d \rangle$, with weight $\delta(s, d) = w(s, c) + w(c, d) = 11$. Analogously, there are infinitely many paths from $s$ to $e$: $\langle s, e \rangle$, $\langle s, e, f, e \rangle$, $\langle s, e, f, e, f, e \rangle$, and so on. Since the cycle $\langle e, f, e \rangle$ has weight $3 + (-6) = -3 < 0$, however, there is no shortest path from $s$ to $e$. By traversing the negative-weight cycle $\langle e, f, e \rangle$ arbitrarily many times, we can find paths from $s$ to $e$ with arbitrarily large negative weights, and so $\delta(s, e) = -\infty$. Similarly, $\delta(s, f) = -\infty$. Because $g$ is reachable from $f$, we can also find paths with arbitrarily large negative weights from $s$ to $g$, and $\delta(s, g) = -\infty$. Vertices $h$, $i$, and $j$ also form a negative-weight cycle. They are not reachable from $s$, however, and so $\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$.

Some shortest-paths algorithms, such as Dijkstra's algorithm, assume that all edge weights in the input graph are nonnegative, as in the road-map example. Oth-

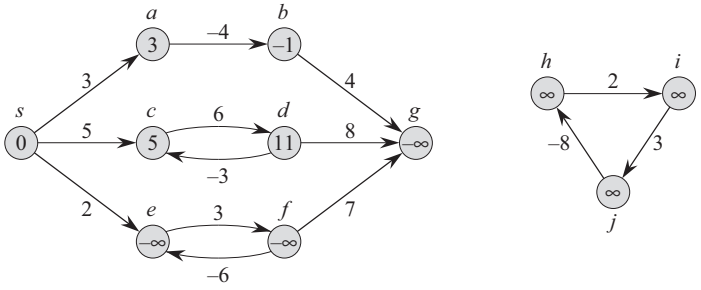



**Figure 24.1** Negative edge weights in a directed graph. Shown within each vertex is its shortest-path weight from source $s$. Because vertices $e$ and $f$ form a negative-weight cycle reachable from $s$, they have shortest-path weights of $-\infty$. Because vertex $g$ is reachable from a vertex whose shortest-path weight is $-\infty$, it, too, has a shortest-path weight of $-\infty$. Vertices such as $h$, $i$, and $j$ are not reachable from $s$, and so their shortest-path weights are $\infty$, even though they lie on a negative-weight cycle.

ers, such as the Bellman-Ford algorithm, allow negative-weight edges in the input graph and produce a correct answer as long as no negative-weight cycles are reachable from the source. Typically, if there is such a negative-weight cycle, the algorithm can detect and report its existence.

### Cycles

Can a shortest path contain a cycle? As we have just seen, it cannot contain a negative-weight cycle. Nor can it contain a positive-weight cycle, since removing the cycle from the path produces a path with the same source and destination vertices and a lower path weight. That is, if $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a path and $c = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ is a positive-weight cycle on this path (so that $v_i = v_j$ and $w(c) > 0$), then the path $p' = \langle v_0, v_1, \ldots, v_i, v_{j+1}, v_{j+2}, \ldots, v_k \rangle$ has weight $w(p') = w(p) - w(c) < w(p)$, and so $p$ cannot be a shortest path from $v_0$ to $v_k$.

That leaves only 0-weight cycles. We can remove a 0-weight cycle from any path to produce another path whose weight is the same. Thus, if there is a shortest path from a source vertex $s$ to a destination vertex $v$ that contains a 0-weight cycle, then there is another shortest path from $s$ to $v$ without this cycle. As long as a shortest path has 0-weight cycles, we can repeatedly remove these cycles from the path until we have a shortest path that is cycle-free. Therefore, without loss of generality we can assume that when we are finding shortest paths, they have no cycles. Since any acyclic path in a graph $G = (V, E)$ contains at most $|V|$ distinct vertices, it also contains at most $|V| - 1$ edges. Thus, we can restrict our attention to shortest paths of at most $|V| - 1$ edges.

### Representing shortest paths

We often wish to compute not only shortest-path weights, but the vertices on shortest paths as well. The representation we use for shortest paths is similar to the one we used for breadth-first trees in Section 22.2. Given a graph $G = (V, E)$, we maintain for each vertex $v \in V$ a ***predecessor*** $\pi[v]$ that is either another vertex or NIL. The shortest-paths algorithms in this chapter set the $\pi$ attributes so that the chain of predecessors originating at a vertex $v$ runs backwards along a shortest path from $s$ to $v$. Thus, given a vertex $v$ for which $\pi[v] \neq \text{NIL}$, the procedure PRINT-PATH$(G, s, v)$ from Section 22.2 can be used to print a shortest path from $s$ to $v$.

During the execution of a shortest-paths algorithm, however, the $\pi$ values need not indicate shortest paths. As in breadth-first search, we shall be interested in the ***predecessor subgraph*** $G_\pi = (V_\pi, E_\pi)$ induced by the $\pi$ values. Here again, we define the vertex set $V_\pi$ to be the set of vertices of $G$ with non-NIL predecessors, plus the source $s$:

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

The directed edge set $E_\pi$ is the set of edges induced by the $\pi$ values for vertices in $V_\pi$:

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

We shall prove that the $\pi$ values produced by the algorithms in this chapter have the property that at termination $G_\pi$ is a "shortest-paths tree"—informally, a rooted tree containing a shortest path from the source $s$ to every vertex that is reachable from $s$. A shortest-paths tree is like the breadth-first tree from Section 22.2, but it contains shortest paths from the source defined in terms of edge weights instead of numbers of edges. To be precise, let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbf{R}$, and assume that $G$ contains no negative-weight cycles reachable from the source vertex $s \in V$, so that shortest paths are well defined. A ***shortest-paths tree*** rooted at $s$ is a directed subgraph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$, such that

1. $V'$ is the set of vertices reachable from $s$ in $G$,
2. $G'$ forms a rooted tree with root $s$, and
3. for all $v \in V'$, the unique simple path from $s$ to $v$ in $G'$ is a shortest path from $s$ to $v$ in $G$.

Shortest paths are not necessarily unique, and neither are shortest-paths trees. For example, Figure 24.2 shows a weighted, directed graph and two shortest-paths trees with the same root.

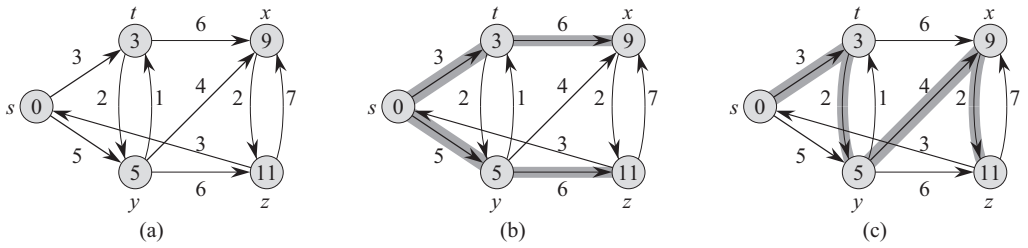



**Figure 24.2** **(a)** A weighted, directed graph with shortest-path weights from source $s$. **(b)** The shaded edges form a shortest-paths tree rooted at the source $s$. **(c)** Another shortest-paths tree with the same root.

## Relaxation

The algorithms in this chapter use the technique of ***relaxation***. For each vertex $v \in V$, we maintain an attribute $d[v]$, which is an upper bound on the weight of a shortest path from source $s$ to $v$. We call $d[v]$ a ***shortest-path estimate***. We initialize the shortest-path estimates and predecessors by the following $\Theta(V)$-time procedure.

INITIALIZE-SINGLE-SOURCE$(G, s)$
1 **for** each vertex $v \in V[G]$
2 **do** $d[v] \leftarrow \infty$
3 $\pi[v] \leftarrow$ NIL
4 $d[s] \leftarrow 0$

After initialization, $\pi[v] =$ NIL for all $v \in V$, $d[s] = 0$, and $d[v] = \infty$ for $v \in V - \{s\}$.

The process of ***relaxing***[1] an edge $(u, v)$ consists of testing whether we can improve the shortest path to $v$ found so far by going through $u$ and, if so, updating $d[v]$ and $\pi[v]$. A relaxation step may decrease the value of the shortest-path estimate $d[v]$ and update $v$'s predecessor field $\pi[v]$. The following code performs a relaxation step on edge $(u, v)$.

[1]It may seem strange that the term "relaxation" is used for an operation that tightens an upper bound. The use of the term is historical. The outcome of a relaxation step can be viewed as a relaxation of the constraint $d[v] \le d[u] + w(u, v)$, which, by the triangle inequality (Lemma 24.10), must be satisfied if $d[u] = \delta(s, u)$ and $d[v] = \delta(s, v)$. That is, if $d[v] \le d[u] + w(u, v)$, there is no "pressure" to satisfy this constraint, so the constraint is "relaxed."

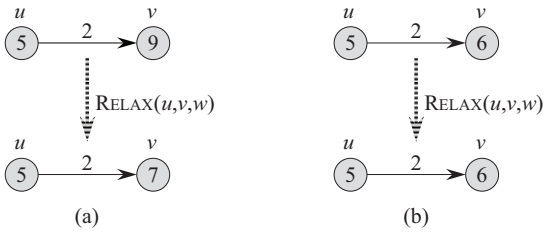



**Figure 24.3** Relaxation of an edge $(u, v)$ with weight $w(u, v) = 2$. The shortest-path estimate of each vertex is shown within the vertex. **(a)** Because $d[v] > d[u] + w(u, v)$ prior to relaxation, the value of $d[v]$ decreases. **(b)** Here, $d[v] \leq d[u] + w(u, v)$ before the relaxation step, and so $d[v]$ is unchanged by relaxation.

```
RELAX(u, v, w)
1  if d[v] > d[u] + w(u, v)
2     then d[v] ← d[u] + w(u, v)
3          π[v] ← u
```

Figure 24.3 shows two examples of relaxing an edge, one in which a shortest-path estimate decreases and one in which no estimate changes.

Each algorithm in this chapter calls INITIALIZE-SINGLE-SOURCE and then repeatedly relaxes edges. Moreover, relaxation is the only means by which shortest-path estimates and predecessors change. The algorithms in this chapter differ in how many times they relax each edge and the order in which they relax edges. In Dijkstra's algorithm and the shortest-paths algorithm for directed acyclic graphs, each edge is relaxed exactly once. In the Bellman-Ford algorithm, each edge is relaxed many times.

### Properties of shortest paths and relaxation

To prove the algorithms in this chapter correct, we shall appeal to several properties of shortest paths and relaxation. We state these properties here, and Section 24.5 proves them formally. For your reference, each property stated here includes the appropriate lemma or corollary number from Section 24.5. The latter five of these properties, which refer to shortest-path estimates or the predecessor subgraph, implicitly assume that the graph is initialized with a call to INITIALIZE-SINGLE-SOURCE$(G, s)$ and that the only way that shortest-path estimates and the predecessor subgraph change are by some sequence of relaxation steps.

**Triangle inequality** (Lemma 24.10)
For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

**Upper-bound property** (Lemma 24.11)
We always have $d[v] \geq \delta(s, v)$ for all vertices $v \in V$, and once $d[v]$ achieves the value $\delta(s, v)$, it never changes.

**No-path property** (Corollary 24.12)
If there is no path from $s$ to $v$, then we always have $d[v] = \delta(s, v) = \infty$.

**Convergence property** (Lemma 24.14)
If $s \rightsquigarrow u \rightarrow v$ is a shortest path in $G$ for some $u, v \in V$, and if $d[u] = \delta(s, u)$ at any time prior to relaxing edge $(u, v)$, then $d[v] = \delta(s, v)$ at all times afterward.

**Path-relaxation property** (Lemma 24.15)
If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $s = v_0$ to $v_k$, and the edges of $p$ are relaxed in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $d[v_k] = \delta(s, v_k)$. This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of $p$.

**Predecessor-subgraph property** (Lemma 24.17)
Once $d[v] = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at $s$.

## Chapter outline

Section 24.1 presents the Bellman-Ford algorithm, which solves the single-source shortest-paths problem in the general case in which edges can have negative weight. The Bellman-Ford algorithm is remarkable in its simplicity, and it has the further benefit of detecting whether a negative-weight cycle is reachable from the source. Section 24.2 gives a linear-time algorithm for computing shortest paths from a single source in a directed acyclic graph. Section 24.3 covers Dijkstra's algorithm, which has a lower running time than the Bellman-Ford algorithm but requires the edge weights to be nonnegative. Section 24.4 shows how the Bellman-Ford algorithm can be used to solve a special case of "linear programming." Finally, Section 24.5 proves the properties of shortest paths and relaxation stated above.

We require some conventions for doing arithmetic with infinities. We shall assume that for any real number $a \neq -\infty$, we have $a + \infty = \infty + a = \infty$. Also, to make our proofs hold in the presence of negative-weight cycles, we shall assume that for any real number $a \neq \infty$, we have $a + (-\infty) = (-\infty) + a = -\infty$.

All algorithms in this chapter assume that the directed graph $G$ is stored in the adjacency-list representation. Additionally, stored with each edge is its weight, so that as we traverse each adjacency list, we can determine the edge weights in $O(1)$ time per edge.

## 24.1 The Bellman-Ford algorithm

The ***Bellman-Ford algorithm*** solves the single-source shortest-paths problem in the general case in which edge weights may be negative. Given a weighted, directed graph $G = (V, E)$ with source $s$ and weight function $w : E \to \mathbf{R}$, the Bellman-Ford algorithm returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.

The algorithm uses relaxation, progressively decreasing an estimate $d[v]$ on the weight of a shortest path from the source $s$ to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$. The algorithm returns TRUE if and only if the graph contains no negative-weight cycles that are reachable from the source.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i ← 1 to |V[G]| − 1
3       do for each edge (u, v) ∈ E[G]
4            do RELAX(u, v, w)
5  for each edge (u, v) ∈ E[G]
6       do if d[v] > d[u] + w(u, v)
7            then return FALSE
8  return TRUE
```

Figure 24.4 shows the execution of the Bellman-Ford algorithm on a graph with 5 vertices. After initializing the $d$ and $\pi$ values of all vertices in line 1, the algorithm makes $|V| - 1$ passes over the edges of the graph. Each pass is one iteration of the **for** loop of lines 2–4 and consists of relaxing each edge of the graph once. Figures 24.4(b)–(e) show the state of the algorithm after each of the four passes over the edges. After making $|V| - 1$ passes, lines 5–8 check for a negative-weight cycle and return the appropriate boolean value. (We'll see a little later why this check works.)

The Bellman-Ford algorithm runs in time $O(VE)$, since the initialization in line 1 takes $\Theta(V)$ time, each of the $|V| - 1$ passes over the edges in lines 2–4 takes $\Theta(E)$ time, and the **for** loop of lines 5–7 takes $O(E)$ time.

To prove the correctness of the Bellman-Ford algorithm, we start by showing that if there are no negative-weight cycles, the algorithm computes correct shortest-path weights for all vertices reachable from the source.

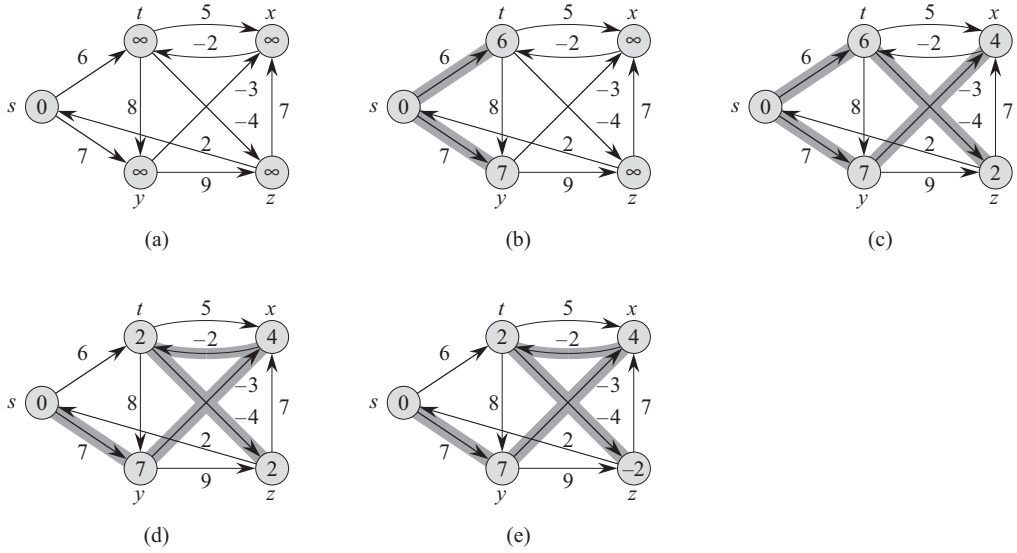



**Figure 24.4** The execution of the Bellman-Ford algorithm. The source is vertex $s$. The $d$ values are shown within the vertices, and shaded edges indicate predecessor values: if edge $(u, v)$ is shaded, then $\pi[v] = u$. In this particular example, each pass relaxes the edges in the order $(t, x)$, $(t, y)$, $(t, z)$, $(x, t)$, $(y, x)$, $(y, z)$, $(z, x)$, $(z, s)$, $(s, t)$, $(s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. The $d$ and $\pi$ values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

***Lemma 24.2***

Let $G = (V, E)$ be a weighted, directed graph with source $s$ and weight function $w : E \to \mathbf{R}$, and assume that $G$ contains no negative-weight cycles that are reachable from $s$. Then, after the $|V| - 1$ iterations of the **for** loop of lines 2–4 of BELLMAN-FORD, we have $d[v] = \delta(s, v)$ for all vertices $v$ that are reachable from $s$.

***Proof*** We prove the lemma by appealing to the path-relaxation property. Consider any vertex $v$ that is reachable from $s$, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any acyclic shortest path from $s$ to $v$. Path $p$ has at most $|V| - 1$ edges, and so $k \leq |V| - 1$. Each of the $|V| - 1$ iterations of the **for** loop of lines 2–4 relaxes all $E$ edges. Among the edges relaxed in the $i$th iteration, for $i = 1, 2, \ldots, k$, is $(v_{i-1}, v_i)$. By the path-relaxation property, therefore, $d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v)$. ■

***Corollary 24.3***
Let $G = (V, E)$ be a weighted, directed graph with source vertex $s$ and weight function $w : E \to \mathbf{R}$. Then for each vertex $v \in V$, there is a path from $s$ to $v$ if and only if BELLMAN-FORD terminates with $d[v] < \infty$ when it is run on $G$.

***Proof*** The proof is left as Exercise 24.1-2. ■

***Theorem 24.4 (Correctness of the Bellman-Ford algorithm)***
Let BELLMAN-FORD be run on a weighted, directed graph $G = (V, E)$ with source $s$ and weight function $w : E \to \mathbf{R}$. If $G$ contains no negative-weight cycles that are reachable from $s$, then the algorithm returns TRUE, we have $d[v] = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph $G_\pi$ is a shortest-paths tree rooted at $s$. If $G$ does contain a negative-weight cycle reachable from $s$, then the algorithm returns FALSE.

***Proof*** Suppose that graph $G$ contains no negative-weight cycles that are reachable from the source $s$. We first prove the claim that at termination, $d[v] = \delta(s, v)$ for all vertices $v \in V$. If vertex $v$ is reachable from $s$, then Lemma 24.2 proves this claim. If $v$ is not reachable from $s$, then the claim follows from the no-path property. Thus, the claim is proven. The predecessor-subgraph property, along with the claim, implies that $G_\pi$ is a shortest-paths tree. Now we use the claim to show that BELLMAN-FORD returns TRUE. At termination, we have for all edges $(u, v) \in E$,

$$\begin{aligned} d[v] &= \delta(s, v) \\ &\le \delta(s, u) + w(u, v) \quad \text{(by the triangle inequality)} \\ &= d[u] + w(u, v) , \end{aligned}$$

and so none of the tests in line 6 causes BELLMAN-FORD to return FALSE. It therefore returns TRUE.

Conversely, suppose that graph $G$ contains a negative-weight cycle that is reachable from the source $s$; let this cycle be $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$. Then,

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 . \tag{24.1}$$

Assume for the purpose of contradiction that the Bellman-Ford algorithm returns TRUE. Thus, $d[v_i] \le d[v_{i-1}] + w(v_{i-1}, v_i)$ for $i = 1, 2, \ldots, k$. Summing the inequalities around cycle $c$ gives us

$$\begin{aligned}\sum_{i=1}^{k} d[v_i] &\leq \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i) \ .\end{aligned}$$

Since $v_0 = v_k$, each vertex in $c$ appears exactly once in each of the summations $\sum_{i=1}^{k} d[v_i]$ and $\sum_{i=1}^{k} d[v_{i-1}]$, and so

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}] \ .$$

Moreover, by Corollary 24.3, $d[v_i]$ is finite for $i = 1, 2, \ldots, k$. Thus,

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i) \ ,$$

which contradicts inequality (24.1). We conclude that the Bellman-Ford algorithm returns TRUE if graph $G$ contains no negative-weight cycles reachable from the source, and FALSE otherwise. ■

## Exercises

### *24.1-1*

Run the Bellman-Ford algorithm on the directed graph of Figure 24.4, using vertex $z$ as the source. In each pass, relax edges in the same order as in the figure, and show the $d$ and $\pi$ values after each pass. Now, change the weight of edge $(z, x)$ to 4 and run the algorithm again, using $s$ as the source.

### *24.1-2*

Prove Corollary 24.3.

### *24.1-3*

Given a weighted, directed graph $G = (V, E)$ with no negative-weight cycles, let $m$ be the maximum over all pairs of vertices $u, v \in V$ of the minimum number of edges in a shortest path from $u$ to $v$. (Here, the shortest path is by weight, not the number of edges.) Suggest a simple change to the Bellman-Ford algorithm that allows it to terminate in $m + 1$ passes.

### *24.1-4*

Modify the Bellman-Ford algorithm so that it sets $d[v]$ to $-\infty$ for all vertices $v$ for which there is a negative-weight cycle on some path from the source to $v$.

***24.1-5*** ⋆

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbf{R}$. Give an $O(VE)$-time algorithm to find, for each vertex $v \in V$, the value $\delta^*(v) = \min_{u \in V} \{\delta(u, v)\}$.

***24.1-6*** ⋆

Suppose that a weighted, directed graph $G = (V, E)$ has a negative-weight cycle. Give an efficient algorithm to list the vertices of one such cycle. Prove that your algorithm is correct.

## 24.2 Single-source shortest paths in directed acyclic graphs

By relaxing the edges of a weighted dag (directed acyclic graph) $G = (V, E)$ according to a topological sort of its vertices, we can compute shortest paths from a single source in $\Theta(V + E)$ time. Shortest paths are always well defined in a dag, since even if there are negative-weight edges, no negative-weight cycles can exist.

The algorithm starts by topologically sorting the dag (see Section 22.4) to impose a linear ordering on the vertices. If there is a path from vertex $u$ to vertex $v$, then $u$ precedes $v$ in the topological sort. We make just one pass over the vertices in the topologically sorted order. As we process each vertex, we relax each edge that leaves the vertex.

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4       do for each vertex v ∈ Adj[u]
5            do RELAX(u, v, w)
```

Figure 24.5 shows the execution of this algorithm.

The running time of this algorithm is easy to analyze. As shown in Section 22.4, the topological sort of line 1 can be performed in $\Theta(V + E)$ time. The call of INITIALIZE-SINGLE-SOURCE in line 2 takes $\Theta(V)$ time. There is one iteration per vertex in the **for** loop of lines 3–5. For each vertex, the edges that leave the vertex are each examined exactly once. Thus, there are a total of $|E|$ iterations of the inner **for** loop of lines 4–5. (We have used an aggregate analysis here.) Because each iteration of the inner **for** loop takes $\Theta(1)$ time, the total running time is $\Theta(V + E)$, which is linear in the size of an adjacency-list representation of the graph.

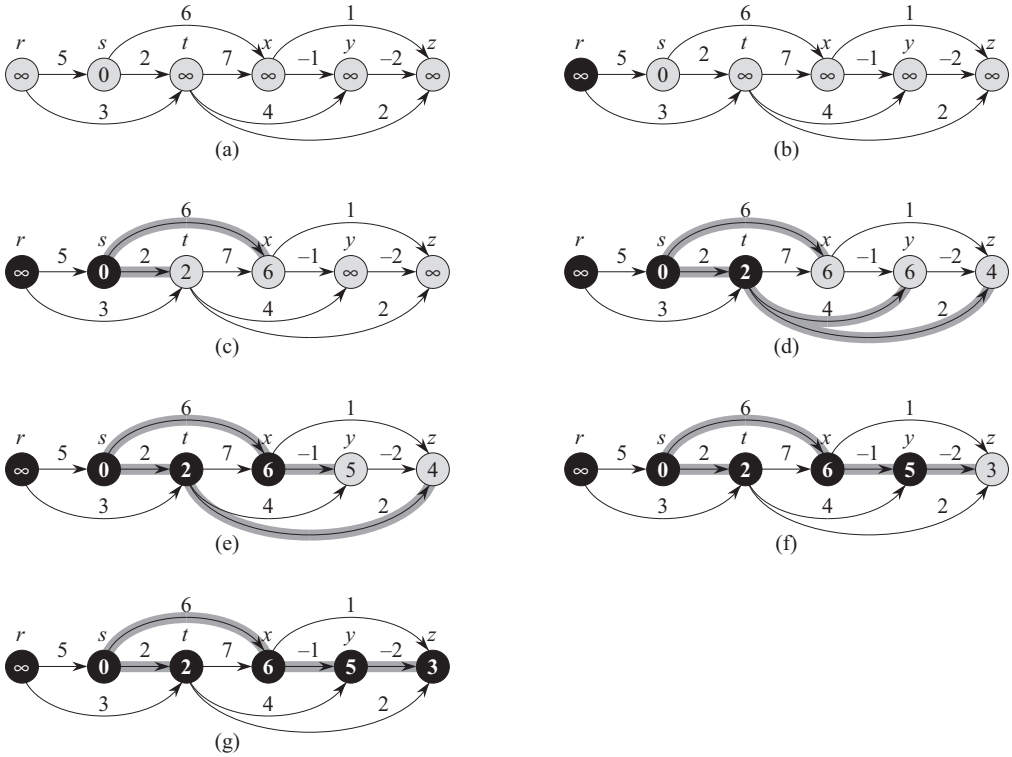



**Figure 24.5** The execution of the algorithm for shortest paths in a directed acyclic graph. The vertices are topologically sorted from left to right. The source vertex is $s$. The $d$ values are shown within the vertices, and shaded edges indicate the $\pi$ values. **(a)** The situation before the first iteration of the **for** loop of lines 3–5. **(b)–(g)** The situation after each iteration of the **for** loop of lines 3–5. The newly blackened vertex in each iteration was used as $u$ in that iteration. The values shown in part (g) are the final values.

The following theorem shows that the DAG-SHORTEST-PATHS procedure correctly computes the shortest paths.

***Theorem 24.5***

If a weighted, directed graph $G = (V, E)$ has source vertex $s$ and no cycles, then at the termination of the DAG-SHORTEST-PATHS procedure, $d[v] = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph $G_\pi$ is a shortest-paths tree.

***Proof*** We first show that $d[v] = \delta(s, v)$ for all vertices $v \in V$ at termination. If $v$ is not reachable from $s$, then $d[v] = \delta(s, v) = \infty$ by the no-path property. Now, suppose that $v$ is reachable from $s$, so that there is a shortest path $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = v$. Because we process the vertices in topologically sorted order, the edges on $p$ are relaxed in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. The path-relaxation property implies that $d[v_i] = \delta(s, v_i)$ at termination for $i = 0, 1, \ldots, k$. Finally, by the predecessor-subgraph property, $G_\pi$ is a shortest-paths tree. ■

An interesting application of this algorithm arises in determining critical paths in ***PERT chart***[2] analysis. Edges represent jobs to be performed, and edge weights represent the times required to perform particular jobs. If edge $(u, v)$ enters vertex $v$ and edge $(v, x)$ leaves $v$, then job $(u, v)$ must be performed prior to job $(v, x)$. A path through this dag represents a sequence of jobs that must be performed in a particular order. A ***critical path*** is a *longest* path through the dag, corresponding to the longest time to perform an ordered sequence of jobs. The weight of a critical path is a lower bound on the total time to perform all the jobs. We can find a critical path by either

- negating the edge weights and running DAG-SHORTEST-PATHS, or
- running DAG-SHORTEST-PATHS, with the modification that we replace "$\infty$" by "$-\infty$" in line 2 of INITIALIZE-SINGLE-SOURCE and "$>$" by "$<$" in the RELAX procedure.

## Exercises

***24.2-1***
Run DAG-SHORTEST-PATHS on the directed graph of Figure 24.5, using vertex $r$ as the source.

***24.2-2***
Suppose we change line 3 of DAG-SHORTEST-PATHS to read

3 **for** the first $|V| - 1$ vertices, taken in topologically sorted order

Show that the procedure would remain correct.

***24.2-3***
The PERT chart formulation given above is somewhat unnatural. It would be more natural for vertices to represent jobs and edges to represent sequencing con-

[2] "PERT" is an acronym for "program evaluation and review technique."

straints; that is, edge $(u, v)$ would indicate that job $u$ must be performed before job $v$. Weights would then be assigned to vertices, not edges. Modify the DAG-SHORTEST-PATHS procedure so that it finds a longest path in a directed acyclic graph with weighted vertices in linear time.

***24.2-4***
Give an efficient algorithm to count the total number of paths in a directed acyclic graph. Analyze your algorithm.

## 24.3 Dijkstra's algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative. In this section, therefore, we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$. As we shall see, with a good implementation, the running time of Dijkstra's algorithm is lower than that of the Bellman-Ford algorithm.

Dijkstra's algorithm maintains a set $S$ of vertices whose final shortest-path weights from the source $s$ have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds $u$ to $S$, and relaxes all edges leaving $u$. In the following implementation, we use a min-priority queue $Q$ of vertices, keyed by their $d$ values.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S ← ∅
3  Q ← V[G]
4  while Q ≠ ∅
5        do u ← EXTRACT-MIN(Q)
6           S ← S ∪ {u}
7           for each vertex v ∈ Adj[u]
8                 do RELAX(u, v, w)
```

Dijkstra's algorithm relaxes edges as shown in Figure 24.6. Line 1 performs the usual initialization of $d$ and $\pi$ values, and line 2 initializes the set $S$ to the empty set. The algorithm maintains the invariant that $Q = V - S$ at the start of each iteration of the **while** loop of lines 4–8. Line 3 initializes the min-priority queue $Q$ to contain all the vertices in $V$; since $S = \emptyset$ at that time, the invariant is true after line 3. Each time through the **while** loop of lines 4–8, a vertex $u$ is extracted from $Q = V - S$ and added to set $S$, thereby maintaining the invariant. (The first time through this loop, $u = s$.) Vertex $u$, therefore, has the smallest shortest-path

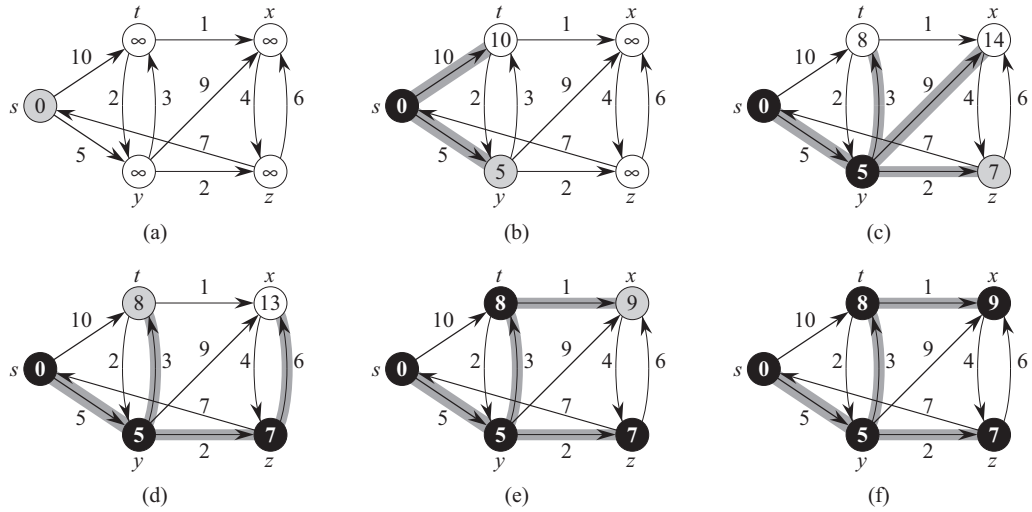



**Figure 24.6** The execution of Dijkstra's algorithm. The source $s$ is the leftmost vertex. The shortest-path estimates are shown within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set $S$, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 4–8. The shaded vertex has the minimum $d$ value and is chosen as vertex $u$ in line 5. **(b)–(f)** The situation after each successive iteration of the **while** loop. The shaded vertex in each part is chosen as vertex $u$ in line 5 of the next iteration. The $d$ and $\pi$ values shown in part (f) are the final values.

estimate of any vertex in $V - S$. Then, lines 7–8 relax each edge $(u, v)$ leaving $u$, thus updating the estimate $d[v]$ and the predecessor $\pi[v]$ if the shortest path to $v$ can be improved by going through $u$. Observe that vertices are never inserted into $Q$ after line 3 and that each vertex is extracted from $Q$ and added to $S$ exactly once, so that the **while** loop of lines 4–8 iterates exactly $|V|$ times.

Because Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in $V - S$ to add to set $S$, we say that it uses a greedy strategy. Greedy strategies are presented in detail in Chapter 16, but you need not have read that chapter to understand Dijkstra's algorithm. Greedy strategies do not always yield optimal results in general, but as the following theorem and its corollary show, Dijkstra's algorithm does indeed compute shortest paths. The key is to show that each time a vertex $u$ is added to set $S$, we have $d[u] = \delta(s, u)$.

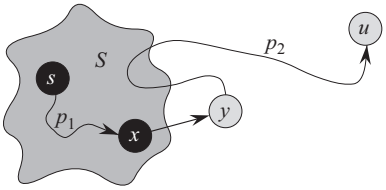



**Figure 24.7** The proof of Theorem 24.6. Set $S$ is nonempty just before vertex $u$ is added to it. A shortest path $p$ from source $s$ to vertex $u$ can be decomposed into $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$, where $y$ is the first vertex on the path that is not in $S$ and $x \in S$ immediately precedes $y$. Vertices $x$ and $y$ are distinct, but we may have $s = x$ or $y = u$. Path $p_2$ may or may not reenter set $S$.

***Theorem 24.6 (Correctness of Dijkstra's algorithm)***
Dijkstra's algorithm, run on a weighted, directed graph $G = (V, E)$ with nonnegative weight function $w$ and source $s$, terminates with $d[u] = \delta(s, u)$ for all vertices $u \in V$.

***Proof*** We use the following loop invariant:

> At the start of each iteration of the **while** loop of lines 4–8, $d[v] = \delta(s, v)$ for each vertex $v \in S$.

It suffices to show for each vertex $u \in V$, we have $d[u] = \delta(s, u)$ at the time when $u$ is added to set $S$. Once we show that $d[u] = \delta(s, u)$, we rely on the upper-bound property to show that the equality holds at all times thereafter.

**Initialization:** Initially, $S = \emptyset$, and so the invariant is trivially true.

**Maintenance:** We wish to show that in each iteration, $d[u] = \delta(s, u)$ for the vertex added to set $S$. For the purpose of contradiction, let $u$ be the first vertex for which $d[u] \neq \delta(s, u)$ when it is added to set $S$. We shall focus our attention on the situation at the beginning of the iteration of the **while** loop in which $u$ is added to $S$ and derive the contradiction that $d[u] = \delta(s, u)$ at that time by examining a shortest path from $s$ to $u$. We must have $u \neq s$ because $s$ is the first vertex added to set $S$ and $d[s] = \delta(s, s) = 0$ at that time. Because $u \neq s$, we also have that $S \neq \emptyset$ just before $u$ is added to $S$. There must be some path from $s$ to $u$, for otherwise $d[u] = \delta(s, u) = \infty$ by the no-path property, which would violate our assumption that $d[u] \neq \delta(s, u)$. Because there is at least one path, there is a shortest path $p$ from $s$ to $u$. Prior to adding $u$ to $S$, path $p$ connects a vertex in $S$, namely $s$, to a vertex in $V - S$, namely $u$. Let us consider the first vertex $y$ along $p$ such that $y \in V - S$, and let $x \in S$ be $y$'s predecessor. Thus, as shown in Figure 24.7, path $p$ can be decomposed as $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$. (Either of paths $p_1$ or $p_2$ may have no edges.)

We claim that $d[y] = \delta(s, y)$ when $u$ is added to $S$. To prove this claim, observe that $x \in S$. Then, because $u$ is chosen as the first vertex for which $d[u] \neq \delta(s, u)$ when it is added to $S$, we had $d[x] = \delta(s, x)$ when $x$ was added to $S$. Edge $(x, y)$ was relaxed at that time, so the claim follows from the convergence property.

We can now obtain a contradiction to prove that $d[u] = \delta(s, u)$. Because $y$ occurs before $u$ on a shortest path from $s$ to $u$ and all edge weights are nonnegative (notably those on path $p_2$), we have $\delta(s, y) \leq \delta(s, u)$, and thus

$$\begin{aligned} d[y] &= \delta(s, y) \\ &\leq \delta(s, u) \\ &\leq d[u] \qquad \text{(by the upper-bound property)} \ . \end{aligned} \tag{24.2}$$

But because both vertices $u$ and $y$ were in $V - S$ when $u$ was chosen in line 5, we have $d[u] \leq d[y]$. Thus, the two inequalities in (24.2) are in fact equalities, giving

$$d[y] = \delta(s, y) = \delta(s, u) = d[u] \ .$$

Consequently, $d[u] = \delta(s, u)$, which contradicts our choice of $u$. We conclude that $d[u] = \delta(s, u)$ when $u$ is added to $S$, and that this equality is maintained at all times thereafter.

**Termination:** At termination, $Q = \emptyset$ which, along with our earlier invariant that $Q = V - S$, implies that $S = V$. Thus, $d[u] = \delta(s, u)$ for all vertices $u \in V$. ■

***Corollary 24.7***

If we run Dijkstra's algorithm on a weighted, directed graph $G = (V, E)$ with nonnegative weight function $w$ and source $s$, then at termination, the predecessor subgraph $G_\pi$ is a shortest-paths tree rooted at $s$.

***Proof*** Immediate from Theorem 24.6 and the predecessor-subgraph property. ■

### Analysis

How fast is Dijkstra's algorithm? It maintains the min-priority queue $Q$ by calling three priority-queue operations: INSERT (implicit in line 3), EXTRACT-MIN (line 5), and DECREASE-KEY (implicit in RELAX, which is called in line 8). INSERT is invoked once per vertex, as is EXTRACT-MIN. Because each vertex $v \in V$ is added to set $S$ exactly once, each edge in the adjacency list $Adj[v]$ is examined in the **for** loop of lines 7–8 exactly once during the course of the algorithm. Since the total number of edges in all the adjacency lists is $|E|$, there are a total of $|E|$ iterations of this **for** loop, and thus a total of at most $|E|$ DECREASE-KEY operations. (Observe once again that we are using aggregate analysis.)

The running time of Dijkstra's algorithm depends on how the min-priority queue is implemented. Consider first the case in which we maintain the min-priority queue by taking advantage of the vertices being numbered 1 to $|V|$. We simply store $d[v]$ in the $v$th entry of an array. Each INSERT and DECREASE-KEY operation takes $O(1)$ time, and each EXTRACT-MIN operation takes $O(V)$ time (since we have to search through the entire array), for a total time of $O(V^2+E) = O(V^2)$.

If the graph is sufficiently sparse—in particular, $E = o(V^2/\lg V)$—it is practical to implement the min-priority queue with a binary min-heap. (As discussed in Section 6.5, an important implementation detail is that vertices and corresponding heap elements must maintain handles to each other.) Each EXTRACT-MIN operation then takes time $O(\lg V)$. As before, there are $|V|$ such operations. The time to build the binary min-heap is $O(V)$. Each DECREASE-KEY operation takes time $O(\lg V)$, and there are still at most $|E|$ such operations. The total running time is therefore $O((V + E) \lg V)$, which is $O(E \lg V)$ if all vertices are reachable from the source. This running time is an improvement over the straightforward $O(V^2)$-time implementation if $E = o(V^2/\lg V)$.

We can in fact achieve a running time of $O(V \lg V + E)$ by implementing the min-priority queue with a Fibonacci heap (see Chapter 20). The amortized cost of each of the $|V|$ EXTRACT-MIN operations is $O(\lg V)$, and each DECREASE-KEY call, of which there are at most $|E|$, takes only $O(1)$ amortized time. Historically, the development of Fibonacci heaps was motivated by the observation that in Dijkstra's algorithm there are typically many more DECREASE-KEY calls than EXTRACT-MIN calls, so any method of reducing the amortized time of each DECREASE-KEY operation to $o(\lg V)$ without increasing the amortized time of EXTRACT-MIN would yield an asymptotically faster implementation than with binary heaps.

Dijkstra's algorithm bears some similarity to both breadth-first search (see Section 22.2) and Prim's algorithm for computing minimum spanning trees (see Section 23.2). It is like breadth-first search in that set $S$ corresponds to the set of black vertices in a breadth-first search; just as vertices in $S$ have their final shortest-path weights, so do black vertices in a breadth-first search have their correct breadth-first distances. Dijkstra's algorithm is like Prim's algorithm in that both algorithms use a min-priority queue to find the "lightest" vertex outside a given set (the set $S$ in Dijkstra's algorithm and the tree being grown in Prim's algorithm), add this vertex into the set, and adjust the weights of the remaining vertices outside the set accordingly.

## Exercises

***24.3-1***
Run Dijkstra's algorithm on the directed graph of Figure 24.2, first using vertex $s$ as the source and then using vertex $z$ as the source. In the style of Figure 24.6, show the $d$ and $\pi$ values and the vertices in set $S$ after each iteration of the **while** loop.

***24.3-2***
Give a simple example of a directed graph with negative-weight edges for which Dijkstra's algorithm produces incorrect answers. Why doesn't the proof of Theorem 24.6 go through when negative-weight edges are allowed?

***24.3-3***
Suppose we change line 4 of Dijkstra's algorithm to the following.

4 **while** $|Q| > 1$

This change causes the **while** loop to execute $|V| - 1$ times instead of $|V|$ times. Is this proposed algorithm correct?

***24.3-4***
We are given a directed graph $G = (V, E)$ on which each edge $(u, v) \in E$ has an associated value $r(u, v)$, which is a real number in the range $0 \leq r(u, v) \leq 1$ that represents the reliability of a communication channel from vertex $u$ to vertex $v$. We interpret $r(u, v)$ as the probability that the channel from $u$ to $v$ will not fail, and we assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.

***24.3-5***
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \{1, 2, \ldots, W\}$ for some positive integer $W$, and assume that no two vertices have the same shortest-path weights from source vertex $s$. Now suppose that we define an unweighted, directed graph $G' = (V \cup V', E')$ by replacing each edge $(u, v) \in E$ with $w(u, v)$ unit-weight edges in series. How many vertices does $G'$ have? Now suppose that we run a breadth-first search on $G'$. Show that the order in which vertices in $V$ are colored black in the breadth-first search of $G'$ is the same as the order in which the vertices of $V$ are extracted from the priority queue in line 5 of DIJKSTRA when run on $G$.

***24.3-6***
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \{0, 1, \ldots, W\}$ for some nonnegative integer $W$. Modify Dijkstra's algorithm to compute the shortest paths from a given source vertex $s$ in $O(WV + E)$ time.

***24.3-7***
Modify your algorithm from Exercise 24.3-6 to run in $O((V + E) \lg W)$ time. (*Hint:* How many distinct shortest-path estimates can there be in $V - S$ at any point in time?)

***24.3-8***
Suppose that we are given a weighted, directed graph $G = (V, E)$ in which edges that leave the source vertex $s$ may have negative weights, all other edge weights are nonnegative, and there are no negative-weight cycles. Argue that Dijkstra's algorithm correctly finds shortest paths from $s$ in this graph.

## 24.4 Difference constraints and shortest paths

Chapter 29 studies the general linear-programming problem, in which we wish to optimize a linear function subject to a set of linear inequalities. In this section, we investigate a special case of linear programming that can be reduced to finding shortest paths from a single source. The single-source shortest-paths problem that results can then be solved using the Bellman-Ford algorithm, thereby also solving the linear-programming problem.

### Linear programming

In the general ***linear-programming problem***, we are given an $m \times n$ matrix $A$, an $m$-vector $b$, and an $n$-vector $c$. We wish to find a vector $x$ of $n$ elements that maximizes the ***objective function*** $\sum_{i=1}^{n} c_i x_i$ subject to the $m$ constraints given by $Ax \leq b$.

Although the simplex algorithm, which is the focus of Chapter 29, does not always run in time polynomial in the size of its input, there are other linear-programming algorithms that do run in polynomial time. There are several reasons that it is important to understand the setup of linear-programming problems. First, knowing that a given problem can be cast as a polynomial-sized linear-programming problem immediately means that there is a polynomial-time algorithm for the problem. Second, there are many special cases of linear programming for which faster algorithms exist. For example, as shown in this section, the single-source shortest-paths problem is a special case of linear programming. Other problems that can be cast as linear programming include the single-pair shortest-path problem (Exercise 24.4-4) and the maximum-flow problem (Exercise 26.1-8).

Sometimes we don't really care about the objective function; we just wish to find any ***feasible solution***, that is, any vector $x$ that satisfies $Ax \leq b$, or to determine that no feasible solution exists. We shall focus on one such ***feasibility problem***.

### Systems of difference constraints

In a ***system of difference constraints***, each row of the linear-programming matrix $A$ contains one 1 and one $-1$, and all other entries of $A$ are 0. Thus, the constraints given by $Ax \le b$ are a set of $m$ ***difference constraints*** involving $n$ unknowns, in which each constraint is a simple linear inequality of the form

$$x_j - x_i \le b_k \,,$$

where $1 \le i, j \le n$ and $1 \le k \le m$.

For example, consider the problem of finding the 5-vector $x = (x_i)$ that satisfies

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \le \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix} .$$

This problem is equivalent to finding the unknowns $x_i$, for $i = 1, 2, \ldots, 5$, such that the following 8 difference constraints are satisfied:

$$x_1 - x_2 \le 0 \,, \tag{24.3}$$
$$x_1 - x_5 \le -1 \,, \tag{24.4}$$
$$x_2 - x_5 \le 1 \,, \tag{24.5}$$
$$x_3 - x_1 \le 5 \,, \tag{24.6}$$
$$x_4 - x_1 \le 4 \,, \tag{24.7}$$
$$x_4 - x_3 \le -1 \,, \tag{24.8}$$
$$x_5 - x_3 \le -3 \,, \tag{24.9}$$
$$x_5 - x_4 \le -3 \,. \tag{24.10}$$

One solution to this problem is $x = (-5, -3, 0, -1, -4)$, as can be verified directly by checking each inequality. In fact, there is more than one solution to this problem. Another is $x' = (0, 2, 5, 4, 1)$. These two solutions are related: each component of $x'$ is 5 larger than the corresponding component of $x$. This fact is not mere coincidence.

***Lemma 24.8***

Let $x = (x_1, x_2, \ldots, x_n)$ be a solution to a system $Ax \le b$ of difference constraints, and let $d$ be any constant. Then $x + d = (x_1 + d, x_2 + d, \ldots, x_n + d)$ is a solution to $Ax \le b$ as well.

***Proof*** For each $x_i$ and $x_j$, we have $(x_j + d) - (x_i + d) = x_j - x_i$. Thus, if $x$ satisfies $Ax \leq b$, so does $x + d$. ■

Systems of difference constraints occur in many different applications. For example, the unknowns $x_i$ may be times at which events are to occur. Each constraint can be viewed as stating that there must be at least a certain amount of time, or at most a certain amount of time, between two events. Perhaps the events are jobs to be performed during the assembly of a product. If we apply an adhesive that takes 2 hours to set at time $x_1$ and we have to wait until it sets to install a part at time $x_2$, then we have the constraint that $x_2 \geq x_1 + 2$ or, equivalently, that $x_1 - x_2 \leq -2$. Alternatively, we might require that the part be installed after the adhesive has been applied but no later than the time that the adhesive has set halfway. In this case, we get the pair of constraints $x_2 \geq x_1$ and $x_2 \leq x_1 + 1$ or, equivalently, $x_1 - x_2 \leq 0$ and $x_2 - x_1 \leq 1$.

### Constraint graphs

It is beneficial to interpret systems of difference constraints from a graph-theoretic point of view. The idea is that in a system $Ax \leq b$ of difference constraints, the $m \times n$ linear-programming matrix $A$ can be viewed as the transpose of an incidence matrix (see Exercise 22.1-7) for a graph with $n$ vertices and $m$ edges. Each vertex $v_i$ in the graph, for $i = 1, 2, \ldots, n$, corresponds to one of the $n$ unknown variables $x_i$. Each directed edge in the graph corresponds to one of the $m$ inequalities involving two unknowns.

More formally, given a system $Ax \leq b$ of difference constraints, the corresponding ***constraint graph*** is a weighted, directed graph $G = (V, E)$, where

$$V = \{v_0, v_1, \ldots, v_n\}$$

and

$$\begin{aligned} E = {} & \{(v_i, v_j) : x_j - x_i \leq b_k \text{ is a constraint}\} \\ & \cup \{(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots, (v_0, v_n)\} \ . \end{aligned}$$

The additional vertex $v_0$ is incorporated, as we shall see shortly, to guarantee that every other vertex is reachable from it. Thus, the vertex set $V$ consists of a vertex $v_i$ for each unknown $x_i$, plus an additional vertex $v_0$. The edge set $E$ contains an edge for each difference constraint, plus an edge $(v_0, v_i)$ for each unknown $x_i$. If $x_j - x_i \leq b_k$ is a difference constraint, then the weight of edge $(v_i, v_j)$ is $w(v_i, v_j) = b_k$. The weight of each edge leaving $v_0$ is 0. Figure 24.8 shows the constraint graph for the system (24.3)–(24.10) of difference constraints.

The following theorem shows that we can find a solution to a system of difference constraints by finding shortest-path weights in the corresponding constraint graph.

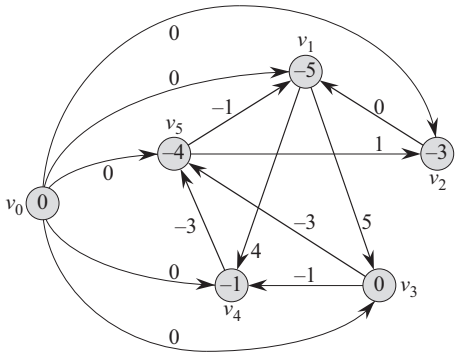



**Figure 24.8** The constraint graph corresponding to the system (24.3)–(24.10) of difference constraints. The value of $\delta(v_0, v_i)$ is shown in each vertex $v_i$. A feasible solution to the system is $x = (-5, -3, 0, -1, -4)$.

***Theorem 24.9***

Given a system $Ax \leq b$ of difference constraints, let $G = (V, E)$ be the corresponding constraint graph. If $G$ contains no negative-weight cycles, then

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \ldots, \delta(v_0, v_n)) \tag{24.11}$$

is a feasible solution for the system. If $G$ contains a negative-weight cycle, then there is no feasible solution for the system.

***Proof*** We first show that if the constraint graph contains no negative-weight cycles, then equation (24.11) gives a feasible solution. Consider any edge $(v_i, v_j) \in E$. By the triangle inequality, $\delta(v_0, v_j) \leq \delta(v_0, v_i) + w(v_i, v_j)$ or, equivalently, $\delta(v_0, v_j) - \delta(v_0, v_i) \leq w(v_i, v_j)$. Thus, letting $x_i = \delta(v_0, v_i)$ and $x_j = \delta(v_0, v_j)$ satisfies the difference constraint $x_j - x_i \leq w(v_i, v_j)$ that corresponds to edge $(v_i, v_j)$.

Now we show that if the constraint graph contains a negative-weight cycle, then the system of difference constraints has no feasible solution. Without loss of generality, let the negative-weight cycle be $c = \langle v_1, v_2, \ldots, v_k \rangle$, where $v_1 = v_k$. (The vertex $v_0$ cannot be on cycle $c$, because it has no entering edges.) Cycle $c$ corresponds to the following difference constraints:

$$\begin{aligned}
x_2 - x_1 &\leq w(v_1, v_2) , \\
x_3 - x_2 &\leq w(v_2, v_3) , \\
&\vdots \\
x_k - x_{k-1} &\leq w(v_{k-1}, v_k) , \\
x_1 - x_k &\leq w(v_k, v_1) .
\end{aligned}$$

Suppose that there is a solution for $x$ satisfying each of these $k$ inequalities. This solution must also satisfy the inequality that results when we sum the $k$ inequalities together. If we sum the left-hand sides, each unknown $x_i$ is added in once and subtracted out once, so that the left-hand side of the sum is 0. The right-hand side sums to $w(c)$, and thus we obtain $0 \leq w(c)$. But since $c$ is a negative-weight cycle, $w(c) < 0$, and we obtain the contradiction that $0 \leq w(c) < 0$. ■

### Solving systems of difference constraints

Theorem 24.9 tells us that we can use the Bellman-Ford algorithm to solve a system of difference constraints. Because there are edges from the source vertex $v_0$ to all other vertices in the constraint graph, any negative-weight cycle in the constraint graph is reachable from $v_0$. If the Bellman-Ford algorithm returns TRUE, then the shortest-path weights give a feasible solution to the system. In Figure 24.8, for example, the shortest-path weights provide the feasible solution $x = (-5, -3, 0, -1, -4)$, and by Lemma 24.8, $x = (d-5, d-3, d, d-1, d-4)$ is also a feasible solution for any constant $d$. If the Bellman-Ford algorithm returns FALSE, there is no feasible solution to the system of difference constraints.

A system of difference constraints with $m$ constraints on $n$ unknowns produces a graph with $n+1$ vertices and $n+m$ edges. Thus, using the Bellman-Ford algorithm, we can solve the system in $O((n+1)(n+m)) = O(n^2+nm)$ time. Exercise 24.4-5 asks you to modify the algorithm to run in $O(nm)$ time, even if $m$ is much less than $n$.

### Exercises

#### *24.4-1*

Find a feasible solution or determine that no feasible solution exists for the following system of difference constraints:

$$
\begin{aligned}
x_1 - x_2 &\leq 1, \\
x_1 - x_4 &\leq -4, \\
x_2 - x_3 &\leq 2, \\
x_2 - x_5 &\leq 7, \\
x_2 - x_6 &\leq 5, \\
x_3 - x_6 &\leq 10, \\
x_4 - x_2 &\leq 2, \\
x_5 - x_1 &\leq -1, \\
x_5 - x_4 &\leq 3, \\
x_6 - x_3 &\leq -8.
\end{aligned}
$$

***24.4-2***
Find a feasible solution or determine that no feasible solution exists for the following system of difference constraints:

$$\begin{aligned}
x_1 - x_2 &\le 4, \\
x_1 - x_5 &\le 5, \\
x_2 - x_4 &\le -6, \\
x_3 - x_2 &\le 1, \\
x_4 - x_1 &\le 3, \\
x_4 - x_3 &\le 5, \\
x_4 - x_5 &\le 10, \\
x_5 - x_3 &\le -4, \\
x_5 - x_4 &\le -8.
\end{aligned}$$

***24.4-3***
Can any shortest-path weight from the new vertex $v_0$ in a constraint graph be positive? Explain.

***24.4-4***
Express the single-pair shortest-path problem as a linear program.

***24.4-5***
Show how to modify the Bellman-Ford algorithm slightly so that when it is used to solve a system of difference constraints with $m$ inequalities on $n$ unknowns, the running time is $O(nm)$.

***24.4-6***
Suppose that in addition to a system of difference constraints, we want to handle ***equality constraints*** of the form $x_i = x_j + b_k$. Show how the Bellman-Ford algorithm can be adapted to solve this variety of constraint system.

***24.4-7***
Show how a system of difference constraints can be solved by a Bellman-Ford-like algorithm that runs on a constraint graph without the extra vertex $v_0$.

***24.4-8*** ⋆
Let $Ax \le b$ be a system of $m$ difference constraints in $n$ unknowns. Show that the Bellman-Ford algorithm, when run on the corresponding constraint graph, maximizes $\sum_{i=1}^{n} x_i$ subject to $Ax \le b$ and $x_i \le 0$ for all $x_i$.

***24.4-9*** ⋆
Show that the Bellman-Ford algorithm, when run on the constraint graph for a system $Ax \leq b$ of difference constraints, minimizes the quantity $(\max\{x_i\} - \min\{x_i\})$ subject to $Ax \leq b$. Explain how this fact might come in handy if the algorithm is used to schedule construction jobs.

***24.4-10***
Suppose that every row in the matrix $A$ of a linear program $Ax \leq b$ corresponds to a difference constraint, a single-variable constraint of the form $x_i \leq b_k$, or a single-variable constraint of the form $-x_i \leq b_k$. Show how to adapt the Bellman-Ford algorithm to solve this variety of constraint system.

***24.4-11***
Give an efficient algorithm to solve a system $Ax \leq b$ of difference constraints when all of the elements of $b$ are real-valued and all of the unknowns $x_i$ must be integers.

***24.4-12*** ⋆
Give an efficient algorithm to solve a system $Ax \leq b$ of difference constraints when all of the elements of $b$ are real-valued and a specified subset of some, but not necessarily all, of the unknowns $x_i$ must be integers.

## 24.5 Proofs of shortest-paths properties

Throughout this chapter, our correctness arguments have relied on the triangle inequality, upper-bound property, no-path property, convergence property, path-relaxation property, and predecessor-subgraph property. We stated these properties without proof at the beginning of this chapter. In this section, we prove them.

### The triangle inequality

In studying breadth-first search (Section 22.2), we proved as Lemma 22.1 a simple property of shortest distances in unweighted graphs. The triangle inequality generalizes the property to weighted graphs.

***Lemma 24.10 (Triangle inequality)***
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbf{R}$ and source vertex $s$. Then, for all edges $(u, v) \in E$, we have

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

***Proof*** Suppose that there is a shortest path $p$ from source $s$ to vertex $v$. Then $p$ has no more weight than any other path from $s$ to $v$. Specifically, path $p$ has no more weight than the particular path that takes a shortest path from source $s$ to vertex $u$ and then takes edge $(u, v)$.

Exercise 24.5-3 asks you to handle the case in which there is no shortest path from $s$ to $v$. ■

### Effects of relaxation on shortest-path estimates

The next group of lemmas describes how shortest-path estimates are affected when we execute a sequence of relaxation steps on the edges of a weighted, directed graph that has been initialized by INITIALIZE-SINGLE-SOURCE.

***Lemma 24.11 (Upper-bound property)***
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbf{R}$. Let $s \in V$ be the source vertex, and let the graph be initialized by INITIALIZE-SINGLE-SOURCE$(G, s)$. Then, $d[v] \geq \delta(s, v)$ for all $v \in V$, and this invariant is maintained over any sequence of relaxation steps on the edges of $G$. Moreover, once $d[v]$ achieves its lower bound $\delta(s, v)$, it never changes.

***Proof*** We prove the invariant $d[v] \geq \delta(s, v)$ for all vertices $v \in V$ by induction over the number of relaxation steps.

For the basis, $d[v] \geq \delta(s, v)$ is certainly true after initialization, since $d[s] = 0 \geq \delta(s, s)$ (note that $\delta(s, s)$ is $-\infty$ if $s$ is on a negative-weight cycle and 0 otherwise) and $d[v] = \infty$ implies $d[v] \geq \delta(s, v)$ for all $v \in V - \{s\}$.

For the inductive step, consider the relaxation of an edge $(u, v)$. By the inductive hypothesis, $d[x] \geq \delta(s, x)$ for all $x \in V$ prior to the relaxation. The only $d$ value that may change is $d[v]$. If it changes, we have

$$\begin{aligned} d[v] &= d[u] + w(u, v) \\ &\geq \delta(s, u) + w(u, v) && \text{(by inductive hypothesis)} \\ &\geq \delta(s, v) && \text{(by the triangle inequality)} . \end{aligned}$$

and so the invariant is maintained.

To see that the value of $d[v]$ never changes once $d[v] = \delta(s, v)$, note that having achieved its lower bound, $d[v]$ cannot decrease because we have just shown that $d[v] \geq \delta(s, v)$, and it cannot increase because relaxation steps do not increase $d$ values. ■

***Corollary 24.12 (No-path property)***
Suppose that in a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbf{R}$, no path connects a source vertex $s \in V$ to a given vertex $v \in V$. Then, after the graph is initialized by INITIALIZE-SINGLE-SOURCE$(G, s)$, we

have $d[v] = \delta(s, v) = \infty$, and this equality is maintained as an invariant over any sequence of relaxation steps on the edges of $G$.

***Proof*** By the upper-bound property, we always have $\infty = \delta(s, v) \leq d[v]$, and thus $d[v] = \infty = \delta(s, v)$. ■

***Lemma 24.13***
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbf{R}$, and let $(u, v) \in E$. Then, immediately after relaxing edge $(u, v)$ by executing RELAX$(u, v, w)$, we have $d[v] \leq d[u] + w(u, v)$.

***Proof*** If, just prior to relaxing edge $(u, v)$, we have $d[v] > d[u] + w(u, v)$, then $d[v] = d[u] + w(u, v)$ afterward. If, instead, $d[v] \leq d[u] + w(u, v)$ just before the relaxation, then neither $d[u]$ nor $d[v]$ changes, and so $d[v] \leq d[u] + w(u, v)$ afterward. ■

***Lemma 24.14 (Convergence property)***
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbf{R}$, let $s \in V$ be a source vertex, and let $s \rightsquigarrow u \to v$ be a shortest path in $G$ for some vertices $u, v \in V$. Suppose that $G$ is initialized by INITIALIZE-SINGLE-SOURCE$(G, s)$ and then a sequence of relaxation steps that includes the call RELAX$(u, v, w)$ is executed on the edges of $G$. If $d[u] = \delta(s, u)$ at any time prior to the call, then $d[v] = \delta(s, v)$ at all times after the call.

***Proof*** By the upper-bound property, if $d[u] = \delta(s, u)$ at some point prior to relaxing edge $(u, v)$, then this equality holds thereafter. In particular, after relaxing edge $(u, v)$, we have

$$\begin{aligned} d[v] &\leq d[u] + w(u, v) && \text{(by Lemma 24.13)} \\ &= \delta(s, u) + w(u, v) \\ &= \delta(s, v) && \text{(by Lemma 24.1)} \ . \end{aligned}$$

By the upper-bound property, $d[v] \geq \delta(s, v)$, from which we conclude that $d[v] = \delta(s, v)$, and this equality is maintained thereafter. ■

***Lemma 24.15 (Path-relaxation property)***
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbf{R}$, and let $s \in V$ be a source vertex. Consider any shortest path $p = \langle v_0, v_1, \ldots, v_k \rangle$ from $s = v_0$ to $v_k$. If $G$ is initialized by INITIALIZE-SINGLE-SOURCE$(G, s)$ and then a sequence of relaxation steps occurs that includes, in order, relaxations of edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $d[v_k] = \delta(s, v_k)$ after these relaxations and at all times afterward. This property holds no matter what other edge

relaxations occur, including relaxations that are intermixed with relaxations of the edges of $p$.

***Proof*** We show by induction that after the $i$th edge of path $p$ is relaxed, we have $d[v_i] = \delta(s, v_i)$. For the basis, $i = 0$, and before any edges of $p$ have been relaxed, we have from the initialization that $d[v_0] = d[s] = 0 = \delta(s, s)$. By the upper-bound property, the value of $d[s]$ never changes after initialization.

For the inductive step, we assume that $d[v_{i-1}] = \delta(s, v_{i-1})$, and we examine the relaxation of edge $(v_{i-1}, v_i)$. By the convergence property, after this relaxation, we have $d[v_i] = \delta(s, v_i)$, and this equality is maintained at all times thereafter. ■

### Relaxation and shortest-paths trees

We now show that once a sequence of relaxations has caused the shortest-path estimates to converge to shortest-path weights, the predecessor subgraph $G_\pi$ induced by the resulting $\pi$ values is a shortest-paths tree for $G$. We start with the following lemma, which shows that the predecessor subgraph always forms a rooted tree whose root is the source.

***Lemma 24.16***
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbf{R}$, let $s \in V$ be a source vertex, and assume that $G$ contains no negative-weight cycles that are reachable from $s$. Then, after the graph is initialized by INITIALIZE-SINGLE-SOURCE$(G, s)$, the predecessor subgraph $G_\pi$ forms a rooted tree with root $s$, and any sequence of relaxation steps on edges of $G$ maintains this property as an invariant.

***Proof*** Initially, the only vertex in $G_\pi$ is the source vertex, and the lemma is trivially true. Consider a predecessor subgraph $G_\pi$ that arises after a sequence of relaxation steps. We shall first prove that $G_\pi$ is acyclic. Suppose for the sake of contradiction that some relaxation step creates a cycle in the graph $G_\pi$. Let the cycle be $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_k = v_0$. Then, $\pi[v_i] = v_{i-1}$ for $i = 1, 2, \ldots, k$ and, without loss of generality, we can assume that it was the relaxation of edge $(v_{k-1}, v_k)$ that created the cycle in $G_\pi$.

We claim that all vertices on cycle $c$ are reachable from the source $s$. Why? Each vertex on $c$ has a non-NIL predecessor, and so each vertex on $c$ was assigned a finite shortest-path estimate when it was assigned its non-NIL $\pi$ value. By the upper-bound property, each vertex on cycle $c$ has a finite shortest-path weight, which implies that it is reachable from $s$.

We shall examine the shortest-path estimates on $c$ just prior to the call RELAX$(v_{k-1}, v_k, w)$ and show that $c$ is a negative-weight cycle, thereby contradicting the assumption that $G$ contains no negative-weight cycles that are reachable

from the source. Just before the call, we have $\pi[v_i] = v_{i-1}$ for $i = 1, 2, \ldots, k-1$. Thus, for $i = 1, 2, \ldots, k-1$, the last update to $d[v_i]$ was by the assignment $d[v_i] \leftarrow d[v_{i-1}] + w(v_{i-1}, v_i)$. If $d[v_{i-1}]$ changed since then, it decreased. Therefore, just before the call RELAX$(v_{k-1}, v_k, w)$, we have

$$d[v_i] \geq d[v_{i-1}] + w(v_{i-1}, v_i) \qquad \text{for all } i = 1, 2, \ldots, k-1 . \tag{24.12}$$

Because $\pi[v_k]$ is changed by the call, immediately beforehand we also have the strict inequality

$$d[v_k] > d[v_{k-1}] + w(v_{k-1}, v_k) .$$

Summing this strict inequality with the $k-1$ inequalities (24.12), we obtain the sum of the shortest-path estimates around cycle $c$:

$$\begin{aligned} \sum_{i=1}^{k} d[v_i] &> \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i) . \end{aligned}$$

But

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}] ,$$

since each vertex in the cycle $c$ appears exactly once in each summation. This equality implies

$$0 > \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

Thus, the sum of weights around the cycle $c$ is negative, which provides the desired contradiction.

We have now proven that $G_\pi$ is a directed, acyclic graph. To show that it forms a rooted tree with root $s$, it suffices (see Exercise B.5-2) to prove that for each vertex $v \in V_\pi$, there is a unique path from $s$ to $v$ in $G_\pi$.

We first must show that a path from $s$ exists for each vertex in $V_\pi$. The vertices in $V_\pi$ are those with non-NIL $\pi$ values, plus $s$. The idea here is to prove by induction that a path exists from $s$ to all vertices in $V_\pi$. The details are left as Exercise 24.5-6.

To complete the proof of the lemma, we must now show that for any vertex $v \in V_\pi$, there is at most one path from $s$ to $v$ in the graph $G_\pi$. Suppose otherwise. That is, suppose that there are two simple paths from $s$ to some vertex $v$: $p_1$, which can be decomposed into $s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v$, and $p_2$, which can be decomposed

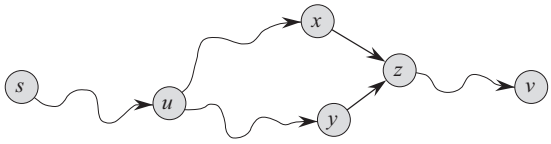



**Figure 24.9** Showing that a path in $G_\pi$ from source $s$ to vertex $v$ is unique. If there are two paths $p_1$ ($s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v$) and $p_2$ ($s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v$), where $x \neq y$, then $\pi[z] = x$ and $\pi[z] = y$, a contradiction.

into $s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v$, where $x \neq y$. (See Figure 24.9.) But then, $\pi[z] = x$ and $\pi[z] = y$, which implies the contradiction that $x = y$. We conclude that there exists a unique simple path in $G_\pi$ from $s$ to $v$, and thus $G_\pi$ forms a rooted tree with root $s$. ■

We can now show that if, after we have performed a sequence of relaxation steps, all vertices have been assigned their true shortest-path weights, then the predecessor subgraph $G_\pi$ is a shortest-paths tree.

***Lemma 24.17 (Predecessor-subgraph property)***
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \rightarrow \mathbf{R}$, let $s \in V$ be a source vertex, and assume that $G$ contains no negative-weight cycles that are reachable from $s$. Let us call INITIALIZE-SINGLE-SOURCE$(G, s)$ and then execute any sequence of relaxation steps on edges of $G$ that produces $d[v] = \delta(s, v)$ for all $v \in V$. Then, the predecessor subgraph $G_\pi$ is a shortest-paths tree rooted at $s$.

***Proof*** We must prove that the three properties of shortest-paths trees given on page 584 hold for $G_\pi$. To show the first property, we must show that $V_\pi$ is the set of vertices reachable from $s$. By definition, a shortest-path weight $\delta(s, v)$ is finite if and only if $v$ is reachable from $s$, and thus the vertices that are reachable from $s$ are exactly those with finite $d$ values. But a vertex $v \in V - \{s\}$ has been assigned a finite value for $d[v]$ if and only if $\pi[v] \neq$ NIL. Thus, the vertices in $V_\pi$ are exactly those reachable from $s$.

The second property follows directly from Lemma 24.16.

It remains, therefore, to prove the last property of shortest-paths trees: for each vertex $v \in V_\pi$, the unique simple path $s \overset{p}{\rightsquigarrow} v$ in $G_\pi$ is a shortest path from $s$ to $v$ in $G$. Let $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = v$. For $i = 1, 2, \ldots, k$, we have both $d[v_i] = \delta(s, v_i)$ and $d[v_i] \geq d[v_{i-1}] + w(v_{i-1}, v_i)$, from which we conclude $w(v_{i-1}, v_i) \leq \delta(s, v_i) - \delta(s, v_{i-1})$. Summing the weights along path $p$

yields

$$\begin{aligned} w(p) &= \sum_{i=1}^{k} w(v_{i-1}, v_i) \\ &\le \sum_{i=1}^{k} (\delta(s, v_i) - \delta(s, v_{i-1})) \\ &= \delta(s, v_k) - \delta(s, v_0) && \text{(because the sum telescopes)} \\ &= \delta(s, v_k) && \text{(because } \delta(s, v_0) = \delta(s, s) = 0) \ . \end{aligned}$$

Thus, $w(p) \le \delta(s, v_k)$. Since $\delta(s, v_k)$ is a lower bound on the weight of any path from $s$ to $v_k$, we conclude that $w(p) = \delta(s, v_k)$, and thus $p$ is a shortest path from $s$ to $v = v_k$. ■

## Exercises

***24.5-1***

Give two shortest-paths trees for the directed graph of Figure 24.2 other than the two shown.

***24.5-2***

Give an example of a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbf{R}$ and source $s$ such that $G$ satisfies the following property: For every edge $(u, v) \in E$, there is a shortest-paths tree rooted at $s$ that contains $(u, v)$ and another shortest-paths tree rooted at $s$ that does not contain $(u, v)$.

***24.5-3***

Embellish the proof of Lemma 24.10 to handle cases in which shortest-path weights are $\infty$ or $-\infty$.

***24.5-4***

Let $G = (V, E)$ be a weighted, directed graph with source vertex $s$, and let $G$ be initialized by INITIALIZE-SINGLE-SOURCE$(G, s)$. Prove that if a sequence of relaxation steps sets $\pi[s]$ to a non-NIL value, then $G$ contains a negative-weight cycle.

***24.5-5***

Let $G = (V, E)$ be a weighted, directed graph with no negative-weight edges. Let $s \in V$ be the source vertex, and suppose that we allow $\pi[v]$ to be the predecessor of $v$ on *any* shortest path to $v$ from source $s$ if $v \in V - \{s\}$ is reachable from $s$, and NIL otherwise. Give an example of such a graph $G$ and an assignment of $\pi$ values that produces a cycle in $G_\pi$. (By Lemma 24.16, such an assignment cannot be produced by a sequence of relaxation steps.)

***24.5-6***
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbf{R}$ and no negative-weight cycles. Let $s \in V$ be the source vertex, and let $G$ be initialized by INITIALIZE-SINGLE-SOURCE$(G, s)$. Prove that for every vertex $v \in V_\pi$, there exists a path from $s$ to $v$ in $G_\pi$ and that this property is maintained as an invariant over any sequence of relaxations.

***24.5-7***
Let $G = (V, E)$ be a weighted, directed graph that contains no negative-weight cycles. Let $s \in V$ be the source vertex, and let $G$ be initialized by INITIALIZE-SINGLE-SOURCE$(G, s)$. Prove that there exists a sequence of $|V| - 1$ relaxation steps that produces $d[v] = \delta(s, v)$ for all $v \in V$.

***24.5-8***
Let $G$ be an arbitrary weighted, directed graph with a negative-weight cycle reachable from the source vertex $s$. Show that an infinite sequence of relaxations of the edges of $G$ can always be constructed such that every relaxation causes a shortest-path estimate to change.

## Problems

***24-1 Yen's improvement to Bellman-Ford***
Suppose that we order the edge relaxations in each pass of the Bellman-Ford algorithm as follows. Before the first pass, we assign an arbitrary linear order $v_1, v_2, \ldots, v_{|V|}$ to the vertices of the input graph $G = (V, E)$. Then, we partition the edge set $E$ into $E_f \cup E_b$, where $E_f = \{(v_i, v_j) \in E : i < j\}$ and $E_b = \{(v_i, v_j) \in E : i > j\}$. (Assume that $G$ contains no self-loops, so that every edge is in either $E_f$ or $E_b$.) Define $G_f = (V, E_f)$ and $G_b = (V, E_b)$.

***a.*** Prove that $G_f$ is acyclic with topological sort $\langle v_1, v_2, \ldots, v_{|V|}\rangle$ and that $G_b$ is acyclic with topological sort $\langle v_{|V|}, v_{|V|-1}, \ldots, v_1\rangle$.

Suppose that we implement each pass of the Bellman-Ford algorithm in the following way. We visit each vertex in the order $v_1, v_2, \ldots, v_{|V|}$, relaxing edges of $E_f$ that leave the vertex. We then visit each vertex in the order $v_{|V|}, v_{|V|-1}, \ldots, v_1$, relaxing edges of $E_b$ that leave the vertex.

***b.*** Prove that with this scheme, if $G$ contains no negative-weight cycles that are reachable from the source vertex $s$, then after only $\lceil |V|/2 \rceil$ passes over the edges, $d[v] = \delta(s, v)$ for all vertices $v \in V$.

***c.*** Does this scheme improve the asymptotic running time of the Bellman-Ford algorithm?

### *24-2 Nesting boxes*

A $d$-dimensional box with dimensions $(x_1, x_2, \ldots, x_d)$ ***nests*** within another box with dimensions $(y_1, y_2, \ldots, y_d)$ if there exists a permutation $\pi$ on $\{1, 2, \ldots, d\}$ such that $x_{\pi(1)} < y_1, x_{\pi(2)} < y_2, \ldots, x_{\pi(d)} < y_d$.

***a.*** Argue that the nesting relation is transitive.

***b.*** Describe an efficient method to determine whether or not one $d$-dimensional box nests inside another.

***c.*** Suppose that you are given a set of $n$ $d$-dimensional boxes $\{B_1, B_2, \ldots, B_n\}$. Describe an efficient algorithm to determine the longest sequence $\langle B_{i_1}, B_{i_2}, \ldots, B_{i_k} \rangle$ of boxes such that $B_{i_j}$ nests within $B_{i_{j+1}}$ for $j = 1, 2, \ldots, k - 1$. Express the running time of your algorithm in terms of $n$ and $d$.

### *24-3 Arbitrage*

***Arbitrage*** is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that 1 U.S. dollar buys 46.4 Indian rupees, 1 Indian rupee buys 2.5 Japanese yen, and 1 Japanese yen buys 0.0091 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy $46.4 \times 2.5 \times 0.0091 = 1.0556$ U.S. dollars, thus turning a profit of 5.56 percent.

Suppose that we are given $n$ currencies $c_1, c_2, \ldots, c_n$ and an $n \times n$ table $R$ of exchange rates, such that one unit of currency $c_i$ buys $R[i, j]$ units of currency $c_j$.

***a.*** Give an efficient algorithm to determine whether or not there exists a sequence of currencies $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} \rangle$ such that

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1 \, .$$

Analyze the running time of your algorithm.

***b.*** Give an efficient algorithm to print out such a sequence if one exists. Analyze the running time of your algorithm.

### *24-4 Gabow's scaling algorithm for single-source shortest paths*

A ***scaling*** algorithm solves a problem by initially considering only the highest-order bit of each relevant input value (such as an edge weight). It then refines the initial solution by looking at the two highest-order bits. It progressively looks at

more and more high-order bits, refining the solution each time, until all bits have been considered and the correct solution has been computed.

In this problem, we examine an algorithm for computing the shortest paths from a single source by scaling edge weights. We are given a directed graph $G = (V, E)$ with nonnegative integer edge weights $w$. Let $W = \max_{(u,v)\in E}\{w(u, v)\}$. Our goal is to develop an algorithm that runs in $O(E \lg W)$ time. We assume that all vertices are reachable from the source.

The algorithm uncovers the bits in the binary representation of the edge weights one at a time, from the most significant bit to the least significant bit. Specifically, let $k = \lceil \lg(W + 1) \rceil$ be the number of bits in the binary representation of $W$, and for $i = 1, 2, \ldots, k$, let $w_i(u, v) = \lfloor w(u, v)/2^{k-i} \rfloor$. That is, $w_i(u, v)$ is the "scaled-down" version of $w(u, v)$ given by the $i$ most significant bits of $w(u, v)$. (Thus, $w_k(u, v) = w(u, v)$ for all $(u, v) \in E$.) For example, if $k = 5$ and $w(u, v) = 25$, which has the binary representation $\langle 11001 \rangle$, then $w_3(u, v) = \langle 110 \rangle = 6$. As another example with $k = 5$, if $w(u, v) = \langle 00100 \rangle = 4$, then $w_3(u, v) = \langle 001 \rangle = 1$. Let us define $\delta_i(u, v)$ as the shortest-path weight from vertex $u$ to vertex $v$ using weight function $w_i$. Thus, $\delta_k(u, v) = \delta(u, v)$ for all $u, v \in V$. For a given source vertex $s$, the scaling algorithm first computes the shortest-path weights $\delta_1(s, v)$ for all $v \in V$, then computes $\delta_2(s, v)$ for all $v \in V$, and so on, until it computes $\delta_k(s, v)$ for all $v \in V$. We assume throughout that $|E| \geq |V| - 1$, and we shall see that computing $\delta_i$ from $\delta_{i-1}$ takes $O(E)$ time, so that the entire algorithm takes $O(kE) = O(E \lg W)$ time.

***a.*** Suppose that for all vertices $v \in V$, we have $\delta(s, v) \leq |E|$. Show that we can compute $\delta(s, v)$ for all $v \in V$ in $O(E)$ time.

***b.*** Show that we can compute $\delta_1(s, v)$ for all $v \in V$ in $O(E)$ time.

Let us now focus on computing $\delta_i$ from $\delta_{i-1}$.

***c.*** Prove that for $i = 2, 3, \ldots, k$, we have either $w_i(u, v) = 2w_{i-1}(u, v)$ or $w_i(u, v) = 2w_{i-1}(u, v) + 1$. Then, prove that

$$2\delta_{i-1}(s, v) \leq \delta_i(s, v) \leq 2\delta_{i-1}(s, v) + |V| - 1$$

for all $v \in V$.

***d.*** Define for $i = 2, 3, \ldots, k$ and all $(u, v) \in E$,

$$\widehat{w}_i(u, v) = w_i(u, v) + 2\delta_{i-1}(s, u) - 2\delta_{i-1}(s, v) .$$

Prove that for $i = 2, 3, \ldots, k$ and all $u, v \in V$, the "reweighted" value $\widehat{w}_i(u, v)$ of edge $(u, v)$ is a nonnegative integer.

***e.*** Now, define $\widehat{\delta}_i(s, v)$ as the shortest-path weight from $s$ to $v$ using the weight function $\widehat{w}_i$. Prove that for $i = 2, 3, \ldots, k$ and all $v \in V$,

$$\delta_i(s, v) = \widehat{\delta}_i(s, v) + 2\delta_{i-1}(s, v)$$

and that $\widehat{\delta}_i(s, v) \leq |E|$.

***f.*** Show how to compute $\delta_i(s, v)$ from $\delta_{i-1}(s, v)$ for all $v \in V$ in $O(E)$ time, and conclude that $\delta(s, v)$ can be computed for all $v \in V$ in $O(E \lg W)$ time.

### *24-5 Karp's minimum mean-weight cycle algorithm*

Let $G = (V, E)$ be a directed graph with weight function $w : E \to \mathbf{R}$, and let $n = |V|$. We define the ***mean weight*** of a cycle $c = \langle e_1, e_2, \ldots, e_k \rangle$ of edges in $E$ to be

$$\mu(c) = \frac{1}{k} \sum_{i=1}^{k} w(e_i) \; .$$

Let $\mu^* = \min_c \mu(c)$, where $c$ ranges over all directed cycles in $G$. A cycle $c$ for which $\mu(c) = \mu^*$ is called a ***minimum mean-weight cycle***. This problem investigates an efficient algorithm for computing $\mu^*$.

Assume without loss of generality that every vertex $v \in V$ is reachable from a source vertex $s \in V$. Let $\delta(s, v)$ be the weight of a shortest path from $s$ to $v$, and let $\delta_k(s, v)$ be the weight of a shortest path from $s$ to $v$ consisting of *exactly* $k$ edges. If there is no path from $s$ to $v$ with exactly $k$ edges, then $\delta_k(s, v) = \infty$.

***a.*** Show that if $\mu^* = 0$, then $G$ contains no negative-weight cycles and $\delta(s, v) = \min_{0 \leq k \leq n-1} \delta_k(s, v)$ for all vertices $v \in V$.

***b.*** Show that if $\mu^* = 0$, then

$$\max_{0 \leq k \leq n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} \geq 0$$

for all vertices $v \in V$. (*Hint:* Use both properties from part (a).)

***c.*** Let $c$ be a 0-weight cycle, and let $u$ and $v$ be any two vertices on $c$. Suppose that $\mu^* = 0$ and that the weight of the path from $u$ to $v$ along the cycle is $x$. Prove that $\delta(s, v) = \delta(s, u) + x$. (*Hint:* The weight of the path from $v$ to $u$ along the cycle is $-x$.)

***d.*** Show that if $\mu^* = 0$, then on each minimum mean-weight cycle there exists a vertex $v$ such that

$$\max_{0 \le k \le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0 .$$

(*Hint:* Show that a shortest path to any vertex on a minimum mean-weight cycle can be extended along the cycle to make a shortest path to the next vertex on the cycle.)

***e.*** Show that if $\mu^* = 0$, then

$$\min_{v \in V} \max_{0 \le k \le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0 .$$

***f.*** Show that if we add a constant $t$ to the weight of each edge of $G$, then $\mu^*$ is increased by $t$. Use this fact to show that

$$\mu^* = \min_{v \in V} \max_{0 \le k \le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} .$$

***g.*** Give an $O(VE)$-time algorithm to compute $\mu^*$.

***24-6 Bitonic shortest paths***

A sequence is ***bitonic*** if it monotonically increases and then monotonically decreases, or if it can be circularly shifted to monotonically increase and then monotonically decrease. For example the sequences $\langle 1, 4, 6, 8, 3, -2 \rangle$, $\langle 9, 2, -4, -10, -5 \rangle$, and $\langle 1, 2, 3, 4 \rangle$ are bitonic, but $\langle 1, 3, 12, 4, 2, 10 \rangle$ is not bitonic. (See Chapter 27 for a discussion of bitonic sorters, and see Problem 15-1 for the bitonic euclidean traveling-salesman problem.)

Suppose that we are given a directed graph $G = (V, E)$ with weight function $w : E \to \mathbf{R}$, and we wish to find single-source shortest paths from a source vertex $s$. We are given one additional piece of information: for each vertex $v \in V$, the weights of the edges along any shortest path from $s$ to $v$ form a bitonic sequence.

Give the most efficient algorithm you can to solve this problem, and analyze its running time.

## Chapter notes

Dijkstra's algorithm [75] appeared in 1959, but it contained no mention of a priority queue. The Bellman-Ford algorithm is based on separate algorithms by Bellman [35] and Ford [93]. Bellman describes the relation of shortest paths to difference constraints. Lawler [196] describes the linear-time algorithm for shortest paths in a dag, which he considers part of the folklore.

When edge weights are relatively small nonnegative integers, more efficient algorithms can be used to solve the single-source shortest-paths problem. The sequence of values returned by the EXTRACT-MIN calls in Dijkstra's algorithm is monotonically increasing over time. As discussed in the chapter notes for Chapter 6, in this case there are several data structures that can implement the various priority-queue operations more efficiently than a binary heap or a Fibonacci heap. Ahuja, Mehlhorn, Orlin, and Tarjan [8] give an algorithm that runs in $O(E + V\sqrt{\lg W})$ time on graphs with nonnegative edge weights, where $W$ is the largest weight of any edge in the graph. The best bounds are by Thorup [299], who gives an algorithm that runs in $O(E \lg\lg V)$ time, and by Raman, who gives an algorithm that runs in $O(E + V \min\{(\lg V)^{1/3+\epsilon}, (\lg W)^{1/4+\epsilon}\})$ time. These two algorithms use an amount of space that depends on the word size of the underlying machine. Although the amount of space used can be unbounded in the size of the input, it can be reduced to be linear in the size of the input using randomized hashing.

For undirected graphs with integer weights, Thorup [298] gives an $O(V + E)$-time algorithm for single-source shortest paths. In contrast to the algorithms mentioned in the previous paragraph, this algorithm is not an implementation of Dijkstra's algorithm, since the sequence of values returned by EXTRACT-MIN calls is not monotonically increasing over time.

For graphs with negative edge weights, an algorithm due to Gabow and Tarjan [104] runs in $O(\sqrt{V}E \lg(VW))$ time, and one by Goldberg [118] runs in $O(\sqrt{V}E \lg W)$ time, where $W = \max_{(u,v)\in E}\{|w(u, v)|\}$.

Cherkassky, Goldberg, and Radzik [57] conducted extensive experiments comparing various shortest-path algorithms.

# 25 All-Pairs Shortest Paths

In this chapter, we consider the problem of finding shortest paths between all pairs of vertices in a graph. This problem might arise in making a table of distances between all pairs of cities for a road atlas. As in Chapter 24, we are given a weighted, directed graph $G = (V, E)$ with a weight function $w : E \to \mathbf{R}$ that maps edges to real-valued weights. We wish to find, for every pair of vertices $u, v \in V$, a shortest (least-weight) path from $u$ to $v$, where the weight of a path is the sum of the weights of its constituent edges. We typically want the output in tabular form: the entry in $u$'s row and $v$'s column should be the weight of a shortest path from $u$ to $v$.

We can solve an all-pairs shortest-paths problem by running a single-source shortest-paths algorithm $|V|$ times, once for each vertex as the source. If all edge weights are nonnegative, we can use Dijkstra's algorithm. If we use the linear-array implementation of the min-priority queue, the running time is $O(V^3 + VE) = O(V^3)$. The binary min-heap implementation of the min-priority queue yields a running time of $O(VE \lg V)$, which is an improvement if the graph is sparse. Alternatively, we can implement the min-priority queue with a Fibonacci heap, yielding a running time of $O(V^2 \lg V + VE)$.

If negative-weight edges are allowed, Dijkstra's algorithm can no longer be used. Instead, we must run the slower Bellman-Ford algorithm once from each vertex. The resulting running time is $O(V^2E)$, which on a dense graph is $O(V^4)$. In this chapter we shall see how to do better. We shall also investigate the relation of the all-pairs shortest-paths problem to matrix multiplication and study its algebraic structure.

Unlike the single-source algorithms, which assume an adjacency-list representation of the graph, most of the algorithms in this chapter use an adjacency-matrix representation. (Johnson's algorithm for sparse graphs uses adjacency lists.) For convenience, we assume that the vertices are numbered $1, 2, \ldots, |V|$, so that the input is an $n \times n$ matrix $W$ representing the edge weights of an $n$-vertex directed graph $G = (V, E)$. That is, $W = (w_{ij})$, where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \ , \\ \text{the weight of directed edge } (i, j) & \text{if } i \neq j \text{ and } (i, j) \in E \ , \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E \ . \end{cases} \tag{25.1}$$

Negative-weight edges are allowed, but we assume for the time being that the input graph contains no negative-weight cycles.

The tabular output of the all-pairs shortest-paths algorithms presented in this chapter is an $n \times n$ matrix $D = (d_{ij})$, where entry $d_{ij}$ contains the weight of a shortest path from vertex $i$ to vertex $j$. That is, if we let $\delta(i, j)$ denote the shortest-path weight from vertex $i$ to vertex $j$ (as in Chapter 24), then $d_{ij} = \delta(i, j)$ at termination.

To solve the all-pairs shortest-paths problem on an input adjacency matrix, we need to compute not only the shortest-path weights but also a ***predecessor matrix*** $\Pi = (\pi_{ij})$, where $\pi_{ij}$ is NIL if either $i = j$ or there is no path from $i$ to $j$, and otherwise $\pi_{ij}$ is the predecessor of $j$ on some shortest path from $i$. Just as the predecessor subgraph $G_\pi$ from Chapter 24 is a shortest-paths tree for a given source vertex, the subgraph induced by the $i$th row of the $\Pi$ matrix should be a shortest-paths tree with root $i$. For each vertex $i \in V$, we define the ***predecessor subgraph*** of $G$ for $i$ as $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$ , where

$$V_{\pi,i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\}$$

and

$$E_{\pi,i} = \{(\pi_{ij}, j) : j \in V_{\pi,i} - \{i\}\} \ .$$

If $G_{\pi,i}$ is a shortest-paths tree, then the following procedure, which is a modified version of the PRINT-PATH procedure from Chapter 22, prints a shortest path from vertex $i$ to vertex $j$.

```
PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)
1  if i = j
2     then print i
3     else if π_ij = NIL
4             then print "no path from" i "to" j "exists"
5             else PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_ij)
6                  print j
```

In order to highlight the essential features of the all-pairs algorithms in this chapter, we won't cover the creation and properties of predecessor matrices as extensively as we dealt with predecessor subgraphs in Chapter 24. The basics are covered by some of the exercises.

### Chapter outline

Section 25.1 presents a dynamic-programming algorithm based on matrix multiplication to solve the all-pairs shortest-paths problem. Using the technique of "repeated squaring," this algorithm can be made to run in $\Theta(V^3 \lg V)$ time. Another dynamic-programming algorithm, the Floyd-Warshall algorithm, is given in Section 25.2. The Floyd-Warshall algorithm runs in time $\Theta(V^3)$. Section 25.2 also covers the problem of finding the transitive closure of a directed graph, which is related to the all-pairs shortest-paths problem. Finally, Section 25.3 presents Johnson's algorithm. Unlike the other algorithms in this chapter, Johnson's algorithm uses the adjacency-list representation of a graph. It solves the all-pairs shortest-paths problem in $O(V^2 \lg V + VE)$ time, which makes it a good algorithm for large, sparse graphs.

Before proceeding, we need to establish some conventions for adjacency-matrix representations. First, we shall generally assume that the input graph $G = (V, E)$ has $n$ vertices, so that $n = |V|$. Second, we shall use the convention of denoting matrices by uppercase letters, such as $W$, $L$, or $D$, and their individual elements by subscripted lowercase letters, such as $w_{ij}$, $l_{ij}$, or $d_{ij}$. Some matrices will have parenthesized superscripts, as in $L^{(m)} = \left(l_{ij}^{(m)}\right)$ or $D^{(m)} = \left(d_{ij}^{(m)}\right)$, to indicate iterates. Finally, for a given $n \times n$ matrix $A$, we shall assume that the value of $n$ is stored in the attribute $rows[A]$.

## 25.1 Shortest paths and matrix multiplication

This section presents a dynamic-programming algorithm for the all-pairs shortest-paths problem on a directed graph $G = (V, E)$. Each major loop of the dynamic program will invoke an operation that is very similar to matrix multiplication, so that the algorithm will look like repeated matrix multiplication. We shall start by developing a $\Theta(V^4)$-time algorithm for the all-pairs shortest-paths problem and then improve its running time to $\Theta(V^3 \lg V)$.

Before proceeding, let us briefly recap the steps given in Chapter 15 for developing a dynamic-programming algorithm.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.

(The fourth step, constructing an optimal solution from computed information, is dealt with in the exercises.)

### The structure of a shortest path

We start by characterizing the structure of an optimal solution. For the all-pairs shortest-paths problem on a graph $G = (V, E)$, we have proven (Lemma 24.1) that all subpaths of a shortest path are shortest paths. Suppose that the graph is represented by an adjacency matrix $W = (w_{ij})$. Consider a shortest path $p$ from vertex $i$ to vertex $j$, and suppose that $p$ contains at most $m$ edges. Assuming that there are no negative-weight cycles, $m$ is finite. If $i = j$, then $p$ has weight 0 and no edges. If vertices $i$ and $j$ are distinct, then we decompose path $p$ into $i \overset{p'}{\rightsquigarrow} k \to j$, where path $p'$ now contains at most $m - 1$ edges. By Lemma 24.1, $p'$ is a shortest path from $i$ to $k$, and so $\delta(i, j) = \delta(i, k) + w_{kj}$.

### A recursive solution to the all-pairs shortest-paths problem

Now, let $l_{ij}^{(m)}$ be the minimum weight of any path from vertex $i$ to vertex $j$ that contains at most $m$ edges. When $m = 0$, there is a shortest path from $i$ to $j$ with no edges if and only if $i = j$. Thus,

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \ , \\ \infty & \text{if } i \neq j \ . \end{cases}$$

For $m \geq 1$, we compute $l_{ij}^{(m)}$ as the minimum of $l_{ij}^{(m-1)}$ (the weight of the shortest path from $i$ to $j$ consisting of at most $m-1$ edges) and the minimum weight of any path from $i$ to $j$ consisting of at most $m$ edges, obtained by looking at all possible predecessors $k$ of $j$. Thus, we recursively define

$$\begin{aligned} l_{ij}^{(m)} &= \min\left(l_{ij}^{(m-1)}, \min_{1 \leq k \leq n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\}\right) \\ &= \min_{1 \leq k \leq n}\left\{l_{ik}^{(m-1)} + w_{kj}\right\} \ . \end{aligned} \tag{25.2}$$

The latter equality follows since $w_{jj} = 0$ for all $j$.

What are the actual shortest-path weights $\delta(i, j)$? If the graph contains no negative-weight cycles, then for every pair of vertices $i$ and $j$ for which $\delta(i, j) < \infty$, there is a shortest path from $i$ to $j$ that is simple and thus contains at most $n - 1$ edges. A path from vertex $i$ to vertex $j$ with more than $n - 1$ edges cannot have lower weight than a shortest path from $i$ to $j$. The actual shortest-path weights are therefore given by

$$\delta(i, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots \ . \tag{25.3}$$

### Computing the shortest-path weights bottom up

Taking as our input the matrix $W = (w_{ij})$, we now compute a series of matrices $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$, where for $m = 1, 2, \ldots, n-1$, we have $L^{(m)} = \left(l_{ij}^{(m)}\right)$.

The final matrix $L^{(n-1)}$ contains the actual shortest-path weights. Observe that $l_{ij}^{(1)} = w_{ij}$ for all vertices $i, j \in V$, and so $L^{(1)} = W$.

The heart of the algorithm is the following procedure, which, given matrices $L^{(m-1)}$ and $W$, returns the matrix $L^{(m)}$. That is, it extends the shortest paths computed so far by one more edge.

```
EXTEND-SHORTEST-PATHS(L, W)
1  n ← rows[L]
2  let L' = (l'_ij) be an n × n matrix
3  for i ← 1 to n
4       do for j ← 1 to n
5            do l'_ij ← ∞
6               for k ← 1 to n
7                    do l'_ij ← min(l'_ij, l_ik + w_kj)
8  return L'
```

The procedure computes a matrix $L' = (l'_{ij})$, which it returns at the end. It does so by computing equation (25.2) for all $i$ and $j$, using $L$ for $L^{(m-1)}$ and $L'$ for $L^{(m)}$. (It is written without the superscripts to make its input and output matrices independent of $m$.) Its running time is $\Theta(n^3)$ due to the three nested **for** loops.

Now we can see the relation to matrix multiplication. Suppose we wish to compute the matrix product $C = A \cdot B$ of two $n \times n$ matrices $A$ and $B$. Then, for $i, j = 1, 2, \ldots, n$, we compute

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \ . \tag{25.4}$$

Observe that if we make the substitutions

$$\begin{aligned} l^{(m-1)} &\rightarrow a \ , \\ w &\rightarrow b \ , \\ l^{(m)} &\rightarrow c \ , \\ \min &\rightarrow + \ , \\ + &\rightarrow \cdot \end{aligned}$$

in equation (25.2), we obtain equation (25.4). Thus, if we make these changes to EXTEND-SHORTEST-PATHS and also replace $\infty$ (the identity for min) by 0 (the identity for $+$), we obtain the straightforward $\Theta(n^3)$-time procedure for matrix multiplication:

```
MATRIX-MULTIPLY(A, B)
1  n ← rows[A]
2  let C be an n × n matrix
3  for i ← 1 to n
4       do for j ← 1 to n
5            do c_ij ← 0
6               for k ← 1 to n
7                    do c_ij ← c_ij + a_ik · b_kj
8  return C
```

Returning to the all-pairs shortest-paths problem, we compute the shortest-path weights by extending shortest paths edge by edge. Letting $A \cdot B$ denote the matrix "product" returned by EXTEND-SHORTEST-PATHS$(A, B)$, we compute the sequence of $n - 1$ matrices

$$\begin{aligned}
L^{(1)} &= L^{(0)} \cdot W &&= W\,, \\
L^{(2)} &= L^{(1)} \cdot W &&= W^2\,, \\
L^{(3)} &= L^{(2)} \cdot W &&= W^3\,, \\
&\vdots \\
L^{(n-1)} &= L^{(n-2)} \cdot W &&= W^{n-1}\,.
\end{aligned}$$

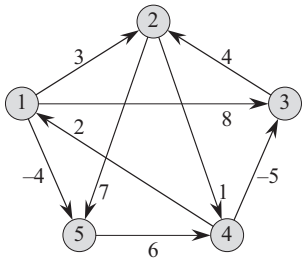
As we argued above, the matrix $L^{(n-1)} = W^{n-1}$ contains the shortest-path weights. The following procedure computes this sequence in $\Theta(n^4)$ time.

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
1  n ← rows[W]
2  L^(1) ← W
3  for m ← 2 to n − 1
4       do L^(m) ← EXTEND-SHORTEST-PATHS(L^(m−1), W)
5  return L^(n−1)
```

Figure 25.1 shows a graph and the matrices $L^{(m)}$ computed by the procedure SLOW-ALL-PAIRS-SHORTEST-PATHS.

### Improving the running time

Our goal, however, is not to compute *all* the $L^{(m)}$ matrices: we are interested only in matrix $L^{(n-1)}$. Recall that in the absence of negative-weight cycles, equation (25.3) implies $L^{(m)} = L^{(n-1)}$ for all integers $m \geq n - 1$. Just as traditional matrix multiplication is associative, so is matrix multiplication defined by the EXTEND-SHORTEST-PATHS procedure (see Exercise 25.1-4). Therefore, we can compute $L^{(n-1)}$ with only $\lceil \lg(n-1) \rceil$ matrix products by computing the sequence




$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

**Figure 25.1** A directed graph and the sequence of matrices $L^{(m)}$ computed by SLOW-ALL-PAIRS-SHORTEST-PATHS. The reader may verify that $L^{(5)} = L^{(4)} \cdot W$ is equal to $L^{(4)}$, and thus $L^{(m)} = L^{(4)}$ for all $m \geq 4$.

$$\begin{aligned} L^{(1)} &= W\,, \\ L^{(2)} &= W^2 &&= W \cdot W\,, \\ L^{(4)} &= W^4 &&= W^2 \cdot W^2 \\ L^{(8)} &= W^8 &&= W^4 \cdot W^4\,, \\ &\vdots \\ L^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} &&= W^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot W^{2^{\lceil \lg(n-1) \rceil - 1}}\,. \end{aligned}$$

Since $2^{\lceil \lg(n-1) \rceil} \geq n - 1$, the final product $L^{(2^{\lceil \lg(n-1) \rceil})}$ is equal to $L^{(n-1)}$.

The following procedure computes the above sequence of matrices by using this technique of ***repeated squaring***.

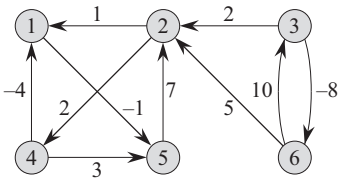



**Figure 25.2** A weighted, directed graph for use in Exercises 25.1-1, 25.2-1, and 25.3-1.

FASTER-ALL-PAIRS-SHORTEST-PATHS($W$)

```
1  n ← rows[W]
2  L^(1) ← W
3  m ← 1
4  while m < n − 1
5       do L^(2m) ← EXTEND-SHORTEST-PATHS(L^(m), L^(m))
6          m ← 2m
7  return L^(m)
```

In each iteration of the **while** loop of lines 4–6, we compute $L^{(2m)} = (L^{(m)})^2$, starting with $m = 1$. At the end of each iteration, we double the value of $m$. The final iteration computes $L^{(n-1)}$ by actually computing $L^{(2m)}$ for some $n - 1 \le 2m < 2n - 2$. By equation (25.3), $L^{(2m)} = L^{(n-1)}$. The next time the test in line 4 is performed, $m$ has been doubled, so now $m \ge n - 1$, the test fails, and the procedure returns the last matrix it computed.

The running time of FASTER-ALL-PAIRS-SHORTEST-PATHS is $\Theta(n^3 \lg n)$ since each of the $\lceil \lg(n-1) \rceil$ matrix products takes $\Theta(n^3)$ time. Observe that the code is tight, containing no elaborate data structures, and the constant hidden in the $\Theta$-notation is therefore small.

## Exercises

***25.1-1***

Run SLOW-ALL-PAIRS-SHORTEST-PATHS on the weighted, directed graph of Figure 25.2, showing the matrices that result for each iteration of the loop. Then do the same for FASTER-ALL-PAIRS-SHORTEST-PATHS.

***25.1-2***

Why do we require that $w_{ii} = 0$ for all $1 \le i \le n$?

***25.1-3***

What does the matrix

$$L^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \cdots & \infty \\ \infty & 0 & \infty & \cdots & \infty \\ \infty & \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \cdots & 0 \end{pmatrix}$$

used in the shortest-paths algorithms correspond to in regular matrix multiplication?

***25.1-4***
Show that matrix multiplication defined by EXTEND-SHORTEST-PATHS is associative.

***25.1-5***
Show how to express the single-source shortest-paths problem as a product of matrices and a vector. Describe how evaluating this product corresponds to a Bellman-Ford-like algorithm (see Section 24.1).

***25.1-6***
Suppose we also wish to compute the vertices on shortest paths in the algorithms of this section. Show how to compute the predecessor matrix $\Pi$ from the completed matrix $L$ of shortest-path weights in $O(n^3)$ time.

***25.1-7***
The vertices on shortest paths can also be computed at the same time as the shortest-path weights. Let us define $\pi_{ij}^{(m)}$ to be the predecessor of vertex $j$ on any minimum-weight path from $i$ to $j$ that contains at most $m$ edges. Modify EXTEND-SHORTEST-PATHS and SLOW-ALL-PAIRS-SHORTEST-PATHS to compute the matrices $\Pi^{(1)}, \Pi^{(2)}, \ldots, \Pi^{(n-1)}$ as the matrices $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$ are computed.

***25.1-8***
The FASTER-ALL-PAIRS-SHORTEST-PATHS procedure, as written, requires us to store $\lceil \lg(n-1) \rceil$ matrices, each with $n^2$ elements, for a total space requirement of $\Theta(n^2 \lg n)$. Modify the procedure to require only $\Theta(n^2)$ space by using only two $n \times n$ matrices.

***25.1-9***
Modify FASTER-ALL-PAIRS-SHORTEST-PATHS so that it can detect the presence of a negative-weight cycle.

***25.1-10***
Give an efficient algorithm to find the length (number of edges) of a minimum-length negative-weight cycle in a graph.

## 25.2 The Floyd-Warshall algorithm

In this section, we shall use a different dynamic-programming formulation to solve the all-pairs shortest-paths problem on a directed graph $G = (V, E)$. The resulting algorithm, known as the ***Floyd-Warshall algorithm***, runs in $\Theta(V^3)$ time. As before, negative-weight edges may be present, but we assume that there are no negative-weight cycles. As in Section 25.1, we shall follow the dynamic-programming process to develop the algorithm. After studying the resulting algorithm, we shall present a similar method for finding the transitive closure of a directed graph.

### The structure of a shortest path

In the Floyd-Warshall algorithm, we use a different characterization of the structure of a shortest path than we used in the matrix-multiplication-based all-pairs algorithms. The algorithm considers the "intermediate" vertices of a shortest path, where an ***intermediate*** vertex of a simple path $p = \langle v_1, v_2, \ldots, v_l \rangle$ is any vertex of $p$ other than $v_1$ or $v_l$, that is, any vertex in the set $\{v_2, v_3, \ldots, v_{l-1}\}$.

The Floyd-Warshall algorithm is based on the following observation. Under our assumption that the vertices of $G$ are $V = \{1, 2, \ldots, n\}$, let us consider a subset $\{1, 2, \ldots, k\}$ of vertices for some $k$. For any pair of vertices $i, j \in V$, consider all paths from $i$ to $j$ whose intermediate vertices are all drawn from $\{1, 2, \ldots, k\}$, and let $p$ be a minimum-weight path from among them. (Path $p$ is simple.) The Floyd-Warshall algorithm exploits a relationship between path $p$ and shortest paths from $i$ to $j$ with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The relationship depends on whether or not $k$ is an intermediate vertex of path $p$.

- If $k$ is not an intermediate vertex of path $p$, then all intermediate vertices of path $p$ are in the set $\{1, 2, \ldots, k-1\}$. Thus, a shortest path from vertex $i$ to vertex $j$ with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$ is also a shortest path from $i$ to $j$ with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.
- If $k$ is an intermediate vertex of path $p$, then we break $p$ down into $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$ as shown in Figure 25.3. By Lemma 24.1, $p_1$ is a shortest path from $i$ to $k$ with all intermediate vertices in the set $\{1, 2, \ldots, k\}$. Because vertex $k$ is not an intermediate vertex of path $p_1$, we see that $p_1$ is a shortest path from $i$ to $k$ with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Similarly, $p_2$ is a shortest path from vertex $k$ to vertex $j$ with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$.

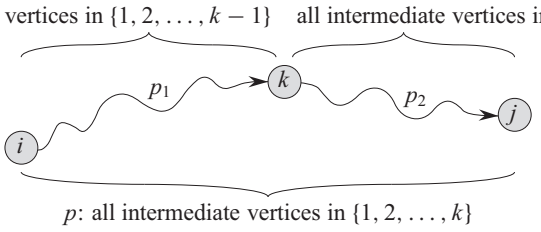



**Figure 25.3** Path $p$ is a shortest path from vertex $i$ to vertex $j$, and $k$ is the highest-numbered intermediate vertex of $p$. Path $p_1$, the portion of path $p$ from vertex $i$ to vertex $k$, has all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The same holds for path $p_2$ from vertex $k$ to vertex $j$.

### A recursive solution to the all-pairs shortest-paths problem

Based on the above observations, we define a recursive formulation of shortest-path estimates that is different from the one in Section 25.1. Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex $i$ to vertex $j$ for which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$. When $k = 0$, a path from vertex $i$ to vertex $j$ with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. Such a path has at most one edge, and hence $d_{ij}^{(0)} = w_{ij}$. A recursive definition following the above discussion is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1\,. \end{cases} \tag{25.5}$$

Because for any path, all intermediate vertices are in the set $\{1, 2, \ldots, n\}$, the matrix $D^{(n)} = \left(d_{ij}^{(n)}\right)$ gives the final answer: $d_{ij}^{(n)} = \delta(i, j)$ for all $i, j \in V$.

### Computing the shortest-path weights bottom up

Based on recurrence (25.5), the following bottom-up procedure can be used to compute the values $d_{ij}^{(k)}$ in order of increasing values of $k$. Its input is an $n \times n$ matrix $W$ defined as in equation (25.1). The procedure returns the matrix $D^{(n)}$ of shortest-path weights.

```
FLOYD-WARSHALL(W)
1  n ← rows[W]
2  D^(0) ← W
3  for k ← 1 to n
4       do for i ← 1 to n
5            do for j ← 1 to n
6                 do d_ij^(k) ← min(d_ij^(k−1), d_ik^(k−1) + d_kj^(k−1))
7  return D^(n)
```

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

**Figure 25.4** The sequence of matrices $D^{(k)}$ and $\Pi^{(k)}$ computed by the Floyd-Warshall algorithm for the graph in Figure 25.1.

Figure 25.4 shows the matrices $D^{(k)}$ computed by the Floyd-Warshall algorithm for the graph in Figure 25.1.

The running time of the Floyd-Warshall algorithm is determined by the triply nested **for** loops of lines 3–6. Because each execution of line 6 takes $O(1)$ time, the algorithm runs in time $\Theta(n^3)$. As in the final algorithm in Section 25.1, the

code is tight, with no elaborate data structures, and so the constant hidden in the $\Theta$-notation is small. Thus, the Floyd-Warshall algorithm is quite practical for even moderate-sized input graphs.

### Constructing a shortest path

There are a variety of different methods for constructing shortest paths in the Floyd-Warshall algorithm. One way is to compute the matrix $D$ of shortest-path weights and then construct the predecessor matrix $\Pi$ from the $D$ matrix. This method can be implemented to run in $O(n^3)$ time (Exercise 25.1-6). Given the predecessor matrix $\Pi$, the PRINT-ALL-PAIRS-SHORTEST-PATH procedure can be used to print the vertices on a given shortest path.

We can compute the predecessor matrix $\Pi$ "on-line" just as the Floyd-Warshall algorithm computes the matrices $D^{(k)}$. Specifically, we compute a sequence of matrices $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$, where $\Pi = \Pi^{(n)}$ and $\pi_{ij}^{(k)}$ is defined to be the predecessor of vertex $j$ on a shortest path from vertex $i$ with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.

We can give a recursive formulation of $\pi_{ij}^{(k)}$. When $k = 0$, a shortest path from $i$ to $j$ has no intermediate vertices at all. Thus,

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty\,, \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty\,. \end{cases} \tag{25.6}$$

For $k \geq 1$, if we take the path $i \rightsquigarrow k \rightsquigarrow j$, where $k \neq j$, then the predecessor of $j$ we choose is the same as the predecessor of $j$ we chose on a shortest path from $k$ with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Otherwise, we choose the same predecessor of $j$ that we chose on a shortest path from $i$ with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Formally, for $k \geq 1$,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\,, \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\,. \end{cases} \tag{25.7}$$

We leave the incorporation of the $\Pi^{(k)}$ matrix computations into the FLOYD-WARSHALL procedure as Exercise 25.2-3. Figure 25.4 shows the sequence of $\Pi^{(k)}$ matrices that the resulting algorithm computes for the graph of Figure 25.1. The exercise also asks for the more difficult task of proving that the predecessor subgraph $G_{\pi,i}$ is a shortest-paths tree with root $i$. Yet another way to reconstruct shortest paths is given as Exercise 25.2-7.

### Transitive closure of a directed graph

Given a directed graph $G = (V, E)$ with vertex set $V = \{1, 2, \ldots, n\}$, we may wish to find out whether there is a path in $G$ from $i$ to $j$ for all vertex pairs $i, j \in V$. The ***transitive closure*** of $G$ is defined as the graph $G^* = (V, E^*)$, where

$E^* = \{(i, j) : \text{there is a path from vertex } i \text{ to vertex } j \text{ in } G\}$ .

One way to compute the transitive closure of a graph in $\Theta(n^3)$ time is to assign a weight of 1 to each edge of $E$ and run the Floyd-Warshall algorithm. If there is a path from vertex $i$ to vertex $j$, we get $d_{ij} < n$. Otherwise, we get $d_{ij} = \infty$.

There is another, similar way to compute the transitive closure of $G$ in $\Theta(n^3)$ time that can save time and space in practice. This method involves substitution of the logical operations $\vee$ (logical OR) and $\wedge$ (logical AND) for the arithmetic operations min and $+$ in the Floyd-Warshall algorithm. For $i, j, k = 1, 2, \ldots, n$, we define $t_{ij}^{(k)}$ to be 1 if there exists a path in graph $G$ from vertex $i$ to vertex $j$ with all intermediate vertices in the set $\{1, 2, \ldots, k\}$, and 0 otherwise. We construct the transitive closure $G^* = (V, E^*)$ by putting edge $(i, j)$ into $E^*$ if and only if $t_{ij}^{(n)} = 1$. A recursive definition of $t_{ij}^{(k)}$, analogous to recurrence (25.5), is

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i, j) \notin E \ , \\ 1 & \text{if } i = j \text{ or } (i, j) \in E \ , \end{cases}$$

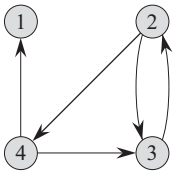
and for $k \geq 1$,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right) \ . \tag{25.8}$$

As in the Floyd-Warshall algorithm, we compute the matrices $T^{(k)} = \left(t_{ij}^{(k)}\right)$ in order of increasing $k$.

```
TRANSITIVE-CLOSURE(G)
 1  n ← |V[G]|
 2  for i ← 1 to n
 3       do for j ← 1 to n
 4            do if i = j or (i, j) ∈ E[G]
 5                 then t_ij^(0) ← 1
 6                 else t_ij^(0) ← 0
 7  for k ← 1 to n
 8       do for i ← 1 to n
 9            do for j ← 1 to n
10                 do t_ij^(k) ← t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
11  return T^(n)
```

Figure 25.5 shows the matrices $T^{(k)}$ computed by the TRANSITIVE-CLOSURE procedure on a sample graph. The TRANSITIVE-CLOSURE procedure, like the Floyd-Warshall algorithm, runs in $\Theta(n^3)$ time. On some computers, though, logical operations on single-bit values execute faster than arithmetic operations on integer words of data. Moreover, because the direct transitive-closure algorithm uses only boolean values rather than integer values, its space requirement is less than the




$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

**Figure 25.5** A directed graph and the matrices $T^{(k)}$ computed by the transitive-closure algorithm.

Floyd-Warshall algorithm's by a factor corresponding to the size of a word of computer storage.

## Exercises

***25.2-1***
Run the Floyd-Warshall algorithm on the weighted, directed graph of Figure 25.2. Show the matrix $D^{(k)}$ that results for each iteration of the outer loop.

***25.2-2***
Show how to compute the transitive closure using the technique of Section 25.1.

***25.2-3***
Modify the FLOYD-WARSHALL procedure to include computation of the $\Pi^{(k)}$ matrices according to equations (25.6) and (25.7). Prove rigorously that for all $i \in V$, the predecessor subgraph $G_{\pi,i}$ is a shortest-paths tree with root $i$. (*Hint:* To show that $G_{\pi,i}$ is acyclic, first show that $\pi_{ij}^{(k)} = l$ implies $d_{ij}^{(k)} \geq d_{il}^{(k)} + w_{lj}$, according to the definition of $\pi_{ij}^{(k)}$. Then, adapt the proof of Lemma 24.16.)

***25.2-4***
As it appears above, the Floyd-Warshall algorithm requires $\Theta(n^3)$ space, since we compute $d_{ij}^{(k)}$ for $i, j, k = 1, 2, \ldots, n$. Show that the following procedure, which simply drops all the superscripts, is correct, and thus only $\Theta(n^2)$ space is required.

```
FLOYD-WARSHALL′(W)
1  n ← rows[W]
2  D ← W
3  for k ← 1 to n
4       do for i ← 1 to n
5            do for j ← 1 to n
6                 do d_ij ← min (d_ij , d_ik + d_kj)
7  return D
```

***25.2-5***

Suppose that we modify the way in which equality is handled in equation (25.7):

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)} , \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \geq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} . \end{cases}$$

Is this alternative definition of the predecessor matrix $\Pi$ correct?

***25.2-6***

How can the output of the Floyd-Warshall algorithm be used to detect the presence of a negative-weight cycle?

***25.2-7***

Another way to reconstruct shortest paths in the Floyd-Warshall algorithm uses values $\phi_{ij}^{(k)}$ for $i, j, k = 1, 2, \ldots, n$, where $\phi_{ij}^{(k)}$ is the highest-numbered intermediate vertex of a shortest path from $i$ to $j$ in which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$. Give a recursive formulation for $\phi_{ij}^{(k)}$, modify the FLOYD-WARSHALL procedure to compute the $\phi_{ij}^{(k)}$ values, and rewrite the PRINT-ALL-PAIRS-SHORTEST-PATH procedure to take the matrix $\Phi = \left(\phi_{ij}^{(n)}\right)$ as an input. How is the matrix $\Phi$ like the $s$ table in the matrix-chain multiplication problem of Section 15.2?

***25.2-8***

Give an $O(VE)$-time algorithm for computing the transitive closure of a directed graph $G = (V, E)$.

***25.2-9***

Suppose that the transitive closure of a directed acyclic graph can be computed in $f(|V|, |E|)$ time, where $f$ is a monotonically increasing function of $|V|$ and $|E|$. Show that the time to compute the transitive closure $G^* = (V, E^*)$ of a general directed graph $G = (V, E)$ is $f(|V|, |E|) + O(V + E^*)$.

## 25.3 Johnson's algorithm for sparse graphs

Johnson's algorithm finds shortest paths between all pairs in $O(V^2 \lg V + VE)$ time. For sparse graphs, it is asymptotically better than either repeated squaring of matrices or the Floyd-Warshall algorithm. The algorithm either returns a matrix of shortest-path weights for all pairs of vertices or reports that the input graph contains a negative-weight cycle. Johnson's algorithm uses as subroutines both Dijkstra's algorithm and the Bellman-Ford algorithm, which are described in Chapter 24.

Johnson's algorithm uses the technique of ***reweighting***, which works as follows. If all edge weights $w$ in a graph $G = (V, E)$ are nonnegative, we can find shortest paths between all pairs of vertices by running Dijkstra's algorithm once from each vertex; with the Fibonacci-heap min-priority queue, the running time of this all-pairs algorithm is $O(V^2 \lg V + VE)$. If $G$ has negative-weight edges but no negative-weight cycles, we simply compute a new set of nonnegative edge weights that allows us to use the same method. The new set of edge weights $\widehat{w}$ must satisfy two important properties.

1. For all pairs of vertices $u, v \in V$, a path $p$ is a shortest path from $u$ to $v$ using weight function $w$ if and only if $p$ is also a shortest path from $u$ to $v$ using weight function $\widehat{w}$.
2. For all edges $(u, v)$, the new weight $\widehat{w}(u, v)$ is nonnegative.

As we shall see in a moment, the preprocessing of $G$ to determine the new weight function $\widehat{w}$ can be performed in $O(VE)$ time.

### Preserving shortest paths by reweighting

As the following lemma shows, it is easy to come up with a reweighting of the edges that satisfies the first property above. We use $\delta$ to denote shortest-path weights derived from weight function $w$ and $\widehat{\delta}$ to denote shortest-path weights derived from weight function $\widehat{w}$.

***Lemma 25.1 (Reweighting does not change shortest paths)***
Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbf{R}$, let $h : V \to \mathbf{R}$ be any function mapping vertices to real numbers. For each edge $(u, v) \in E$, define

$$\widehat{w}(u, v) = w(u, v) + h(u) - h(v) \; . \tag{25.9}$$

Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be any path from vertex $v_0$ to vertex $v_k$. Then $p$ is a shortest path from $v_0$ to $v_k$ with weight function $w$ if and only if it is a shortest path with weight function $\widehat{w}$. That is, $w(p) = \delta(v_0, v_k)$ if and only if $\widehat{w}(p) = \widehat{\delta}(v_0, v_k)$.