Part VI Graph Algorithms

Introduction

Graph problems pervade computer science, and algorithms for working with them are fundamental to the field. Hundreds of interesting computational problems are couched in terms of graphs. This part touches on a few of the more significant ones.

Chapter 20 shows how to represent a graph in a computer and then discusses algorithms based on searching a graph using either breadth-first search or depth-first search. The chapter gives two applications of depth-first search: topologically sorting a directed acyclic graph and decomposing a directed graph into its strongly connected components.

Chapter 21 describes how to compute a minimum-weight spanning tree of a graph: 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 serve as good examples of greedy algorithms (see Chapter 15).

Chapters 22 and 23 consider how to compute shortest paths between vertices when each edge has an associated length or "weight." Chapter 22 shows how to find shortest paths from a given source vertex to all other vertices, and Chapter 23 examines methods to compute shortest paths between every pair of vertices.

Chapter 24 shows how to compute a maximum flow of material in a flow network, which is a directed graph having a specified source vertex of material, a specified sink vertex, 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 help solve a variety of related problems efficiently.

Finally, Chapter 25 explores matchings in bipartite graphs: methods for pairing up vertices that are partitioned into two sets by selecting edges that go between the sets. Bipartite-matching problems model several situations that arise in the real world. The chapter examines how to find a matching of maximum cardinality; the

"stable-marriage problem," which has the highly practical application of matching medical residents to hospitals; and assignment problems, which maximize the total weight of a bipartite matching.

When we characterize 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, we denote the size of the input with two parameters, not just one. We adopt a common notational convention for these parameters. Inside asymptotic notation (such as O-notation or Θ-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 $O(VE)$ time," meaning that the algorithm runs in $O(|V||E|)$ time. 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 $G.V$ and its edge set by $G.E$. That is, the pseudocode views vertex and edge sets as attributes of a graph.

20 Elementary Graph Algorithms

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

Section 20.1 discusses the two most common computational representations of graphs: as adjacency lists and as adjacency matrices. Section 20.2 presents a simple graph-searching algorithm called breadth-first search and shows how to create a breadth-first tree. Section 20.3 presents depth-first search and proves some standard results about the order in which depth-first search visits vertices. Section 20.4 provides our first real application of depth-first search: topologically sorting a directed acyclic graph. A second application of depth-first search, finding the strongly connected components of a directed graph, is the topic of Section 20.5.

20.1 Representations of graphs

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

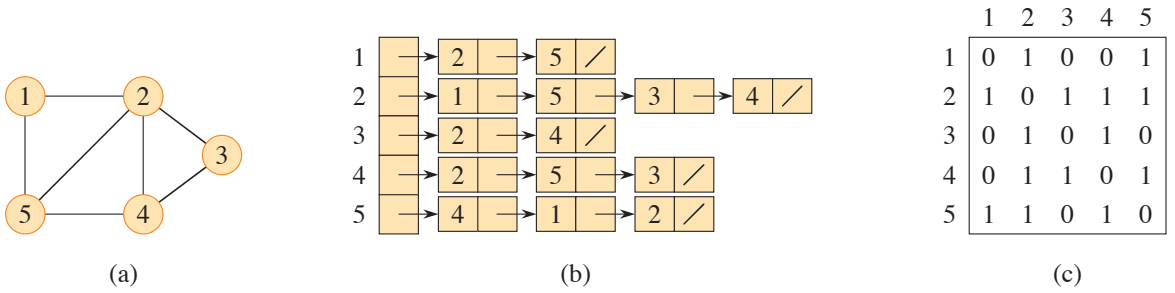


Figure 20.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

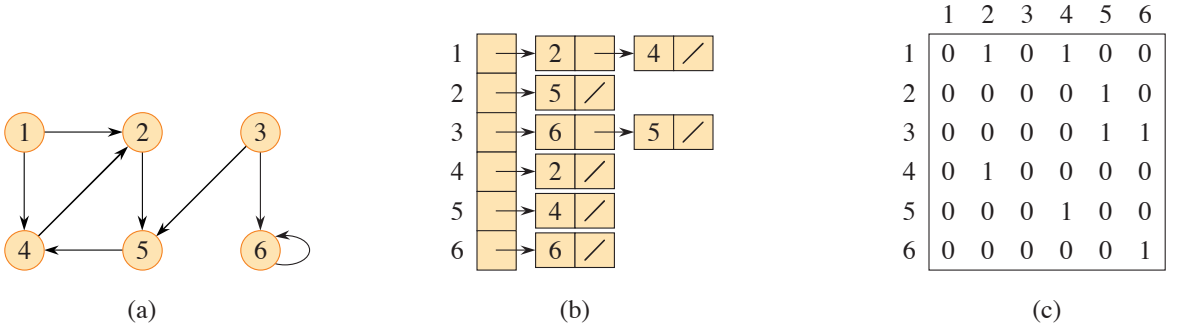


Figure 20.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

all-pairs shortest-paths algorithms presented in Chapter 23 assume that their input graphs are represented by adjacency matrices.

The ***adjacency-list representation*** of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge $(u, v) \in E$. That is, $Adj[u]$ consists of all the vertices adjacent to u in G. (Alternatively, it can contain pointers to these vertices.) Since the adjacency lists represent the edges of a graph, our pseudocode treats the array Adj as an attribute of the graph, just like the edge set E. In pseudocode, therefore, you will see notation such as $G.Adj[u]$. Figure 20.1(b) is an adjacency-list representation of the undirected graph in Figure 20.1(a). Similarly, Figure 20.2(b) is an adjacency-list representation of the directed graph in Figure 20.2(a).

If G is a directed graph, the sum of the lengths of all the adjacency lists is $|E|$, since an edge of the form (u, v) is represented by having v appear in $Adj[u]$. If G is

an undirected graph, the sum of the lengths of all the adjacency lists is $2|E|$, since if (u, v) is an undirected edge, then u appears in v's adjacency list and vice versa. For both directed and undirected graphs, the adjacency-list representation has the desirable property that the amount of memory it requires is $\Theta(V + E)$. Finding each edge in the graph also takes $\Theta(V + E)$ time, rather than just $\Theta(E)$, since each of the $|V|$ adjacency lists must be examined. Of course, if $|E| = \Omega(V)$ —such as in a connected, undirected graph or a strongly connected, directed graph—we can say that finding each edge takes $\Theta(E)$ time.

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

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

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

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

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

Observe the symmetry along the main diagonal of the adjacency matrix in Figure 20.1(c). Since in an undirected graph, (u, v) and (v, u) represent the same edge, the adjacency matrix A of an undirected graph is its own transpose: $A = A^{\mathrm{T}}$. In some applications, it pays to store only the entries on and above the diagonal of the adjacency matrix, thereby cutting the memory needed to store the graph almost in half.

Like the adjacency-list representation of a graph, an adjacency matrix can represent a weighted graph. For example, if $G = (V, E)$ is a weighted graph with edge-weight function w, you can store the weight $w(u, v)$ of the edge $(u, v) \in E$

as the entry in row u and column v of the adjacency matrix. If an edge does not exist, you can store a NIL value as its corresponding matrix entry, though for many problems it is convenient to use a value such as 0 or ∞.

Although the adjacency-list representation is asymptotically at least as space-efficient as the adjacency-matrix representation, adjacency matrices are simpler, and so you might prefer them when graphs are reasonably small. Moreover, adjacency matrices carry a further advantage for unweighted graphs: they require only one bit per entry.

Representing attributes

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

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

Exercises

20.1-1

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

20.1-2

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

20.1-3

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

20.1-4

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

20.1-5

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

20.1-6

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

20.1-7

The ***incidence matrix*** of a directed graph $G = (V, E)$ with no self-loops is a $|V| \times |E|$ matrix $B = (b_{ij})$ such that

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

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

20.1-8

Suppose that instead of a linked list, each array entry $Adj[u]$ is a hash table containing the vertices v for which $(u, v) \in E$, with collisions resolved by chaining. Under the assumption of uniform independent hashing, if all edge lookups are equally likely, what is the expected time to determine whether an edge is in the graph?

What disadvantages does this scheme have? Suggest an alternate data structure for each edge list that solves these problems. Does your alternative have disadvantages compared with the hash table?

20.2 Breadth-first search

Breadth-first search is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms. Prim's minimum-spanning-tree algorithm (Section 21.2) and Dijkstra's single-source shortest-paths algorithm (Section 22.3) use ideas similar to those in breadth-first search.

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

Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. You can think of it as discovering vertices in waves emanating from the source vertex. That is, starting from s, the algorithm first discovers all neighbors of s, which have distance 1. Then it discovers all vertices with distance 2, then all vertices with distance 3, and so on, until it has discovered every vertex reachable from s.

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

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

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

Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex s. Whenever the search discovers a white vertex v in the course of scanning the adjacency list of a gray vertex u, the vertex v and the edge (u, v) are added to the tree. We say that u is the ***predecessor*** or ***parent*** of v in the breadth-first tree. Since every vertex reachable from s is discovered at most once, each vertex reachable from s has exactly one parent. (There is one exception: because s is the root of the breadth-first tree, it has no parent.) Ancestor and descendant relationships in the breadth-first tree are defined relative to the root s as usual: if u is on the simple path in the tree from the root s to vertex v, then u is an ancestor of v and v is a descendant of u.

The breadth-first-search procedure BFS on the following page assumes that the graph $G = (V, E)$ is represented using adjacency lists. It denotes the queue by Q, and it attaches three additional attributes to each vertex v in the graph:

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

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

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

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

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

Although we won't use this loop invariant to prove correctness, it is easy to see that it holds prior to the first iteration and that each iteration of the loop maintains the invariant. Prior to the first iteration, the only gray vertex, and the only vertex

---

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

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

in Q, is the source vertex s. Line 11 determines the gray vertex u at the head of the queue Q and removes it from Q. The **for** loop of lines 12–17 considers each vertex v in the adjacency list of u. If v is white, then it has not yet been discovered, and the procedure discovers it by executing lines 14–17. These lines paint vertex v gray, set v's distance $v.d$ to $u.d + 1$, record u as v's parent $v.\pi$, and place v at the tail of the queue Q. Once the procedure has examined all the vertices on u's adjacency list, it blackens u in line 18, indicating that u is now behind the frontier. The loop invariant is maintained because whenever a vertex is painted gray (in line 14) it is also enqueued (in line 17), and whenever a vertex is dequeued (in line 11) it is also painted black (in line 18).

The results of breadth-first search may depend upon the order in which the neighbors of a given vertex are visited in line 12: the breadth-first tree may vary, but the distances d computed by the algorithm do not. (See Exercise 20.2-5.)

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

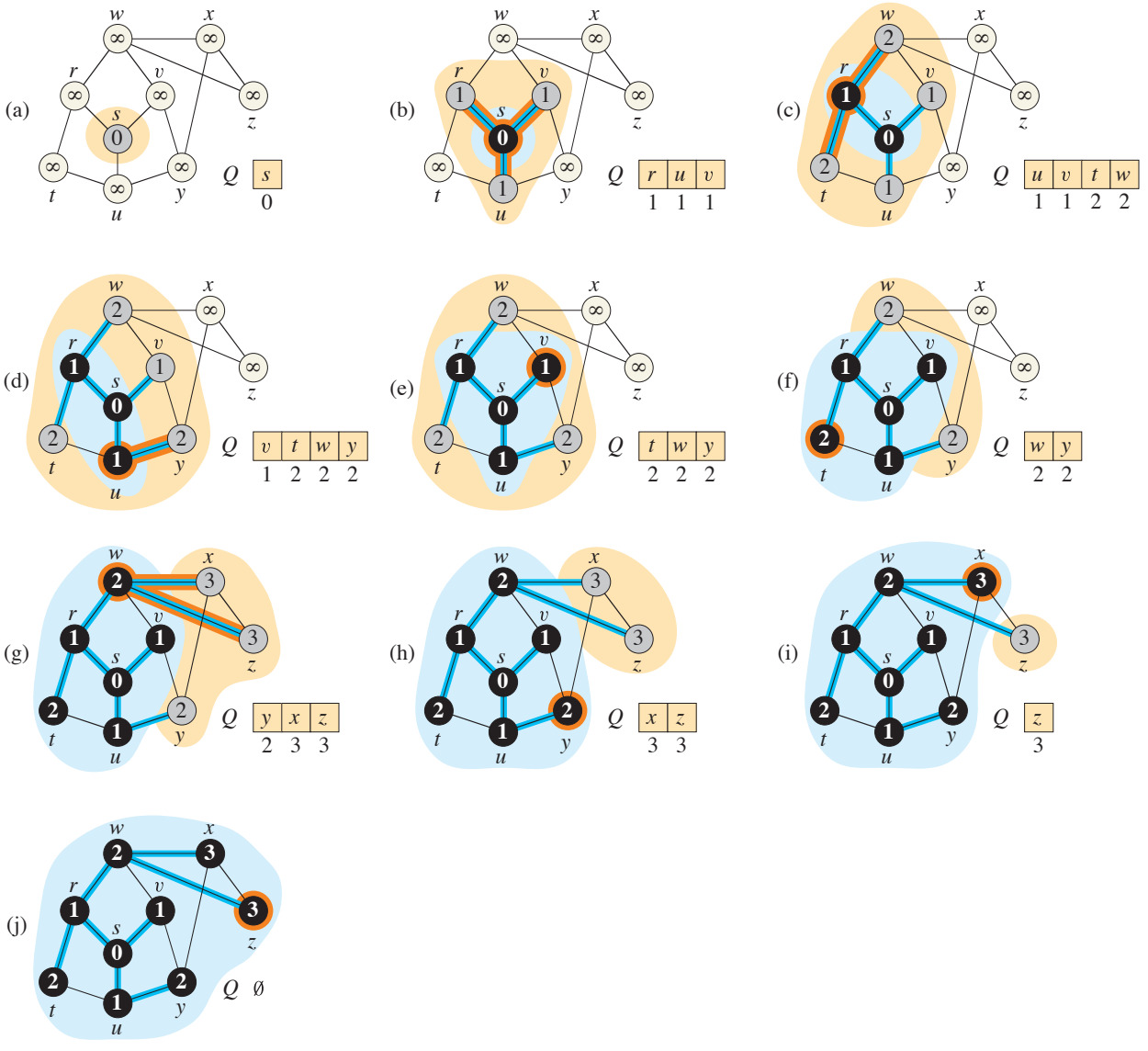


Figure 20.3 The operation of BFS on an undirected graph. Each part shows the graph and the queue Q at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances appear within each vertex and below vertices in the queue. The tan region surrounds the frontier of the search,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

Analysis

Before proving the various properties of breadth-first search, let's take on the easier job of analyzing its running time on an input graph $G = (V, E)$. We use aggregate analysis, as we saw in Section 16.1. After initialization, breadth-first search never whitens a vertex, and thus the test in line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once. The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(V)$. Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all $|V|$ adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(V + E)$. The overhead for initialization is $O(V)$, and thus the total running time of the BFS procedure is $O(V + E)$. Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G.

Shortest paths

Now, let's see why breadth-first search finds the shortest distance from a given source vertex s to each vertex in a graph. Define the ***shortest-path distance*** $\delta(s, v)$ from s to v as the minimum number of edges in any path from vertex s to vertex v. If there is no path from s to v, then $\delta(s, v) = \infty$. We call a path of length $\delta(s, v)$ from s to v a ***shortest path***[2] from s to v. Before showing that breadth-first search correctly computes shortest-path distances, we investigate an important property of shortest-path distances.

Lemma 20.1
Let $G = (V, E)$ be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$,

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

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

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

---

[2] Chapters 22 and 23 generalize shortest paths to weighted graphs, in which every edge has a real-valued weight and the weight of a path is the sum of the weights of its constituent edges. The graphs considered in the present chapter are unweighted or, equivalently, all edges have unit weight.

Lemma 20.2
Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then, for each vertex $v \in V$, the value $v.d$ computed by BFS satisfies $v.d \geq \delta(s, v)$ at all times, including at termination.

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

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

For the inductive step, consider a white vertex v that is discovered during the search from a vertex u. The inductive hypothesis implies that $u.d \geq \delta(s, u)$. The assignment performed by line 15 and Lemma 20.1 give

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

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

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

Lemma 20.3
Suppose that during the execution of BFS on a graph $G = (V, E)$, the queue Q contains the vertices $\langle v_1, v_2, \ldots, v_r \rangle$, where v_1 is the head of Q and v_r is the tail. Then, $v_r.d \leq v_1.d + 1$ and $v_i.d \leq v_{i+1}.d$ for $i = 1, 2, \ldots, r - 1$.

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

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

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

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

Corollary 20.4
Suppose that vertices v_i and v_j are enqueued during the execution of BFS, and that v_i is enqueued before v_j. Then $v_i.d \leq v_j.d$ at the time that v_j is enqueued.

Proof Immediate from Lemma 20.3 and the property that each vertex receives a finite d value at most once during the course of BFS. ■

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

Theorem 20.5 (Correctness of breadth-first search)
Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source s, and upon termination, $v.d = \delta(s, v)$ for all $v \in V$. Moreover, for any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $v.\pi$ followed by the edge $(v.\pi, v)$.

Proof Assume for the purpose of contradiction that some vertex receives a d value not equal to its shortest-path distance. Of all such vertices, let v be a vertex that has the minimum $\delta(s, v)$. By Lemma 20.2, we have $v.d \geq \delta(s, v)$, and thus $v.d > \delta(s, v)$. We cannot have $v = s$, because $s.d = 0$ and $\delta(s, s) = 0$. Vertex v must be reachable from s, for otherwise we would have $\delta(s, v) = \infty \geq v.d$. Let u be the vertex immediately preceding v on some shortest path from s to v (since $v \neq s$, vertex u must exist), so that $\delta(s, v) = \delta(s, u) + 1$. Because $\delta(s, u) < \delta(s, v)$,

and because of how we chose v, we have $u.d = \delta(s,u)$. Putting these properties together gives

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

Now consider the time when BFS chooses to dequeue vertex u from Q in line 11. At this time, vertex v is either white, gray, or black. We shall show that each of these cases leads to a contradiction of inequality (20.1). If v is white, then line 15 sets $v.d = u.d + 1$, contradicting inequality (20.1). If v is black, then it was already removed from the queue and, by Corollary 20.4, we have $v.d \le u.d$, again contradicting inequality (20.1). If v is gray, then it was painted gray upon dequeuing some vertex w, which was removed from Q earlier than u and for which $v.d = w.d + 1$. By Corollary 20.4, however, $w.d \le u.d$, and so $v.d = w.d + 1 \le u.d + 1$, once again contradicting inequality (20.1).

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

Breadth-first trees

The blue edges in Figure 20.3 show the breadth-first tree built by the BFS procedure as it searches the graph. The tree corresponds to the π attributes. More formally, for a graph $G = (V,E)$ with source s, we define the ***predecessor subgraph*** of G as $G_\pi = (V_\pi, E_\pi)$, where

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

and

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

The predecessor subgraph G_π is a ***breadth-first tree*** if V_π consists of the vertices reachable from s and, for all $v \in V_\pi$, the subgraph G_π contains a unique simple path from s to v that is also a shortest path from s to v in G. A breadth-first tree is in fact a tree, since it is connected and $|E_\pi| = |V_\pi| - 1$ (see Theorem B.2 on page 1169). We call the edges in E_π ***tree edges***.

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

Lemma 20.6

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

Proof Line 16 of BFS sets $v.\pi = u$ if and only if $(u, v) \in E$ and $\delta(s, v) < \infty$—that is, if v is reachable from s—and thus V_π consists of the vertices in V reachable from s. Since the predecessor subgraph G_π forms a tree, by Theorem B.2, it contains a unique simple path from s to each vertex in V_π. Applying Theorem 20.5 inductively yields that every such path is a shortest path in G. ■

The PRINT-PATH procedure prints out the vertices on a shortest path from s to v, assuming that BFS has already computed a breadth-first tree. This procedure runs in time linear in the number of vertices in the path printed, since each recursive call is for a path one vertex shorter.

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

Exercises

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

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

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

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

20.2-5

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

20.2-6

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

20.2-7

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

★ ***20.2-8***

The ***diameter*** of a tree $T = (V, E)$ is defined as $\max\{\delta(u, v) : u, v \in V\}$, that is, the largest of all shortest-path distances in the tree. Give an efficient algorithm to compute the diameter of a tree, and analyze the running time of your algorithm.

20.3 Depth-first search

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

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

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

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

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

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

Besides creating a depth-first forest, depth-first search also ***timestamps*** each vertex. Each vertex v has two timestamps: the first timestamp $v.d$ records when v is first discovered (and grayed), and the second timestamp $v.f$ records when the search finishes examining v's adjacency list (and blackens v). These timestamps provide important information about the structure of the graph and are generally helpful in reasoning about the behavior of depth-first search.

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

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

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

---

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

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

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

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

The DFS procedure works as follows. Lines 1–3 paint all vertices white and initialize their π attributes to NIL. Line 4 resets the global time counter. Lines 5–7 check each vertex in V in turn and, when a white vertex is found, visit it by calling DFS-VISIT. Upon every call of DFS-VISIT(G, u) in line 7, vertex u becomes the root of a new tree in the depth-first forest. When DFS returns, every vertex u has been assigned a ***discovery time*** $u.d$ and a ***finish time*** $u.f$.

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

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

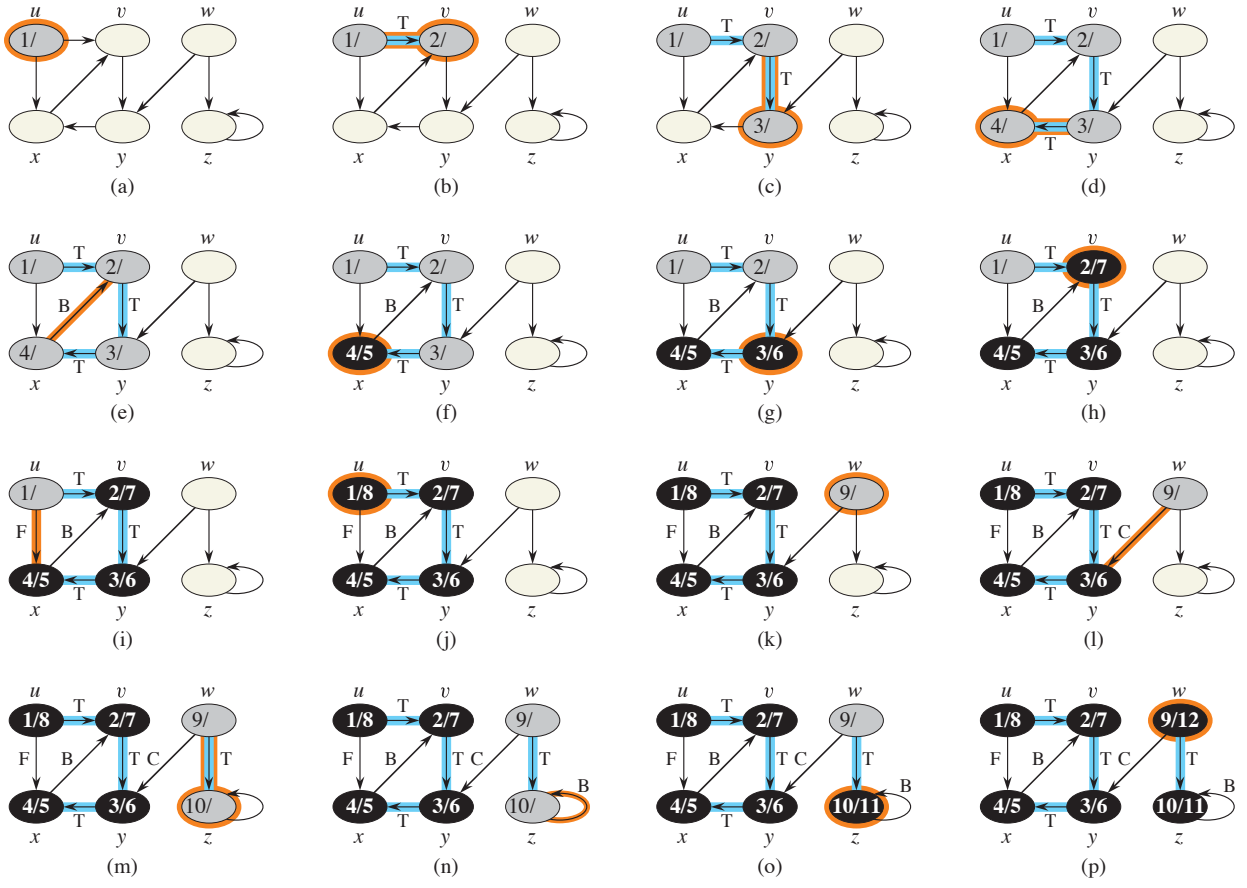


Figure 20.4 The progress of the depth-first-search algorithm DFS on a directed graph. Edges are classified as they are explored: tree edges are labeled T, back edges B, forward edges F, and cross edges C. Timestamps within vertices indicate discovery time/finish times. Tree edges are highlighted in blue.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

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

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

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

Properties of depth-first search

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

Another important property of depth-first search is that discovery and finish times have ***parenthesis structure***. If the DFS-VISIT procedure were to print a left parenthesis "(u" when it discovers vertex u and to print a right parenthesis "u)" when it finishes u, then the printed expression would be well formed in the sense that the parentheses are properly nested. For example, the depth-first search of Figure 20.5(a) corresponds to the parenthesization shown in Figure 20.5(b). The following theorem provides another way to characterize the parenthesis structure.

Theorem 20.7 (Parenthesis theorem)
In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

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

Proof We begin with the case in which $u.d < v.d$. We consider two subcases, according to whether $v.d < u.f$. The first subcase occurs when $v.d < u.f$, so that v was discovered while u was still gray, which implies that v is a descendant of u. Moreover, since v was discovered after u, all of its outgoing edges are explored, and v is finished, before the search returns to and finishes u. In this case, therefore, the interval $[v.d, v.f]$ is entirely contained within the interval $[u.d, u.f]$. In the other subcase, $u.f < v.\ d$, and by inequality (20.4), $u.d < u.\ f < v.\ d < v.\ f$, and thus the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint. Because the intervals are disjoint, neither vertex was discovered while the other was gray, and so neither vertex is a descendant of the other.

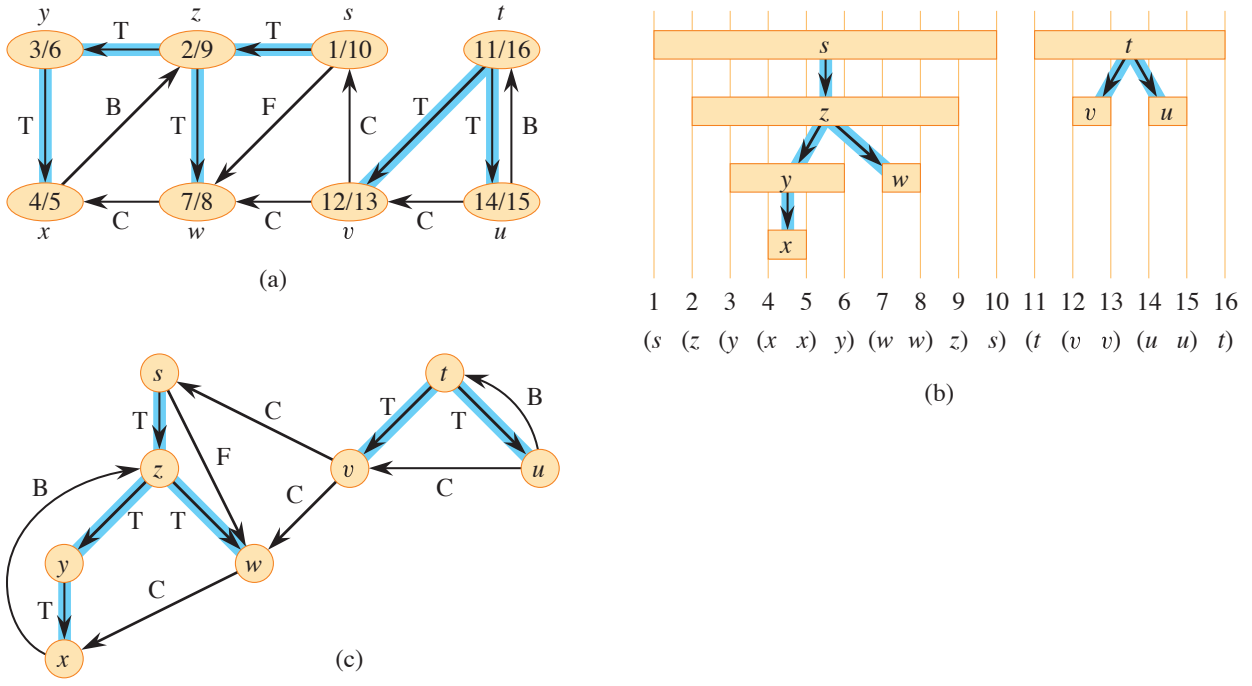


Figure 20.5 Properties of depth-first search. **(a)** The result of a depth-first search of a directed graph. Vertices are timestamped and edge types are indicated as in Figure 20.4. **(b)** Intervals for the discovery time and finish time of each vertex correspond to the parenthesization shown. Each rectangle spans the interval given by the discovery and finish times of the corresponding vertex. Only tree edges are shown. If two intervals overlap, then one is nested within the other, and the vertex corresponding to the smaller interval is a descendant of the vertex corresponding to the larger. **(c)** The graph of part (a) redrawn with all tree and forward edges going down within a depth-first tree and all back edges going up from a descendant to an ancestor.

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

Corollary 20.8 (Nesting of descendants' intervals)
Vertex v is a proper descendant of vertex u in the depth-first forest for a (directed or undirected) graph G if and only if $u.d < v.d < v.f < u.f$.

Proof Immediate from Theorem 20.7. ■

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

Theorem 20.9 (White-path theorem)
In a depth-first forest of a (directed or undirected) graph $G = (V, E)$, vertex v is a descendant of vertex u if and only if at the time $u.d$ that the search discovers u, there is a path from u to v consisting entirely of white vertices.

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

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

Classification of edges

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

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

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

In Figures 20.4 and 20.5, edge labels indicate edge types. Figure 20.5(c) also shows how to redraw the graph of Figure 20.5(a) so that all tree and forward edges head downward in a depth-first tree and all back edges go up. You can redraw any graph in this fashion.

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

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

The first case is immediate from the specification of the algorithm. For the second case, observe that the gray vertices always form a linear chain of descendants corresponding to the stack of active DFS-VISIT invocations. The number of gray vertices is 1 more than the depth in the depth-first forest of the vertex most recently discovered. Depth-first search always explores from the deepest gray vertex, so that an edge that reaches another gray vertex has reached an ancestor. The third case handles the remaining possibility. Exercise 20.3-5 asks you to show that such an edge (u, v) is a forward edge if $u.d < v.d$ and a cross edge if $u.d > v.d$.

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

Theorem 20.10
In a depth-first search of an undirected graph G, every edge of G is either a tree edge or a back edge.

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

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

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

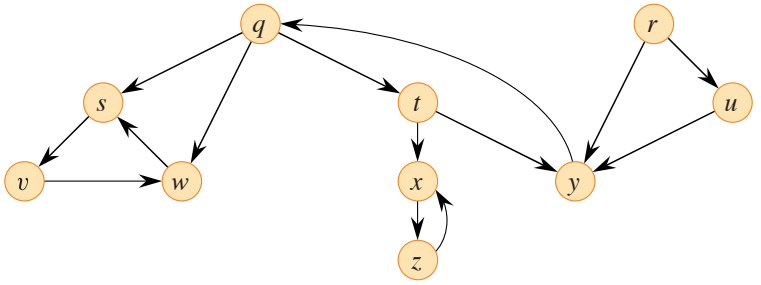


Figure 20.6 A directed graph for use in Exercises 20.3-2 and 20.5-2.

Exercises

20.3-1

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

20.3-2

Show how depth-first search works on the graph of Figure 20.6. Assume that the **for** loop of lines 5–7 of the DFS procedure considers the vertices in alphabetical order, and assume that each adjacency list is ordered alphabetically. Show the discovery and finish times for each vertex, and show the classification of each edge.

20.3-3

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

20.3-4

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

20.3-5

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

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

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

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

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

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

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

20.3-9
Modify the pseudocode for depth-first search so that it prints out every edge in the directed graph G, together with its type. Show what modifications, if any, you need to make if G is undirected.

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

20.3-11
Let $G = (V, E)$ be a connected, undirected graph. Give an $O(V + E)$-time algorithm to compute a path in G that traverses each edge in E exactly once in each direction. Describe how you can find your way out of a maze if you are given a large supply of pennies.

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

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

20.4 Topological sort

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

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

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

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

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

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

Lemma 20.11

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

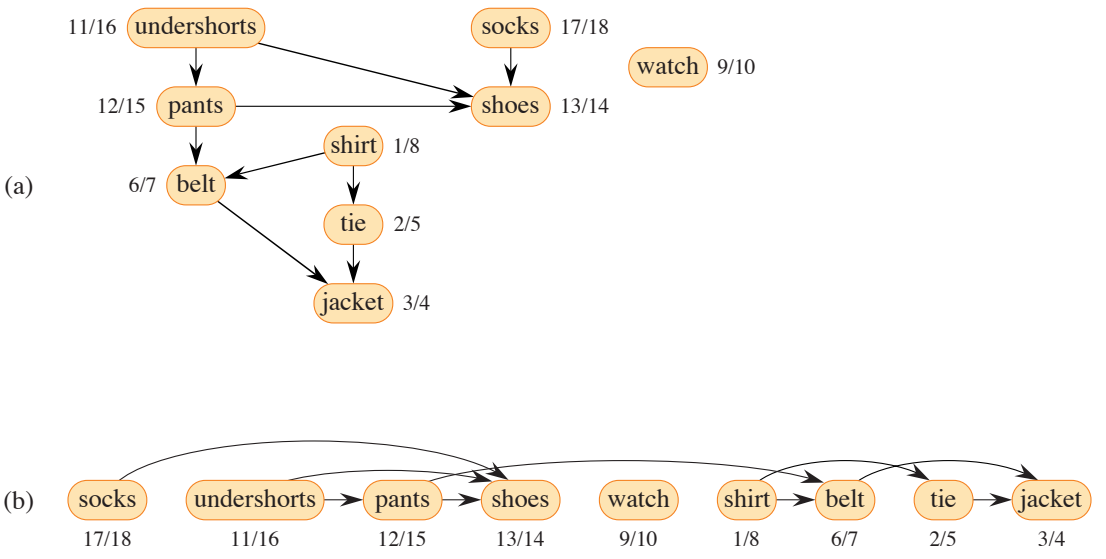


Figure 20.7 **(a)** Professor Bumstead topologically sorts his clothing when getting dressed. Each directed edge (u, v) means that garment u must be put on before garment v. The discovery and finish times from a depth-first search are shown next to each vertex. **(b)** The same graph shown topologically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

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

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

Theorem 20.12

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

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

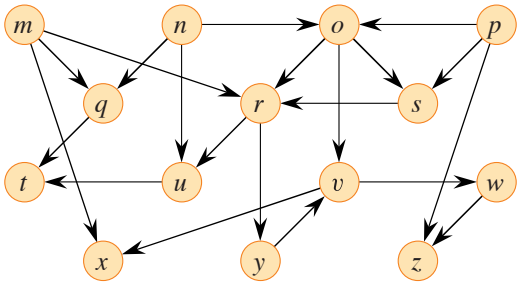


Figure 20.8 A dag for topological sorting.

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

Exercises

20.4-1

Show the ordering of vertices produced by TOPOLOGICAL-SORT when it is run on the dag of Figure 20.8. Assume that the **for** loop of lines 5–7 of the DFS procedure considers the vertices in alphabetical order, and assume that each adjacency list is ordered alphabetically.

20.4-2

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

20.4-3

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

20.4-4

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

20.4-5

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

20.5 Strongly connected components

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

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

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

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

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

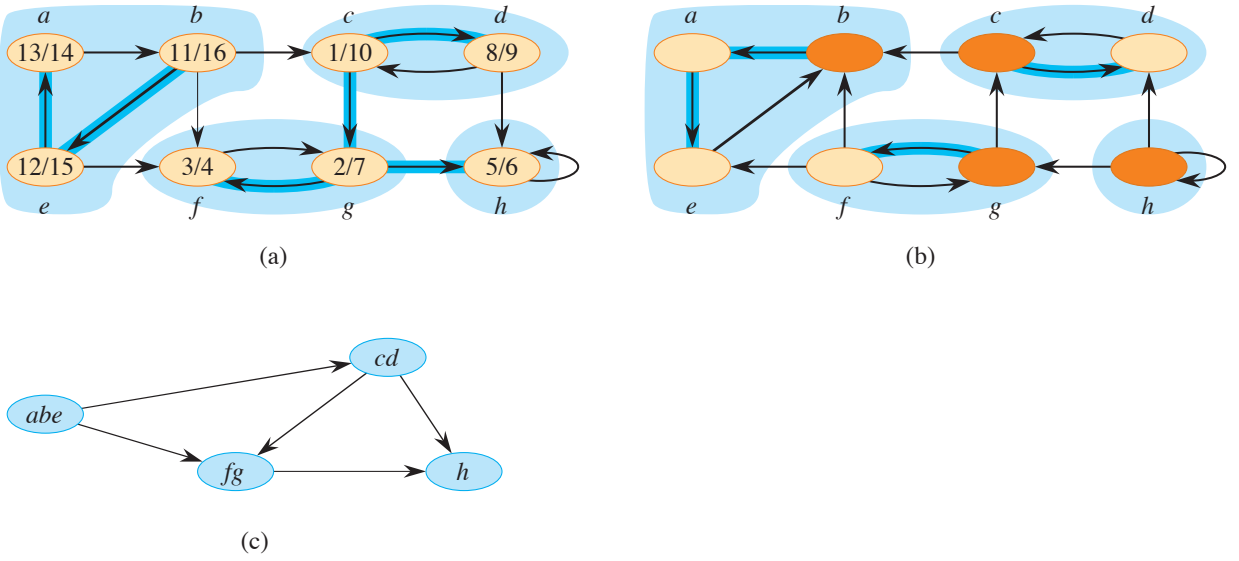


Figure 20.9 **(a)** A directed graph G. Each region shaded light blue is a strongly connected component of G. Each vertex is labeled with its discovery and finish times in a depth-first search, and tree edges are dark blue. **(b)** The graph G^{T}, the transpose of G, with the depth-first forest computed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

STRONGLY-CONNECTED-COMPONENTS(G)

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

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

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

Lemma 20.13
Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$, let $u, v \in C$, let $u', v' \in C'$, and suppose that G contains a path $u \rightsquigarrow u'$. Then G cannot also contain a path $v' \rightsquigarrow v$.

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

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

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

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

Lemma 20.14
Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E$, where $u \in C'$ and $v \in C$. Then $f(C') > f(C)$.

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

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

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

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

Corollary 20.15
Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$, and suppose that $f(C) > f(C')$. Then E^{T} contains no edge (v, u) such that $u \in C'$ and $v \in C$.

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

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

Theorem 20.16
The STRONGLY-CONNECTED-COMPONENTS procedure correctly computes the strongly connected components of the directed graph G provided as its input.

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

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

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

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

Exercises

20.5-1

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

20.5-2

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

20.5-3

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

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

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

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

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

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

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

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

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

Problems

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

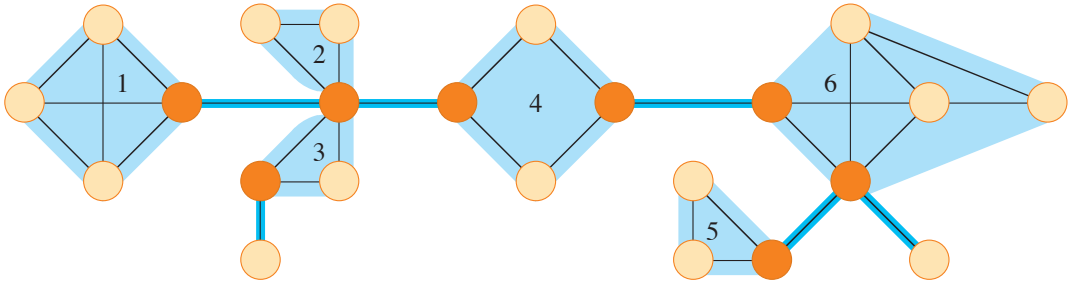


Figure 20.10 The articulation points, bridges, and biconnected components of a connected, undirected graph for use in Problem 20-2. The articulation points are the orange vertices, the bridges are the dark blue edges, and the biconnected components are the edges in the light blue regions, with a *bcc*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

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

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

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

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

20-2 Articulation points, bridges, and biconnected components

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

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

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

c. Let

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

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

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

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

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

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

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

20-3 Euler tour

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

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

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

20-4 Reachability

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

20-5 Inserting and querying vertices in planar graphs

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

Consider the following two operations on a planar graph G:

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

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

Chapter notes

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

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

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

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

21 Minimum Spanning Trees

Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together. To interconnect a set of n pins, the designer 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.

To model this wiring problem, use 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 $(u, v) \in E$, a weight $w(u, v)$ specifies the cost (amount of wire needed) to connect u and v. The goal is 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 21.1 shows an example of a connected graph and a minimum spanning tree.

This chapter studies two ways to solve the minimum-spanning-tree problem. Kruskal's algorithm and Prim's algorithm both run in $O(E \lg V)$ time. Prim's algorithm achieves this bound by using a binary heap as a priority queue. By using Fibonacci heaps instead (see page 478), Prim's algorithm runs in $O(E + V \lg V)$ time. This bound is better than $O(E \lg V)$ whenever $|E|$ grows asymptotically faster than $|V|$.

---

[1] The phrase "minimum spanning tree" is a shortened form of the phrase "minimum-weight spanning tree." There is no point in minimizing the number of edges in T, since all spanning trees have exactly $|V| - 1$ edges by Theorem B.2 on page 1169.

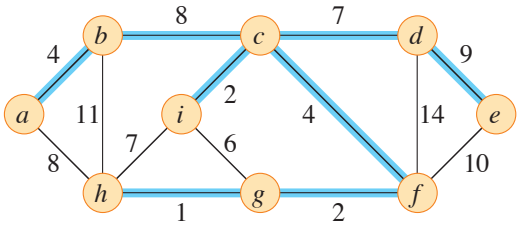


Figure 21.1 A minimum spanning tree for a connected graph. The weights on edges are shown, and the blue edges form a minimum spanning tree. The total weight of the tree shown is 37. This minimum spanning tree is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another spanning tree with weight 37.

The two algorithms are greedy algorithms, as described in Chapter 15. Each step of a greedy algorithm must make one of several possible choices. The greedy strategy advocates making the choice that is the best at the moment. Such a strategy does not generally guarantee that it always finds globally optimal solutions to problems. For the minimum-spanning-tree problem, however, we can prove that certain greedy strategies do yield a spanning tree with minimum weight. Although you can read this chapter independently of Chapter 15, the greedy methods presented here are a classic application of the theoretical notions introduced there.

Section 21.1 introduces a "generic" minimum-spanning-tree method that grows a spanning tree by adding one edge at a time. Section 21.2 gives two algorithms that implement the generic method. The first algorithm, due to Kruskal, is similar to the connected-components algorithm from Section 19.1. The second, due to Prim, resembles Dijkstra's shortest-paths algorithm (Section 22.3).

Because a tree is a type of graph, in order to be precise we must define a tree in terms of not just its edges, but its vertices as well. Because this chapter focuses on trees in terms of their edges, we'll implicitly understand that the vertices of a tree T are those that some edge of T is incident on.

21.1 Growing a minimum spanning tree

The input to the minumum-spanning-tree problem is a connected, undirected graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$. The goal is to find a minimum spanning tree for G. The two algorithms considered 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 procedure GENERIC-MST on the facing page, which grows the minimum spanning tree one edge at a time. The generic method manages a set A of edges, maintaining the following loop invariant:

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

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

Each step determines an edge (u, v) that the procedure can add 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 added safely to A while maintaining the invariant.

This generic algorithm uses 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 belong to a minimum spanning tree, and the loop must terminate by the time it has considered all edges. Therefore, the set A 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.

The remainder of this section provides a rule (Theorem 21.1) 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 21.2 illustrates this notion. We say that an edge $(u, v) \in E$ ***crosses*** the cut $(S, V - S)$ if one of its endpoints belongs to S and the other belongs to $V - S$. 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. 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.

The following theorem gives the rule for recognizing safe edges.

Theorem 21.1

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

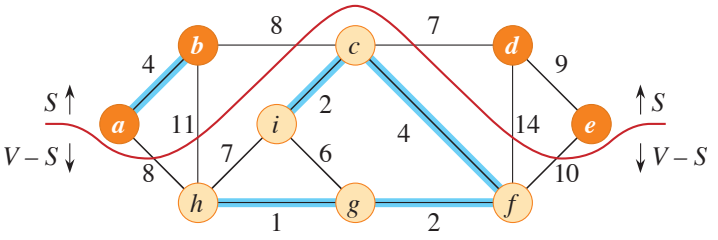


Figure 21.2 A cut $(S, V - S)$ of the graph from Figure 21.1. Orange vertices belong to the set S, and tan vertices belong to $V - S$. The edges crossing the cut are those connecting tan vertices with orange vertices. The edge (d, c) is the unique light edge crossing the cut. Blue edges form a subset A of the edges. The cut $(S, V - S)$ respects A, since no edge of A crosses the cut.

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'll 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 simple path p from u to v in T, as Figure 21.3 illustrates. Since u and v are on opposite sides of the cut $(S, V - S)$, at least one edge in T lies on the simple path p and 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 simple 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, $w(u, v) \le w(x, y)$. Therefore,

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

But T is a minimum spanning tree, so that $w(T) \le w(T')$, and thus, T' must be a minimum spanning tree as well.

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

Theorem 21.1 provides insight into how the GENERIC-MST method works on a connected graph $G = (V, E)$. As the method proceeds, the set A is always acyclic, since it is a subset of a minimum spanning tree and a tree may not contain a cycle.

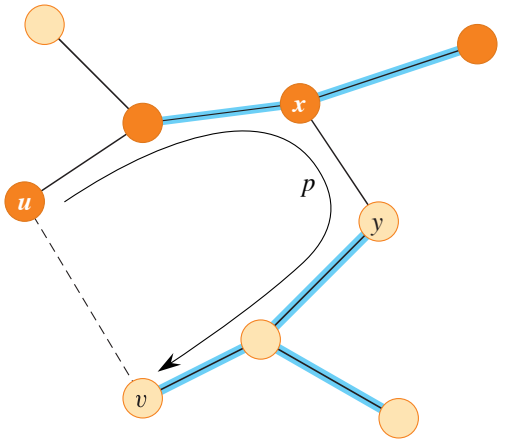


Figure 21.3 The proof of Theorem 21.1. Orange vertices belong to S, and tan vertices belong to $V - S$. Only edges in the minimum spanning tree T are shown, along with edge (u, v), which does not lie in T. The edges in A are blue, and (u, v) is a light edge crossing the cut $(S, V - S)$. The edge (x, y) is an edge on the unique simple path p from u to v in T. To form a minimum spanning tree T' that contains (u, v), remove the edge (x, y) from T and add the edge (u, v).

At any point in the execution, 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 method 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 **while** loop in lines 2–4 of GENERIC-MST executes $|V| - 1$ times because it finds one of the $|V| - 1$ edges of a minimum spanning tree in each iteration. 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 method terminates.

The two algorithms in Section 21.2 use the following corollary to Theorem 21.1.

Corollary 21.2

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

21.1-1

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

21.1-2

Professor Sabatier conjectures the following converse of Theorem 21.1. 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.

21.1-3

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.

21.1-4

Give a simple example of a connected 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.

21.1-5

Let e be a maximum-weight edge on some cycle of connected graph $G = (V, E)$. 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.

21.1-6

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.

21.1-7

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.

21.1-8
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'.

21.1-9
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'.

21.1-10
Given a graph G and a minimum spanning tree T, suppose that the weight of one of the edges in T decreases. 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'.

★ ***21.1-11***
Given a graph G and a minimum spanning tree T, suppose that the weight of one of the edges *not* in T decreases. Give an algorithm for finding the minimum spanning tree in the modified graph.

21.2 The algorithms of Kruskal and Prim

The two minimum-spanning-tree algorithms described in this section elaborate on the generic method. 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 whose vertices are all those of the given graph. The safe edge added to A is always a lowest-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 lowest-weight edge connecting the tree to a vertex not in the tree. Both algorithms assume that the input graph is connected and represented by adjacency lists.

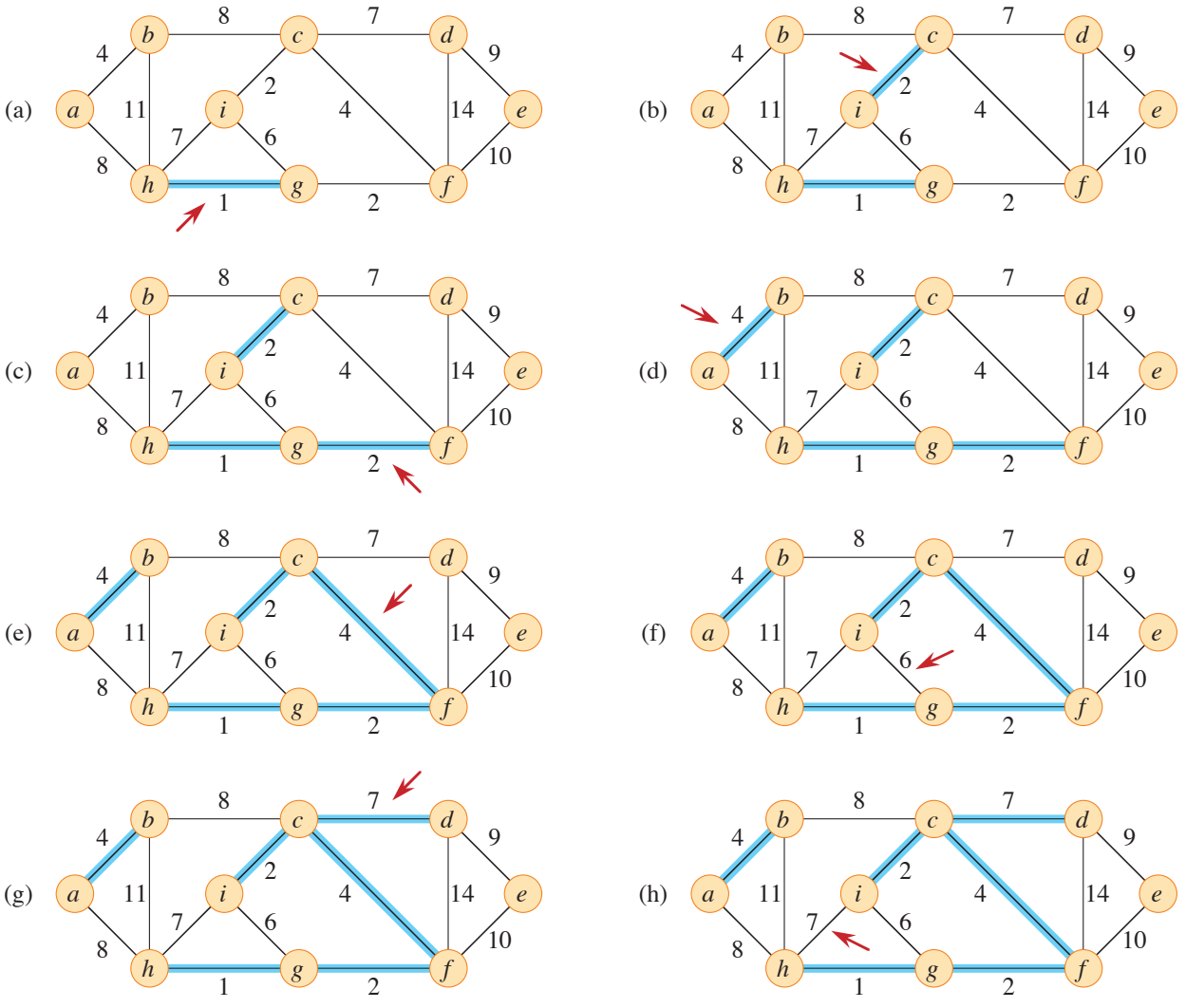


Figure 21.4 The execution of Kruskal's algorithm on the graph from Figure 21.1. Blue edges belong to the forest A being grown. The algorithm considers each edge in sorted order by weight. A red arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

Kruskal's algorithm

Kruskal's algorithm 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) with the lowest 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 21.2 implies

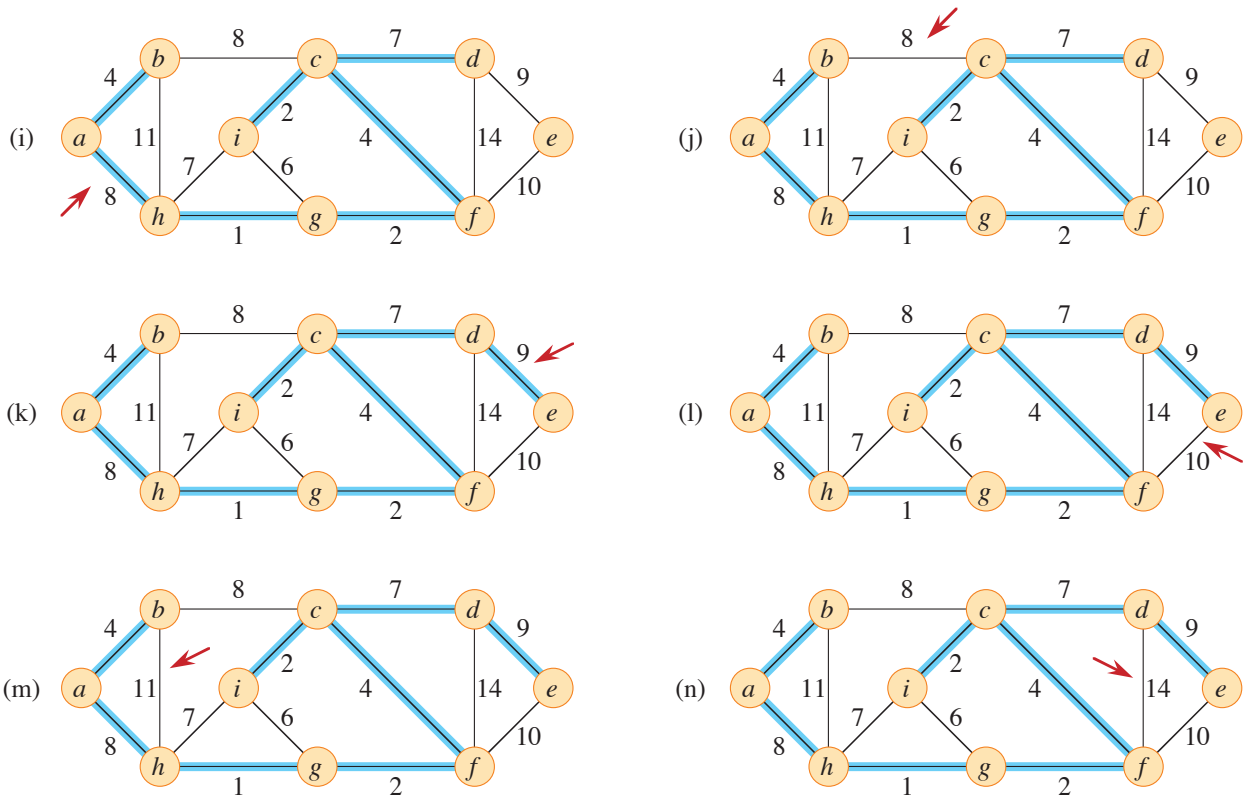


Figure 21.4, continued Further steps in the execution of Kruskal's algorithm.

that (u, v) is a safe edge for C_1. Kruskal's algorithm qualifies as a greedy algorithm because at each step it adds to the forest an edge with the lowest possible weight.

Like the algorithm to compute connected components from Section 19.1, the procedure MST-KRUSKAL on the following page uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the vertices in one tree of the current forest. The operation FIND-SET(u) returns a representative element from the set that contains u. Thus, to determine whether two vertices u and v belong to the same tree, just test whether FIND-SET(u) equals FIND-SET(v). To combine trees, Kruskal's algorithm calls the UNION procedure.

Figure 21.4 shows how Kruskal's algorithm works. Lines 1–3 initialize the set A to the empty set and create $|V|$ trees, one containing each vertex. The **for** loop in lines 6–9 examines edges in order of weight, from lowest to highest. The loop 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 ignored. Otherwise, the two vertices belong to different

```
MST-KRUSKAL(G, w)
 1  A = ∅
 2  for each vertex v ∈ G.V
 3       MAKE-SET(v)
 4  create a single list of the edges in G.E
 5  sort the list of edges into monotonically increasing order by weight w
 6  for each edge (u, v) taken from the sorted list in order
 7       if FIND-SET(u) ≠ FIND-SET(v)
 8            A = A ∪ {(u, v)}
 9            UNION(u, v)
10  return A
```

trees. In this case, line 8 adds the edge (u, v) to A, and line 9 merges the vertices in the two trees.

The running time of Kruskal's algorithm for a graph $G = (V, E)$ depends on the specific implementation of the disjoint-set data structure. Let's assume that it uses the disjoint-set-forest implementation of Section 19.3 with the union-by-rank and path-compression heuristics, since that is the asymptotically fastest implementation known. Initializing the set A in line 1 takes $O(1)$ time, creating a single list of edges in line 4 takes $O(V + E)$ time (which is $O(E)$ because G is connected), and the time to sort the edges in line 5 is $O(E \lg E)$. (We'll 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 6–9 performs $O(E)$ FIND-SET and UNION operations on the disjoint-set forest. Along with the $|V|$ MAKE-SET operations, these disjoint-set operations take a total of $O((V + E)\,\alpha(V))$ time, where α is the very slowly growing function defined in Section 19.4. Because we assume that G is 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 method from Section 21.1. Prim's algorithm operates much like Dijkstra's algorithm for finding shortest paths in a graph, which we'll see in Section 22.3. Prim's algorithm has the property that the edges in the set A always form a single tree. As Figure 21.5 shows, the tree starts from an arbitrary root vertex r and grows until it spans all the vertices in V. Each step adds to the tree A

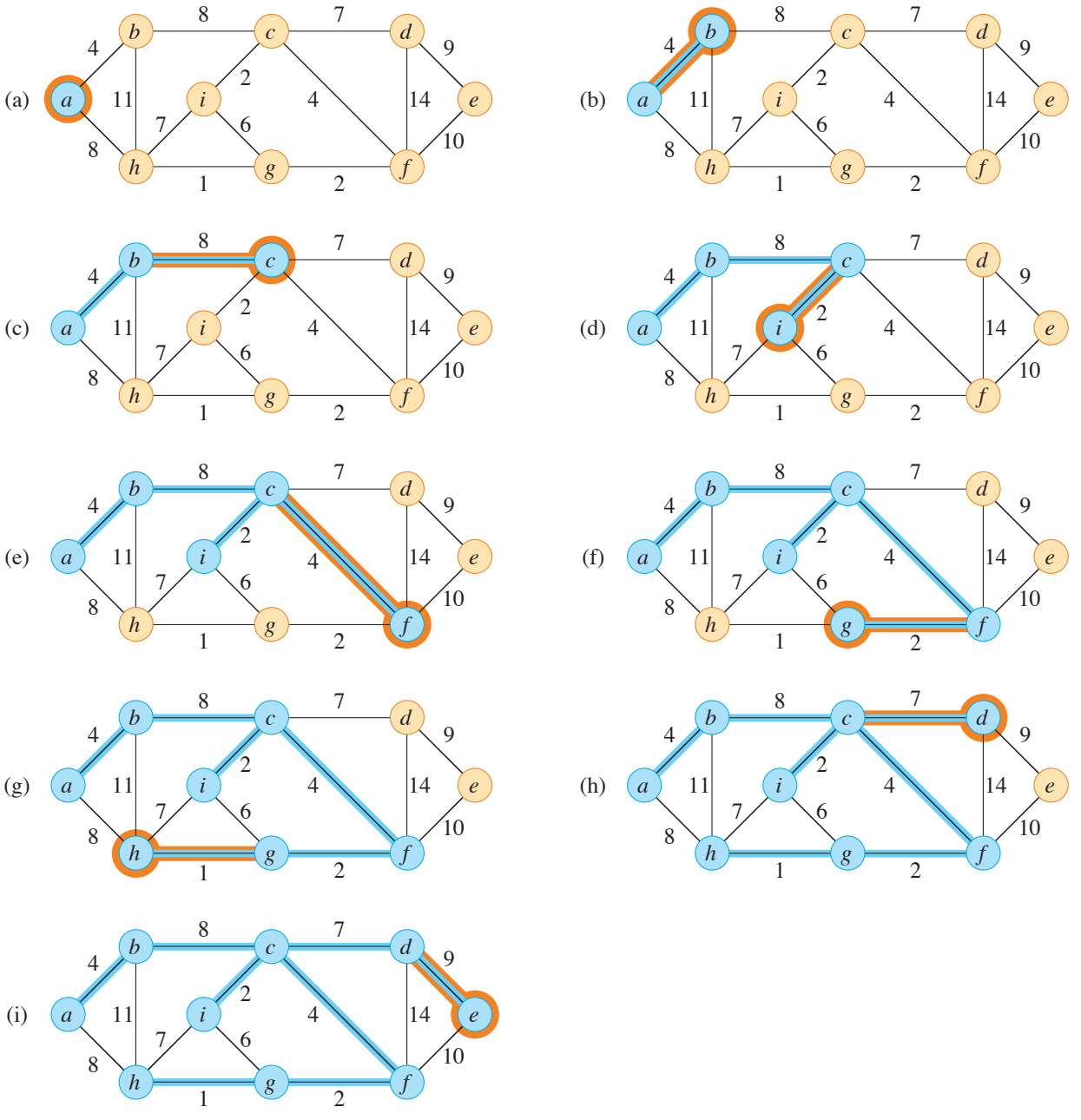


Figure 21.5 The execution of Prim's algorithm on the graph from Figure 21.1. The root vertex is a. Blue vertices and edges belong to the tree being grown, and tan vertices have yet to be added to the tree. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree. The edge and vertex added to the tree are highlighted in orange. In the second step (part (c)), 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.

a light edge that connects A to an isolated vertex—one on which no edge of A is incident. By Corollary 21.2, 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 qualifies as greedy since at each step it adds to the tree an edge that contributes the minimum amount possible to the tree's weight.

In the procedure MST-PRIM below, the connected graph G and the root r of the minimum spanning tree to be grown are inputs to the algorithm. In order to efficiently select a new edge to add into tree A, the algorithm maintains a min-priority queue Q of all vertices that are *not* in the tree, based on a *key* attribute. For each vertex v, the attribute $v.key$ is the minimum weight of any edge connecting v to a vertex in the tree, where by convention, $v.key = \infty$ if there is no such edge. The attribute $v.\pi$ names the parent of v in the tree. The algorithm implicitly maintains the set A from GENERIC-MST as

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

where we interpret the vertices in Q as forming a set. When the algorithm terminates, the min-priority queue Q is empty, and thus the minimum spanning tree A for G is

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

```
MST-PRIM(G, w, r)
 1  for each vertex u ∈ G.V
 2      u.key = ∞
 3      u.π = NIL
 4  r.key = 0
 5  Q = ∅
 6  for each vertex u ∈ G.V
 7      INSERT(Q, u)
 8  while Q ≠ ∅
 9      u = EXTRACT-MIN(Q)          // add u to the tree
10      for each vertex v in G.Adj[u]   // update keys of u's non-tree neighbors
11          if v ∈ Q and w(u, v) < v.key
12              v.π = u
13              v.key = w(u, v)
14              DECREASE-KEY(Q, v, w(u, v))
```

Figure 21.5 shows how Prim's algorithm works. Lines 1–7 set the key of each vertex to ∞ (except for the root r, whose key is set to 0 to make it the first vertex processed), set the parent of each vertex to NIL, and insert each vertex into the min-priority queue Q. The algorithm maintains the following three-part loop invariant:

Prior to each iteration of the **while** loop of lines 8–14,

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

Line 9 identifies a vertex $u \in Q$ incident on a light edge that crosses the cut $(V - Q, Q)$ (with the exception of the first iteration, in which $u = r$ due to lines 4–7). Removing u from the set Q adds it to the set $V - Q$ of vertices in the tree, thus adding the edge $(u, u.\pi)$ to A. The **for** loop of lines 10–14 updates the key and π attributes of every vertex v adjacent to u but not in the tree, thereby maintaining the third part of the loop invariant. Whenever line 13 updates $v.key$, line 14 calls DECREASE-KEY to inform the min-priority queue that v's key has changed.

The running time of Prim's algorithm depends on the specific implementation of the min-priority queue Q. You can implement Q with a binary min-heap (see Chapter 6), including a way to map between vertices and their corresponding heap elements. The BUILD-MIN-HEAP procedure can perform lines 5–7 in $O(V)$ time. In fact, there is no need to call BUILD-MIN-HEAP. You can just put the key of r at the root of the min-heap, and because all other keys are ∞, they can go anywhere else in the min-heap. The body of the **while** loop executes $|V|$ 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 10–14 executes $O(E)$ 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 11 can take constant time if you keep a bit for each vertex that indicates whether it belongs to Q and update the bit when the vertex is removed from Q. Each call to DECREASE-KEY in line 14 takes $O(\lg V)$ 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.

You can further improve the asymptotic running time of Prim's algorithm by implementing the min-priority queue with a Fibonacci heap (see page 478). If a Fibonacci heap holds $|V|$ elements, an EXTRACT-MIN operation takes $O(\lg V)$ amortized time and each INSERT and DECREASE-KEY operation takes only $O(1)$ amortized time. Therefore, by using 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

21.2-1

Kruskal's algorithm can return different spanning trees for the same input graph G, depending on how it breaks ties when the edges are sorted. 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.

21.2-2

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

21.2-3

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

21.2-4

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?

21.2-5

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?

21.2-6

Professor Borden 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.

★ ***21.2-7***
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?

★ ***21.2-8***
Suppose that a graph G has a minimum spanning tree already computed. How quickly can you update the minimum spanning tree upon adding a new vertex and incident edges to G?

Problems

21-1 Second-best minimum spanning tree
Let $G = (V, E)$ be an undirected, connected graph whose weight function is $w : E \to \mathbb{R}$, and suppose that $|E| \geq |V|$ and all edge weights are distinct.

We define a second-best minimum spanning tree 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 $w(T') = \min\{w(T'') : T'' \in \mathcal{T} - \{T\}\}$.

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 the minimum spanning tree of G. Prove that G contains some edge $(u, v) \in T$ and some edge $(x, y) \notin T$ such that $(T - \{(u, v)\}) \cup \{(x, y)\}$ is a second-best minimum spanning tree of G.

c. Now let T be any spanning tree of G and, for any two vertices $u, v \in V$, let $max[u, v]$ denote an edge of maximum weight on the unique simple 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.

21-2 Minimum spanning tree in sparse graphs
For a very sparse connected graph $G = (V, E)$, it is possible to further improve upon the $O(E + V \lg V)$ running time of Prim's algorithm with a Fibonacci heap by preprocessing G to decrease the number of vertices before running Prim's algorithm. In particular, for each vertex u, choose the minimum-weight edge (u, v)

incident on u, and put (u, v) into the minimum spanning tree under construction. Then, contract all chosen edges (see Section B.4). Rather than contracting these edges one at a time, first identify sets of vertices that are united into the same new vertex. Then create the graph that would have resulted from contracting these edges one at a time, but do so by "renaming" edges according to the sets into which their endpoints were placed. Several edges from the original graph might 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, set the minimum spanning tree T being constructed to be empty, and for each edge $(u, v) \in E$, initialize the two attributes $(u, v).orig = (u, v)$ and $(u, v).c = w(u, v)$. 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 on the facing page takes inputs G and T, and it returns a contracted graph G' with updated attributes $orig'$ and c'. The procedure also accumulates edges of G into the minimum spanning tree 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 c' is the weight attribute on the edges of $G'.E$ and r is any vertex in $G'.V$. Prove that $T \cup \{(x, y).orig' : (x, y) \in A\}$ is a minimum spanning tree of G.

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

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

d. Suppose that you run k phases of MST-REDUCE, using the output G' produced by one phase as the input G 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), you run Prim's algorithm by calling MST-PRIM(G', c', r), where G', with weight attribute c', is returned by the last phase and r is any vertex in $G'.V$. 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?

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

21-3 Alternative minimum-spanning-tree algorithms

Consider the three algorithms MAYBE-MST-A, MAYBE-MST-B, and MAYBE-MST-C on the next page. Each one takes a connected graph and a weight function as input and returns a set of edges T. For each algorithm, either prove that T is a minimum spanning tree or prove that T is not necessarily a minimum spanning tree. Also describe the most efficient implementation of each algorithm, regardless of whether it computes a minimum spanning tree.

21-4 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. The value of the bottleneck spanning tree is the weight of the maximum-weight edge in T.

```
MAYBE-MST-A(G, w)
1  sort the edges into monotonically decreasing order of edge weights w
2  T = E
3  for each edge e, taken in monotonically decreasing order by weight
4      if T − {e} is a connected graph
5          T = T − {e}
6  return T

MAYBE-MST-B(G, w)
1  T = ∅
2  for each edge e, taken in arbitrary order
3      if T ∪ {e} has no cycles
4          T = T ∪ {e}
5  return T

MAYBE-MST-C(G, w)
1  T = ∅
2  for each edge e, taken in arbitrary order
3      T = T ∪ {e}
4      if T has a cycle c
5          let e′ be a maximum-weight edge on c
6          T = T − {e′}
7  return 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, you will show how to find a bottleneck spanning tree 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 might want to use a subroutine that contracts sets of edges, as in the MST-REDUCE procedure described in Problem 21-2.)

Chapter notes

Tarjan [429] surveys the minimum-spanning-tree problem and provides excellent advanced material. Graham and Hell [198] compiled a history of the minimum-spanning-tree problem.

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 21-2. Kruskal's algorithm was reported by Kruskal [272] in 1956. The algorithm commonly known as Prim's algorithm was indeed invented by Prim [367], but it was also invented earlier by V. Jarník in 1930.

When $|E| = \Omega(V \lg V)$, Prim's algorithm, implemented with a Fibonacci heap, 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 [156] give an algorithm that runs in $O(E \lg^* V)$ time. Gabow, Galil, Spencer, and Tarjan [165] improved this algorithm to run in $O(E \lg\lg^* V)$ time. Chazelle [83] gives an algorithm that runs in $O(E\, \hat{\alpha}(E, V))$ time, where $\hat{\alpha}(E, V)$ is the functional inverse of Ackermann's function. (See the chapter notes for Chapter 19 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. Pettie and Ramachandran [356] give an algorithm based on precomputed "MST decision trees" that also runs in $O(E\, \hat{\alpha}(E, V))$ time.

A related problem is ***spanning-tree verification***: given a graph $G = (V, E)$ and a tree $T \subseteq E$, determine whether T is a minimum spanning tree of G. King [254] gives a linear-time algorithm to verify a spanning tree, building on earlier work of Komlós [269] and Dixon, Rauch, and Tarjan [120].

The above algorithms are all deterministic and fall into the comparison-based model described in Chapter 8. Karger, Klein, and Tarjan [243] 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 [158] 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.

22 Single-Source Shortest Paths

Suppose that you need to drive from Oceanside, New York, to Oceanside, California, by the shortest possible route. Your GPS contains information about the entire road network of the United States, including the road distance between each pair of adjacent intersections. How can your GPS determine this shortest route?

One possible way is to enumerate all the routes from Oceanside, New York, to Oceanside, California, add up the distances on each route, and select the shortest. But even disallowing routes that contain cycles, your GPS would need to examine an enormous number of possibilities, most of which are simply not worth considering. For example, a route that passes through Miami, Florida, is a poor choice, because Miami is several hundred miles out of the way.

This chapter and Chapter 23 show how to solve such problems efficiently. The input to a ***shortest-paths problem*** is a weighted, directed graph $G = (V, E)$, with a weight function $w : E \to \mathbb{R}$ mapping edges to real-valued weights. The ***weight*** $w(p)$ 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*** $\delta(u, v)$ 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 example of going from Oceanside, New York, to Oceanside, California, your GPS models the road network as a graph: vertices represent intersections, edges represent road segments between intersections, and edge weights represent road distances. The goal is to find a shortest path from a given intersection in

Oceanside, New York (say, Brower Avenue and Skillman Avenue) to a given intersection in Oceanside, California (say, Topeka Street and South Horne Street).

Edge weights can represent metrics other than distances, such as time, cost, penalties, loss, or any other quantity that accumulates linearly along a path and that you want to minimize.

The breadth-first-search algorithm from Section 20.2 is a shortest-paths algorithm that works on unweighted graphs, that is, graphs in which each edge has unit weight. Because many of the concepts from breadth-first search arise in the study of shortest paths in weighted graphs, you might want to review Section 20.2 before proceeding.

Variants

This chapter focuses on the ***single-source shortest-paths problem***: given a graph $G = (V, E)$, find a shortest path from a given ***source vertex*** $s \in V$ to every vertex $v \in V$. The algorithm for the single-source problem can solve many other problems, 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, you 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 you solve the single-source problem with source vertex u, you solve this problem also. Moreover, all known algorithms for this problem have the same worst-case asymptotic running time as the best single-source algorithms.

All-pairs shortest-paths problem: Find a shortest path from u to v for every pair of vertices u and v. Although you can solve this problem by running a single-source algorithm once from each vertex, you often can solve it faster. Additionally, its structure is interesting in its own right. Chapter 23 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 24 also relies on this property.) Recall that optimal substructure is one of the key indicators that dynamic programming (Chapter 14) and the greedy method (Chapter 15) might apply. Dijkstra's algorithm, which we shall see in Section 22.3, is a greedy algorithm, and the Floyd-Warshall algorithm, which finds a shortest path between every pair of vertices (see Sec-

tion 23.2), is a dynamic-programming algorithm. The following lemma states the optimal-substructure property of shortest paths more precisely.

Lemma 22.1 (Subpaths of shortest paths are shortest paths)
Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \le i \le j \le k$, 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 Decompose path p into $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$, so that $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. Now, assume that there is a path p'_{ij} from v_i to v_j with weight $w(p'_{ij}) < w(p_{ij})$. Then, $v_0 \overset{p_{0i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ is a path from v_0 to v_k whose weight $w(p_{0i}) + w(p'_{ij}) + w(p_{jk})$ is less than $w(p)$, which contradicts the assumption that p is a shortest path from v_0 to v_k. ■

Negative-weight edges

Some instances of the single-source shortest-paths problem may include 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 the graph contains 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—you can always find a path with lower weight by following the proposed "shortest" path and then traversing 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 22.1 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 $\langle s, a \rangle$), we have $\delta(s, a) = w(s, a) = 3$. 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 $\delta(s, c) = w(s, c) = 5$, and 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. Because 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, you 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, you can also find paths with arbitrarily large negative weights from s to g,

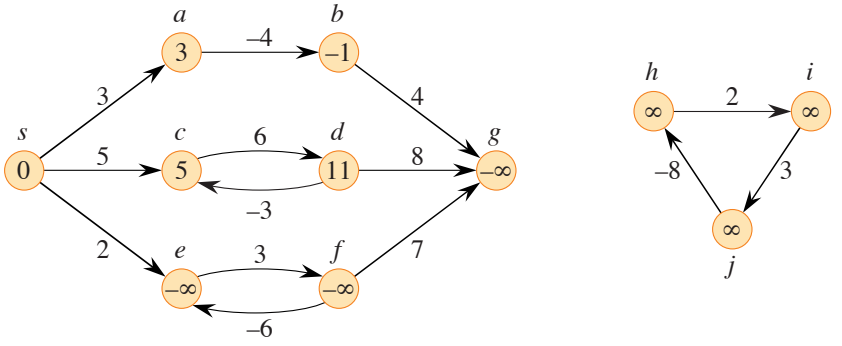


Figure 22.1 Negative edge weights in a directed graph. The shortest-path weight from source s appears within each vertex. Because vertices e and f form a negative-weight cycle reachable from s, they have shortest-path weights of $-\infty$. Because vertex g is reachable from a vertex whose shortest-path 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 ∞, even though they lie on a negative-weight cycle.

and so $\delta(s, g) = -\infty$. 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 a road network. Others, 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. You 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, you can repeatedly remove these cycles from the path until you have a shortest path that is cycle-free. Therefore, without loss of

generality, assume that shortest paths have no cycles, that is, they are simple paths. Since any acyclic path in a graph $G = (V, E)$ contains at most $|V|$ distinct vertices, it also contains at most $|V| - 1$ edges. Assume, therefore, that any shortest path contains at most $|V| - 1$ edges.

Representing shortest paths

It is usually not enough to compute only shortest-path weights. Most applications of shortest paths need to know the vertices on shortest paths as well. For example, if your GPS told you the distance to your destination but not how to get there, it would not be terribly useful. We represent shortest paths similarly to how we represented breadth-first trees in Section 20.2. Given a graph $G = (V, E)$, maintain for each vertex $v \in V$ a ***predecessor*** $v.\pi$ that is either another vertex or NIL. The shortest-paths algorithms in this chapter set the π attributes so that the chain of predecessors originating at a vertex v runs backward along a shortest path from s to v. Thus, given a vertex v for which $v.\pi \neq \text{NIL}$, the procedure PRINT-PATH(G, s, v) from Section 20.2 prints a shortest path from s to v.

In the midst of executing a shortest-paths algorithm, however, the π values might not indicate shortest paths. The ***predecessor subgraph*** $G_\pi = (V_\pi, E_\pi)$ induced by the π values is defined the same for single-source shortest paths as for breadth-first search in equations (20.2) and (20.3) on page 561:

$$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\} ,$$
$$E_\pi = \{(v.\pi, v) \in E : v \in V_\pi - \{s\}\} .$$

We'll prove that the π values produced by the algorithms in this chapter have the property that at termination G_π 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 20.2, 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 $w : E \to \mathbb{R}$, 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.

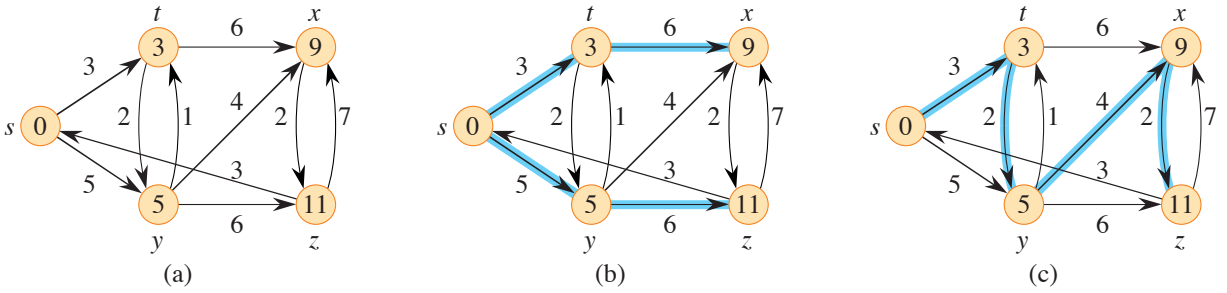


Figure 22.2 **(a)** A weighted, directed graph with shortest-path weights from source s. **(b)** The blue edges form a shortest-paths tree rooted at the source s. **(c)** Another shortest-paths tree with the same root.

Shortest paths are not necessarily unique, and neither are shortest-paths trees. For example, Figure 22.2 shows a weighted, directed graph and two shortest-paths trees with the same root.

Relaxation

The algorithms in this chapter use the technique of ***relaxation***. For each vertex $v \in V$, the single-source shortest paths algorithms maintain an attribute $v.d$, which is an upper bound on the weight of a shortest path from source s to v. We call $v.d$ a ***shortest-path estimate***. To initialize the shortest-path estimates and predecessors, call the $\Theta(V)$-time procedure INITIALIZE-SINGLE-SOURCE. After initialization, we have $v.\pi = \text{NIL}$ for all $v \in V$, $s.d = 0$ and $v.d = \infty$ for $v \in V - \{s\}$.

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2      v.d = ∞
3      v.π = NIL
4  s.d = 0
```

The process of ***relaxing*** an edge (u, v) consists of testing whether going through vertex u improves the shortest path to vertex v found so far and, if so, updating $v.d$ and $v.\pi$. A relaxation step might decrease the value of the shortest-path estimate $v.d$ and update v's predecessor attribute $v.\pi$. The RELAX procedure on the following page performs a relaxation step on edge (u, v) in $O(1)$ time. Figure 22.3 shows two examples of relaxing an edge, one in which a shortest-path estimate decreases and one in which no estimate changes.

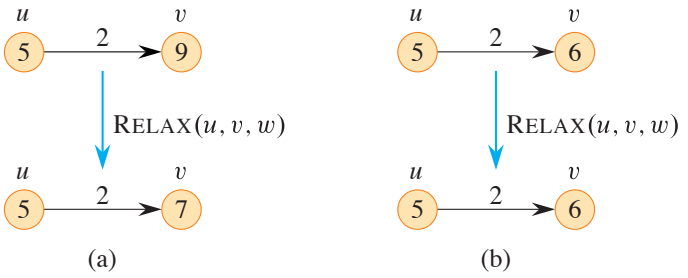


Figure 22.3 Relaxing an edge (u, v) with weight $w(u, v) = 2$. The shortest-path estimate of each vertex appears within the vertex. **(a)** Because $v.d > u.d + w(u, v)$ prior to relaxation, the value of $v.d$ decreases. **(b)** Since we have $v.d \leq u.d + w(u, v)$ before relaxing the edge, the relaxation step leaves $v.d$ unchanged.

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2      v.d = u.d + w(u, v)
3      v.π = u
```

Each algorithm in this chapter calls INITIALIZE-SINGLE-SOURCE and then repeatedly relaxes edges.[1] 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. Dijkstra's algorithm and the shortest-paths algorithm for directed acyclic graphs relax each edge exactly once. The Bellman-Ford algorithm relaxes each edge $|V| - 1$ times.

Properties of shortest paths and relaxation

To prove the algorithms in this chapter correct, we'll appeal to several properties of shortest paths and relaxation. We state these properties here, and Section 22.5 proves them formally. For your reference, each property stated here includes the appropriate lemma or corollary number from Section 22.5. The latter five of these properties, which refer to shortest-path estimates or the predecessor subgraph, im-

[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 $v.d \leq u.d + w(u, v)$, which, by the triangle inequality (Lemma 22.10 on page 633), must be satisfied if $u.d = \delta(s, u)$ and $v.d = \delta(s, v)$. That is, if $v.d \leq u.d + w(u, v)$, there is no "pressure" to satisfy this constraint, so the constraint is "relaxed."

plicitly 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 22.10)
For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Upper-bound property (Lemma 22.11)
We always have $v.d \geq \delta(s, v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s, v)$, it never changes.

No-path property (Corollary 22.12)
If there is no path from s to v, then we always have $v.d = \delta(s, v) = \infty$.

Convergence property (Lemma 22.14)
If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $v.d = \delta(s, v)$ at all times afterward.

Path-relaxation property (Lemma 22.15)
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 $v_k.d = \delta(s, v_k)$. 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 22.17)
Once $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Chapter outline

Section 22.1 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 remarkably simple, and it has the further benefit of detecting whether a negative-weight cycle is reachable from the source. Section 22.2 gives a linear-time algorithm for computing shortest paths from a single source in a directed acyclic graph. Section 22.3 covers Dijkstra's algorithm, which has a lower running time than the Bellman-Ford algorithm but requires the edge weights to be nonnegative. Section 22.4 shows how to use the Bellman-Ford algorithm to solve a special case of linear programming. Finally, Section 22.5 proves the properties of shortest paths and relaxation stated above.

This chapter does arithmetic with infinities, and so we need some conventions for when ∞ or $-\infty$ appears in an arithmetic expression. We 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 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 each algorithm traverses an adjacency list, it can find edge weights in $O(1)$ time per edge.

22.1 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 vertex s and weight function $w : E \to \mathbb{R}$, the Bellman-Ford algorithm returns a boolean value indicating whether 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 procedure BELLMAN-FORD relaxes edges, progressively decreasing an estimate $v.d$ 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 |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  return TRUE
```

Figure 22.4 shows the execution of the Bellman-Ford algorithm on a graph with 5 vertices. After initializing the d and π 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 22.4(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.)

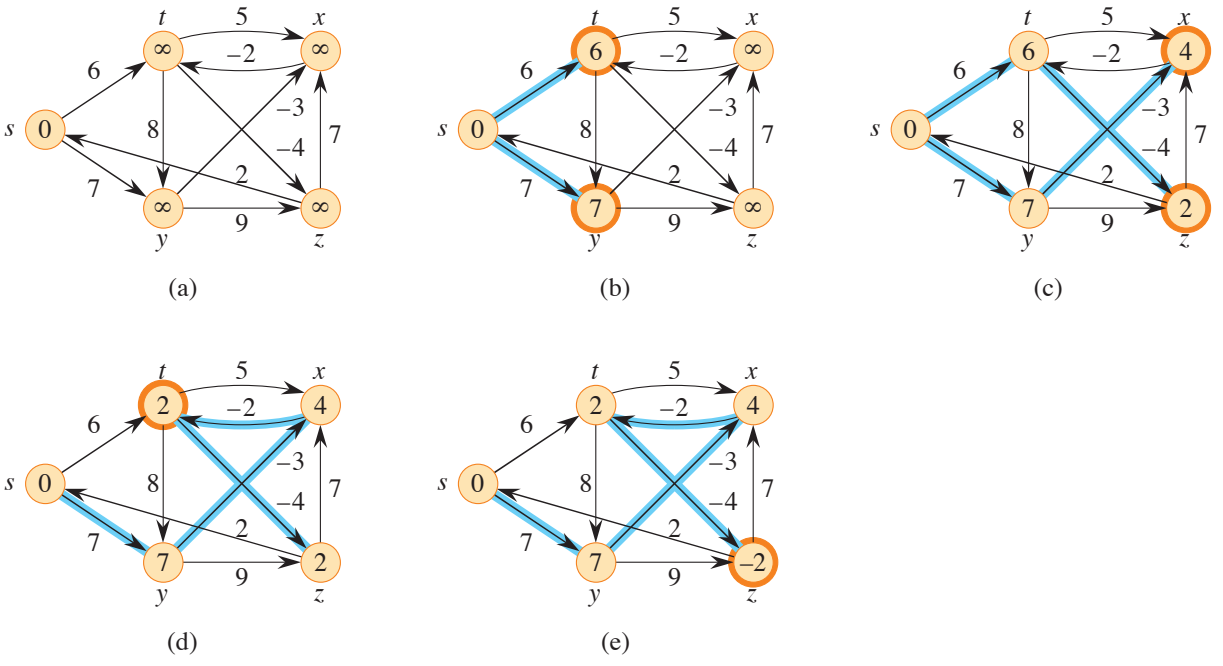


Figure 22.4 The execution of the Bellman-Ford algorithm. The source is vertex s. The d values appear within the vertices, and blue edges indicate predecessor values: if edge (u, v) is blue, then $v.\pi = u$. In this particular example, each pass relaxes the edges in the order $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. **(a)** The situation just before the first pass over the edges. **(b)–(e)** The situation after each successive pass over the edges. Vertices whose shortest-path estimates and predecessors have changed due to a pass are highlighted in orange. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

The Bellman-Ford algorithm runs in $O(V^2 + VE)$ time when the graph is represented by adjacency lists, since the initialization in line 1 takes $\Theta(V)$ time, each of the $|V| - 1$ passes over the edges in lines 2–4 takes $\Theta(V + E)$ time (examining $|V|$ adjacency lists to find the $|E|$ edges), and the **for** loop of lines 5–7 takes $O(V + E)$ time. Fewer than $|V| - 1$ passes over the edges sometimes suffice (see Exercise 22.1-3), which is why we say $O(V^2 + VE)$ time, rather than $\Theta(V^2 + VE)$ time. In the frequent case where $|E| = \Omega(V)$, we can express this running time as $O(VE)$. Exercise 22.1-5 asks you to make the Bellman-Ford algorithm run in $O(VE)$ time even when $|E| = o(V)$.

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.

Lemma 22.2
Let $G = (V, E)$ be a weighted, directed graph with source vertex s and weight function $w : E \to \mathbb{R}$, 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, $v.d = \delta(s, v)$ 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 shortest path from s to v. Because shortest paths are simple, 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 ith iteration, for $i = 1, 2, \ldots, k$, is (v_{i-1}, v_i). By the path-relaxation property, therefore, $v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$. ■

Corollary 22.3
Let $G = (V, E)$ be a weighted, directed graph with source vertex s and weight function $w : E \to \mathbb{R}$. Then, for each vertex $v \in V$, there is a path from s to v if and only if BELLMAN-FORD terminates with $v.d < \infty$ when it is run on G.

Proof The proof is left as Exercise 22.1-2. ■

Theorem 22.4 (Correctness of the Bellman-Ford algorithm)
Let BELLMAN-FORD be run on a weighted, directed graph $G = (V, E)$ with source vertex s and weight function $w : E \to \mathbb{R}$. If G contains no negative-weight cycles that are reachable from s, then the algorithm returns TRUE, $v.d = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph G_π 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, $v.d = \delta(s, v)$ for all vertices $v \in V$. If vertex v is reachable from s, then Lemma 22.2 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_π is a shortest-paths tree. Now we use the claim to show that BELLMAN-FORD returns TRUE. At termination, for all edges $(u, v) \in E$ we have

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

and so none of the tests in line 6 causes BELLMAN-FORD to return FALSE. Therefore, it returns TRUE.

Now, suppose that graph G contains a negative-weight cycle reachable from the source s. Let this cycle be $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$, in which case we have

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 . \tag{22.1}$$

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

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

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

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

Moreover, by Corollary 22.3, $v_i.d$ is finite for $i = 1, 2, \ldots, k$. Thus,

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

which contradicts inequality (22.1). 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

22.1-1

Run the Bellman-Ford algorithm on the directed graph of Figure 22.4, using vertex z as the source. In each pass, relax edges in the same order as in the figure, and show the d and π values after each pass. Now, change the weight of edge (z, x) to 4 and run the algorithm again, using s as the source.

22.1-2

Prove Corollary 22.3.

22.1-3
Given a weighted, directed graph $G = (V, E)$ with no negative-weight cycles, let m be the maximum over all vertices $v \in V$ of the minimum number of edges in a shortest path from the source s 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, even if m is not known in advance.

22.1-4
Modify the Bellman-Ford algorithm so that it sets $v.d$ to $-\infty$ for all vertices v for which there is a negative-weight cycle on some path from the source to v.

22.1-5
Suppose that the graph given as input to the Bellman-Ford algorithm is represented with a list of $|E|$ edges, where each edge indicates the vertices it leaves and enters, along with its weight. Argue that the Bellman-Ford algorithm runs in $O(VE)$ time without the constraint that $|E| = \Omega(V)$. Modify the Bellman-Ford algorithm so that it runs in $O(VE)$ time in all cases when the input graph is represented with adjacency lists.

22.1-6
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$. Give an $O(VE)$-time algorithm to find, for all vertices $v \in V$, the value $\delta^*(v) = \min\{\delta(u, v) : u \in V\}$.

22.1-7
Suppose that a weighted, directed graph $G = (V, E)$ contains a negative-weight cycle. Give an efficient algorithm to list the vertices of one such cycle. Prove that your algorithm is correct.

22.2 Single-source shortest paths in directed acyclic graphs

In this section, we introduce one further restriction on weighted, directed graphs: they are acyclic. That is, we are concerned with weighted dags. Shortest paths are always well defined in a dag, since even if there are negative-weight edges, no negative-weight cycles can exist. We'll see that if the edges of a weighted dag $G = (V, E)$ are relaxed according to a topological sort of its vertices, it takes only $\Theta(V + E)$ time to compute shortest paths from a single source.

The algorithm starts by topologically sorting the dag (see Section 20.4) to impose a linear ordering on the vertices. If the dag contains a path from vertex u to vertex v, then u precedes v in the topological sort. The DAG-SHORTEST-PATHS

procedure makes just one pass over the vertices in the topologically sorted order. As it processes each vertex, it relaxes each edge that leaves the vertex. Figure 22.5 shows the execution of this algorithm.

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

Let's analyze the running time of this algorithm. As shown in Section 20.4, the topological sort of line 1 takes $\Theta(V + E)$ time. The call of INITIALIZE-SINGLE-SOURCE in line 2 takes $\Theta(V)$ time. The **for** loop of lines 3–5 makes one iteration per vertex. Altogether, the **for** loop of lines 4–5 relaxes each edge exactly once. (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.

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

Theorem 22.5

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, $v.d = \delta(s, v)$ for all vertices $v \in V$, and the predecessor subgraph G_π is a shortest-paths tree.

Proof We first show that $v.d = \delta(s, v)$ for all vertices $v \in V$ at termination. If v is not reachable from s, then $v.d = \delta(s, v) = \infty$ 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 DAG-SHORTEST-PATHS processes the vertices in topologically sorted order, it relaxes the edges on p in the order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. The path-relaxation property implies that $v_i.d = \delta(s, v_i)$ at termination for $i = 0, 1, \ldots, k$. Finally, by the predecessor-subgraph property, G_π is a shortest-paths tree. ■

A useful application of this algorithm arises in determining critical paths in ***PERT chart***[2] analysis. A job consists of several tasks. Each task takes a certain

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

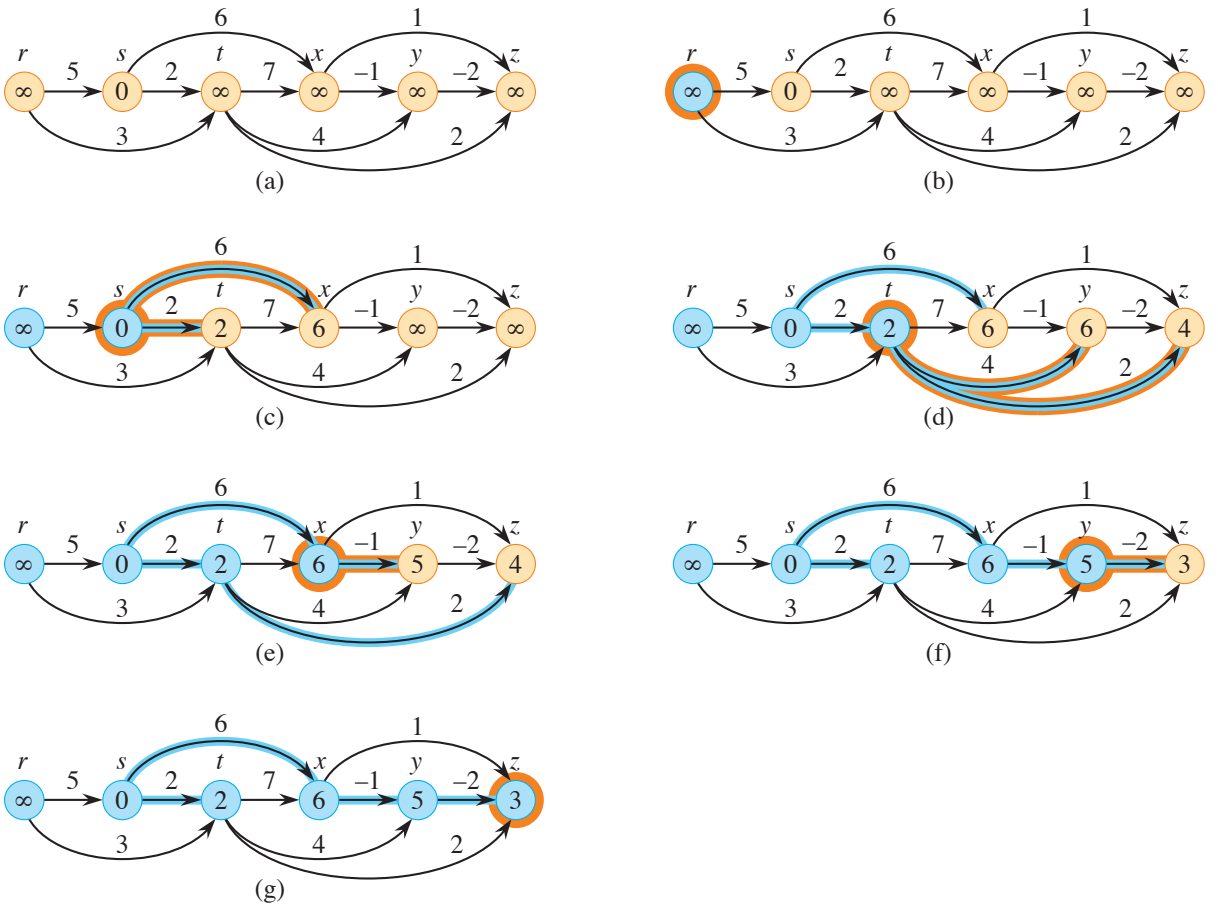


Figure 22.5 The execution of the algorithm for shortest paths in a directed acyclic graph. The vertices are topologically sorted from left to right. The source vertex is s. The d values appear within the vertices, and blue edges indicate the π values. **(a)** The situation before the first iteration of the **for** loop of lines 3–5. **(b)–(g)** The situation after each iteration of the **for** loop of lines 3–5. Blue vertices have had their outgoing edges relaxed. The vertex highlighted in orange was used as u in that iteration. Each edge highlighted in orange caused a d value to change when it was relaxed in that iteration. The values shown in part (g) are the final values.

amount of time, and some tasks must be completed before others can be started. For example, if the job is to build a house, then the foundation must be completed before starting to frame the exterior walls, which must be completed before starting on the roof. Some tasks require more than one other task to be completed before they can be started: before the drywall can be installed over the wall framing, both the electrical system and plumbing must be installed. A dag models the tasks and dependencies. Edges represent tasks, with the weight of an edge indicating the time required to perform the task. Vertices represent "milestones," which are

achieved when all the tasks represented by the edges entering the vertex have been completed. If edge (u, v) enters vertex v and edge (v, x) leaves v, then task (u, v) must be completed before task (v, x) is started. A path through this dag represents a sequence of tasks 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 any sequence of tasks. Thus, the weight of a critical path provides a lower bound on the total time to perform all the tasks, even if as many tasks as possible are performed simultaneously. You can find a critical path by either

- negating the edge weights and running DAG-SHORTEST-PATHS, or
- running DAG-SHORTEST-PATHS, but replacing "∞" by "$-\infty$" in line 2 of INITIALIZE-SINGLE-SOURCE and "$>$" by "$<$" in the RELAX procedure.

Exercises

22.2-1

Show the result of running DAG-SHORTEST-PATHS on the directed acyclic graph of Figure 22.5, using vertex r as the source.

22.2-2

Suppose that you 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 remains correct.

22.2-3

An alternative way to represent a PERT chart looks more like the dag of Figure 20.7 on page 574. Vertices represent tasks and edges represent sequencing constraints, that is, edge (u, v) indicates that task u must be performed before task v. Vertices, not edges, have weights. Modify the DAG-SHORTEST-PATHS procedure so that it finds a longest path in a directed acyclic graph with weighted vertices in linear time.

★ ***22.2-4***

Give an efficient algorithm to count the total number of paths in a directed acyclic graph. The count should include all paths between all pairs of vertices and all paths with 0 edges. Analyze your algorithm.

22.3 Dijkstra's algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$, but it requires nonnegative weights on all edges: $w(u, v) \geq 0$ 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.

You can think of Dijkstra's algorithm as generalizing breadth-first search to weighted graphs. A wave emanates from the source, and the first time that a wave arrives at a vertex, a new wave emanates from that vertex. Whereas breadth-first search operates as if each wave takes unit time to traverse an edge, in a weighted graph, the time for a wave to traverse an edge is given by the edge's weight. Because a shortest path in a weighted graph might not have the fewest edges, a simple, first-in, first-out queue won't suffice for choosing the next vertex from which to send out a wave.

Instead, 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 into S, and relaxes all edges leaving u. The procedure DIJKSTRA replaces the first-in, first-out queue of breadth-first search by 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 = ∅
 4  for each vertex u ∈ G.V
 5      INSERT(Q, u)
 6  while Q ≠ ∅
 7      u = EXTRACT-MIN(Q)
 8      S = S ∪ {u}
 9      for each vertex v in G.Adj[u]
10          RELAX(u, v, w)
11          if the call of RELAX decreased v.d
12              DECREASE-KEY(Q, v, v.d)
```

Dijkstra's algorithm relaxes edges as shown in Figure 22.6. Line 1 initializes the d and π values in the usual way, 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

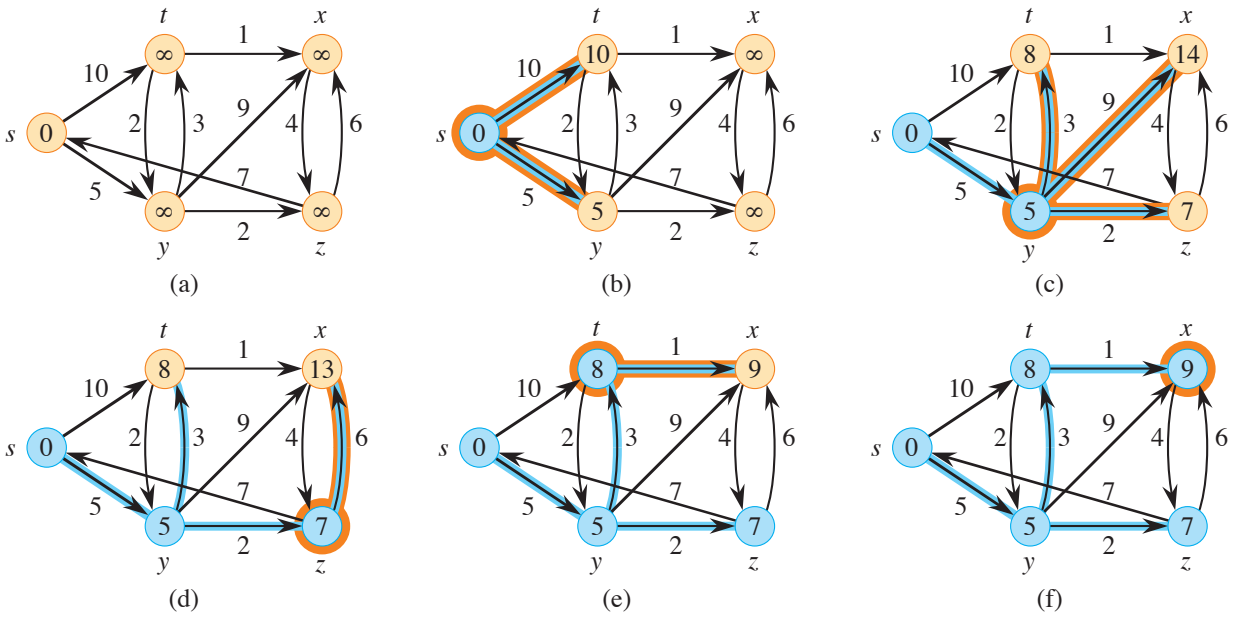


Figure 22.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and blue edges indicate predecessor values. Blue vertices belong to the set S, and tan vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 6–12. **(b)–(f)** The situation after each successive iteration of the **while** loop. In each part, the vertex highlighted in orange was chosen as vertex u in line 7, and each edge highlighted in orange caused a d value and a predecessor to change when the edge was relaxed. The d values and predecessors shown in part (f) are the final values.

of the **while** loop of lines 6–12. Lines 3–5 initialize the min-priority queue Q to contain all the vertices in V. Since $S = \emptyset$ at that time, the invariant is true upon first reaching line 6. Each time through the **while** loop of lines 6–12, line 7 extracts a vertex u from $Q = V - S$ and line 8 adds it to set S, thereby maintaining the invariant. (The first time through this loop, $u = s$.) Vertex u, therefore, has the smallest shortest-path estimate of any vertex in $V - S$. Then, lines 9–12 relax each edge (u, v) leaving u, thus updating the estimate $v.d$ and the predecessor $v.\pi$ if the shortest path to v found so far improves by going through u. Whenever a relaxation step changes the d and π values, the call to DECREASE-KEY in line 12 updates the min-priority queue. The algorithm never inserts vertices into Q after the **for** loop of lines 4–5, and each vertex is extracted from Q and added to S exactly once, so that the **while** loop of lines 6–12 iterates exactly $|V|$ times.

Because Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in $V - S$ to add to set S, you can think of it as using a greedy strategy. Chapter 15 explains greedy strategies in detail, but you need not have read that chapter to understand Dijkstra's algorithm. Greedy strategies do not always yield optimal

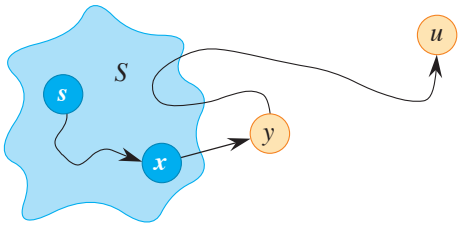


Figure 22.7 The proof of Theorem 22.6. Vertex u is selected to be added into set S in line 7 of DIJKSTRA. Vertex y is the first vertex on a shortest path from the source s to vertex u that is not in set S, and $x \in S$ is y's predecessor on that shortest path. The subpath from y to u may or may not re-enter set S.

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 $u.d = \delta(s, u)$ each time it adds a vertex u to set S.

Theorem 22.6 (Correctness of Dijkstra's algorithm)
Dijkstra's algorithm, run on a weighted, directed graph $G = (V, E)$ with nonnegative weight function w and source vertex s, terminates with $u.d = \delta(s, u)$ for all vertices $u \in V$.

Proof We will show that at the start of each iteration of the **while** loop of lines 6–12, we have $v.d = \delta(s, v)$ for all $v \in S$. The algorithm terminates when $S = V$, so that $v.d = \delta(s, v)$ for all $v \in V$.

The proof is by induction on the number of iterations of the **while** loop, which equals $|S|$ at the start of each iteration. There are two bases: for $|S| = 0$, so that $S = \emptyset$ and the claim is trivially true, and for $|S| = 1$, so that $S = \{s\}$ and $s.d = \delta(s, s) = 0$.

For the inductive step, the inductive hypothesis is that $v.d = \delta(s, v)$ for all $v \in S$. The algorithm extracts vertex u from $V - S$. Because the algorithm adds u into S, we need to show that $u.d = \delta(s, u)$ at that time. If there is no path from s to u, then we are done, by the no-path property. If there is a path from s to u, then, as Figure 22.7 shows, let y be the first vertex on a shortest path from s to u that is not in S, and let $x \in S$ be the predecessor of y on that shortest path. (We could have $y = u$ or $x = s$.) Because y appears no later than u on the shortest path and all edge weights are nonnegative, we have $\delta(s, y) \leq \delta(s, u)$. Because the call of EXTRACT-MIN in line 7 returned u as having the minimum d value in $V - S$, we also have $u.d \leq y.d$, and the upper-bound property gives $\delta(s, u) \leq u.d$.

Since $x \in S$, the inductive hypothesis implies that $x.d = \delta(s, x)$. During the iteration of the **while** loop that added x into S, edge (x, y) was relaxed. By the convergence property, $y.d$ received the value of $\delta(s, y)$ at that time. Thus, we have

$$\delta(s, y) \le \delta(s, u) \le u.d \le y.d \quad \text{and} \quad y.d = \delta(s, y) ,$$

so that

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

Hence, $u.d = \delta(s, u)$, and by the upper-bound property, this value never changes again. ■

Corollary 22.7
After Dijkstra's algorithm is run on a weighted, directed graph $G = (V, E)$ with nonnegative weight function w and source vertex s, the predecessor subgraph G_π is a shortest-paths tree rooted at s.

Proof Immediate from Theorem 22.6 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 (in line 5), EXTRACT-MIN (in line 7), and DECREASE-KEY (in line 12). The algorithm calls both INSERT and EXTRACT-MIN once per vertex. Because each vertex $u \in V$ is added to set S exactly once, each edge in the adjacency list $Adj[u]$ is examined in the **for** loop of lines 9–12 exactly once during the course of the algorithm. Since the total number of edges in all the adjacency lists is $|E|$, this **for** loop iterates a total of $|E|$ times, and thus the algorithm calls DECREASE-KEY at most $|E|$ times overall. (Observe once again that we are using aggregate analysis.)

Just as in Prim's algorithm, the running time of Dijkstra's algorithm depends on the specific implementation of the min-priority queue Q. A simple implementation takes advantage of the vertices being numbered 1 to $|V|$: simply store $v.d$ in the vth entry of an array. Each INSERT and DECREASE-KEY operation takes $O(1)$ time, and each EXTRACT-MIN operation takes $O(V)$ time (since it has 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, $E = o(V^2/\lg V)$—you can improve the running time by implementing the min-priority queue with a binary min-heap that includes a way to map between vertices and their corresponding heap elements. Each EXTRACT-MIN operation then takes $O(\lg V)$ time. As before, there are $|V|$ such operations. The time to build the binary min-heap is $O(V)$. (As noted in Section 21.2, you don't even need to call BUILD-MIN-HEAP.) Each DECREASE-KEY operation takes $O(\lg V)$ time, and there are still at most $|E|$ such operations. The total running time is therefore $O((V + E) \lg V)$, which is $O(E \lg V)$ in the typical case that $|E| = \Omega(V)$. This running time improves upon the straightforward $O(V^2)$-time implementation if $E = o(V^2/\lg V)$.

By implementing the min-priority queue with a Fibonacci heap (see page 478), you can improve the running time to $O(V \lg V + E)$. 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 Dijkstra's algorithm typically makes many more DECREASE-KEY calls than EXTRACT-MIN calls, so that 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 resembles both breadth-first search (see Section 20.2) and Prim's algorithm for computing minimum spanning trees (see Section 21.2). 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

22.3-1

Run Dijkstra's algorithm on the directed graph of Figure 22.2, first using vertex s as the source and then using vertex z as the source. In the style of Figure 22.6, show the d and π values and the vertices in set S after each iteration of the **while** loop.

22.3-2

Give a simple example of a directed graph with negative-weight edges for which Dijkstra's algorithm produces an incorrect answer. Why doesn't the proof of Theorem 22.6 go through when negative-weight edges are allowed?

22.3-3

Suppose that you change line 6 of Dijkstra's algorithm to read

6 **while** $|Q| > 1$

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

22.3-4
Modify the DIJKSTRA procedure so that the priority queue Q is more like the queue in the BFS procedure in that it contains only vertices that have been reached from source s so far: $Q \subseteq V - S$ and $v \in Q$ implies $v.d \neq \infty$.

22.3-5
Professor Gaedel has written a program that he claims implements Dijkstra's algorithm. The program produces $v.d$ and $v.\pi$ for each vertex $v \in V$. Give an $O(V + E)$-time algorithm to check the output of the professor's program. It should determine whether the d and π attributes match those of some shortest-paths tree. You may assume that all edge weights are nonnegative.

22.3-6
Professor Newman thinks that he has worked out a simpler proof of correctness for Dijkstra's algorithm. He claims that Dijkstra's algorithm relaxes the edges of every shortest path in the graph in the order in which they appear on the path, and therefore the path-relaxation property applies to every vertex reachable from the source. Show that the professor is mistaken by constructing a directed graph for which Dijkstra's algorithm relaxes the edges of a shortest path out of order.

22.3-7
Consider 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. Interpret $r(u, v)$ as the probability that the channel from u to v will not fail, and assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.

22.3-8
Let $G = (V, E)$ be a weighted, directed graph with positive 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 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 you run a breadth-first search on G'. Show that the order in which the breadth-first search of G' colors vertices in V black is the same as the order in which Dijkstra's algorithm extracts the vertices of V from the priority queue when it runs on G.

22.3-9
Let $G = (V, E)$ be a weighted, directed graph with nonnegative weight function $w : E \to \{0, 1, \ldots, W\}$ for some nonnegative integer W. Modify Dijkstra's algo-

rithm to compute the shortest paths from a given source vertex s in $O(WV + E)$ time.

22.3-10
Modify your algorithm from Exercise 22.3-9 to run in $O((V + E) \lg W)$ time. (*Hint:* How many distinct shortest-path estimates can $V - S$ contain at any point in time?)

22.3-11
Suppose that you 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.

22.3-12
Suppose that you have a weighted directed graph $G = (V, E)$ in which all edge weights are positive real values in the range $[C, 2C]$ for some positive constant C. Modify Dijkstra's algorithm so that it runs in $O(V + E)$ time.

22.4 Difference constraints and shortest paths

Chapter 29 studies the general linear-programming problem, showing how to optimize a linear function subject to a set of linear inequalities. This section investigates a special case of linear programming that reduces to finding shortest paths from a single source. The Bellman-Ford algorithm then solves the resulting single-source shortest-paths problem, thereby also solving the linear-programming problem.

Linear programming

In the general ***linear-programming problem***, the input is an $m \times n$ matrix A, an m-vector b, and an n-vector c. The goal is 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 $Ax \leq b$.

The most popular method for solving linear programs is the ***simplex algorithm***, which Section 29.1 discusses. Although the simplex algorithm 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. We offer here two reasons to understand the setup of linear-programming problems. First, if you know that you can cast a given problem as a polynomial-sized linear-programming problem, then you im-

mediately have a polynomial-time algorithm to solve the problem. Second, faster algorithms exist for many special cases of linear programming. For example, the single-pair shortest-path problem (Exercise 22.4-4) and the maximum-flow problem (Exercise 24.1-5) are special cases of linear programming.

Sometimes the objective function does not matter: it's enough just to find any ***feasible solution***, that is, any vector x that satisfies $Ax \le b$, or to determine that no feasible solution exists. This section focuses 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 $1 \le i, j \le n, i \ne j$, and $1 \le k \le m$.

For example, consider the problem of finding a 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 values for the unknowns x_1, x_2, x_3, x_4, x_5, satisfying the following 8 difference constraints:

$$x_1 - x_2 \le 0 \,, \tag{22.2}$$
$$x_1 - x_5 \le -1 \,, \tag{22.3}$$
$$x_2 - x_5 \le 1 \,, \tag{22.4}$$
$$x_3 - x_1 \le 5 \,, \tag{22.5}$$
$$x_4 - x_1 \le 4 \,, \tag{22.6}$$
$$x_4 - x_3 \le -1 \,, \tag{22.7}$$
$$x_5 - x_3 \le -3 \,, \tag{22.8}$$
$$x_5 - x_4 \le -3 \,. \tag{22.9}$$

One solution to this problem is $x = (-5, -3, 0, -1, -4)$, which you can verify directly by checking each inequality. In fact, this problem has more than one solution.

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 22.8

Let $x = (x_1, x_2, \ldots, x_n)$ be a solution to a system $Ax \leq b$ 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 $Ax \leq b$ 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 various applications. For example, the unknowns x_i might be times at which events are to occur. Each constraint states that at least a certain amount of time, or at most a certain amount of time, must elapse between two events. Perhaps the events are jobs to be performed during the assembly of a product. If the manufacturer applies an adhesive that takes 2 hours to set at time x_1 and has to wait until it sets to install a part at time x_2, then there is a constraint that $x_2 \geq x_1 + 2$ or, equivalently, that $x_1 - x_2 \leq -2$. Alternatively, the manufacturer might require the part to be installed after the adhesive has been applied but no later than the time that the adhesive has set halfway. In this case, there is a pair of constraints $x_2 \geq x_1$ and $x_2 \leq x_1 + 1$ or, equivalently, $x_1 - x_2 \leq 0$ and $x_2 - x_1 \leq 1$.

If all the constraints have nonnegative numbers on the right-hand side—that is, if $b_i \geq 0$ for $i = 1, 2, \ldots, m$—then finding a feasible solution is trivial: just set all the unknowns x_i equal to each other. Then all the differences are 0, and every constraint is satisfied. The problem of finding a feasible solution to a system of difference constraints is interesting only if at least one constraint has $b_i < 0$.

Constraint graphs

We can interpret systems of difference constraints from a graph-theoretic point of view. For a system $Ax \leq b$ of difference constraints, let's view the $m \times n$ linear-programming matrix A as the transpose of an incidence matrix (see Exercise 20.1-7) 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

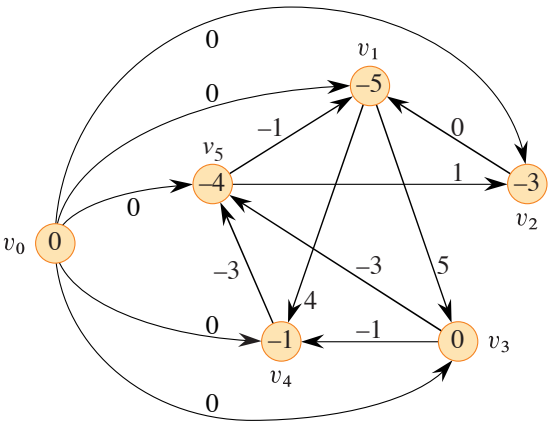


Figure 22.8 The constraint graph corresponding to the system (22.2)–(22.9) of difference constraints. The value of $\delta(v_0, v_i)$ appears in each vertex v_i. One feasible solution to the system is $x = (-5, -3, 0, -1, -4)$.

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

and

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

The constraint graph includes the additional vertex v_0, as we shall see shortly, to guarantee that the graph has some vertex that can reach all other vertices. 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 $x_j - x_i \le b_k$ 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 22.8 shows the constraint graph for the system (22.2)–(22.9) of difference constraints.

The following theorem shows how to solve a system of difference constraints by finding shortest-path weights in the corresponding constraint graph.

Theorem 22.9

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

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \ldots, \delta(v_0, v_n)) \tag{22.10}$$

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 (22.10) gives a feasible solution. Consider any edge $(v_i, v_j) \in E$. The triangle inequality implies that $\delta(v_0, v_j) \le \delta(v_0, v_i) + w(v_i, v_j)$, which is equivalent to $\delta(v_0, v_j) - \delta(v_0, v_i) \le w(v_i, v_j)$. Thus, letting $x_i = \delta(v_0, v_i)$ and $x_j = \delta(v_0, v_j)$ satisfies the difference constraint $x_j - x_i \le w(v_i, v_j)$ 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:

$$\begin{aligned}
x_2 - x_1 &\le w(v_1, v_2) , \\
x_3 - x_2 &\le w(v_2, v_3) , \\
&\vdots \\
x_{k-1} - x_{k-2} &\le w(v_{k-2}, v_{k-1}) , \\
x_k - x_{k-1} &\le w(v_{k-1}, v_k) .
\end{aligned}$$

We'll assume that x has a solution satisfying each of these k inequalities and then derive a contradiction. The solution must also satisfy the inequality that results from summing the k inequalities together. In summing the left-hand sides, each unknown x_i is added in once and subtracted out once (remember that $v_1 = v_k$ implies $x_1 = x_k$), so that the left-hand side sums to 0. The right-hand side sums to the weight $w(c)$ of the cycle, giving $0 \le w(c)$. But since c is a negative-weight cycle, $w(c) < 0$, and we obtain the contradiction that $0 \le w(c) < 0$. ■

Solving systems of difference constraints

Theorem 22.9 suggests how to use the Bellman-Ford algorithm to solve a system of difference constraints. Because the constraint graph contains edges from the source vertex v_0 to all other vertices, 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 22.8, for example, the shortest-path weights provide the feasible solution $x = (-5, -3, 0, -1, -4)$, and by Lemma 22.8, $x = (d - 5, d - 3, d, d - 1, d - 4)$ 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, the Bellman-Ford algorithm provides a way to solve the system in $O((n + 1)(n + m)) = O(n^2 + nm)$ time.

Exercise 22.4-5 asks you to modify the algorithm to run in $O(nm)$ time, even if m is much less than n.

Exercises

22.4-1

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

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

22.4-2

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

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

22.4-3

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

22.4-4

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

22.4-5

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

22.4-6

Consider adding ***equality constraints*** of the form $x_i = x_j + b_k$ to a system of difference constraints. Show how to solve this variety of constraint system.

22.4-7

Show how to solve a system of difference constraints by a Bellman-Ford-like algorithm that runs on a constraint graph without the extra vertex v_0.

★ ### 22.4-8

Let $Ax \leq b$ 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 $Ax \leq b$ and $x_i \leq 0$ for all x_i.

★ ### 22.4-9

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.

22.4-10

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.

22.4-11

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.

★ ### 22.4-12

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.

22.5 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 on page 611. In this section, we prove them.

The triangle inequality

In studying breadth-first search (Section 20.2), we proved as Lemma 20.1 a simple property of shortest distances in unweighted graphs. The triangle inequality generalizes the property to weighted graphs.

Lemma 22.10 (Triangle inequality)
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$ and source vertex s. Then, for all edges $(u, v) \in E$,

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

Proof Suppose that p is a shortest path 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 22.5-3 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 by executing a sequence of relaxation steps on the edges of a weighted, directed graph that has been initialized by INITIALIZE-SINGLE-SOURCE.

Lemma 22.11 (Upper-bound property)
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$. Let $s \in V$ be the source vertex, and let the graph be initialized by INITIALIZE-SINGLE-SOURCE(G, s). Then, $v.d \geq \delta(s, v)$ for all $v \in V$, and this invariant is maintained over any sequence of relaxation steps on the edges of G. Moreover, once $v.d$ achieves its lower bound $\delta(s, v)$, it never changes.

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

For the base case, $v.d \geq \delta(s, v)$ holds after initialization, since if $v.d = \infty$, then $v.d \geq \delta(s, v)$ for all $v \in V - \{s\}$, and since $s.d = 0 \geq \delta(s, s)$. (Note that $\delta(s, s) = -\infty$ if s is on a negative-weight cycle and that $\delta(s, s) = 0$ otherwise.)

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

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

and so the invariant is maintained.

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

Corollary 22.12 (No-path property)
Suppose that in a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, 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 $v.d = \delta(s, v) = \infty$, and this equation 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 $\infty = \delta(s, v) \leq v.d$, and thus $v.d = \infty = \delta(s, v)$. ■

Lemma 22.13
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$, and let $(u, v) \in E$. Then, immediately after edge (u, v) is relaxed by a call of RELAX(u, v, w), we have $v.d \leq u.d + w(u, v)$.

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

Lemma 22.14 (Convergence property)
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$, 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 $u.d = \delta(s, u)$ at any time prior to the call, then $v.d = \delta(s, v)$ at all times after the call.

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

$$\begin{aligned} v.d &\le u.d + w(u, v) && \text{(by Lemma 22.13)} \\ &= \delta(s, u) + w(u, v) \\ &= \delta(s, v) && \text{(by Lemma 22.1 on page 606)} . \end{aligned}$$

The upper-bound property gives $v.d \ge \delta(s, v)$, from which we conclude that $v.d = \delta(s, v)$, and this equation is maintained thereafter. ■

Lemma 22.15 (Path-relaxation property)
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$, 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, relaxing the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$ 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 ith edge of path p is relaxed, we have $v_i.d = \delta(s, v_i)$. For the base case, $i = 0$, and before any edges of p have been relaxed, we have from the initialization that $v_0.d = s.d = 0 = \delta(s, s)$. By the upper-bound property, the value of $s.d$ never changes after initialization.

For the inductive step, assume that $v_{i-1}.d = \delta(s, v_{i-1})$. What happens when edge (v_{i-1}, v_i) is relaxed? By the convergence property, after this relaxation, we have $v_i.d = \delta(s, v_i)$, and this equation 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_π induced by the resulting π 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 22.16
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$, 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_π 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_π is the source vertex, and the lemma is trivially true. Consider a predecessor subgraph G_π that arises after a sequence of relaxation steps. We first prove that G_π is acyclic. Suppose for the sake of contradiction that some relaxation step creates a cycle in the graph G_π. Let the cycle be $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_k = v_0$. Then, $v_i.\pi = v_{i-1}$ for $i = 1, 2, \ldots, k$ and, without loss of generality, assume that relaxing edge (v_{k-1}, v_k) created the cycle in G_π.

We claim that all vertices on cycle c are reachable from the source vertex 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 π value. By the upper-bound property, each vertex on cycle c has a finite shortest-path weight, which means that it is reachable from s.

We'll examine the shortest-path estimates on cycle c immediately before 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 $v_i.\pi = v_{i-1}$ for $i = 1, 2, \ldots, k-1$. Thus, for $i = 1, 2, \ldots, k-1$, the last update to $v_i.d$ was by the assignment $v_i.d = v_{i-1}.d + w(v_{i-1}, v_i)$. If $v_{i-1}.d$ changed since then, it decreased. Therefore, just before the call RELAX(v_{k-1}, v_k, w), we have

$$v_i.d \geq v_{i-1}.d + w(v_{i-1}, v_i) \qquad \text{for all } i = 1, 2, \ldots, k-1 \,. \tag{22.11}$$

Because $v_k.\pi$ is changed by the call RELAX(v_{k-1}, v_k, w), immediately beforehand we also have the strict inequality

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

Summing this strict inequality with the $k-1$ inequalities (22.11), we obtain the sum of the shortest-path estimates around cycle c:

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

But

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

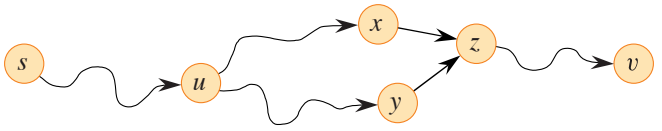


Figure 22.9 Showing that a simple path in G_π from source vertex s to vertex v is unique. If G_π contains two paths p_1 ($s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v$) and p_2 ($s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v$), where $x \neq y$, then $z.\pi = x$ and $z.\pi = y$, a contradiction.

since each vertex in the cycle c appears exactly once in each summation. This equation 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_π is a directed, acyclic graph. To show that it forms a rooted tree with root s, it suffices (see Exercise B.5-2 on page 1175) to prove that for each vertex $v \in V_\pi$, there is a unique simple path from s to v in G_π.

The vertices in V_π are those with non-NIL π values, plus s. Exercise 22.5-6 asks you to prove that a path from s exists to each vertex in V_π.

To complete the proof of the lemma, we now show that for any vertex $v \in V_\pi$, the graph G_π contains at most one simple path from s to v. Suppose otherwise. That is, suppose that, as Figure 22.9 illustrates, G_π contains two simple paths from s to some vertex v: p_1, which we decompose into $s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v$, and p_2, which we decompose into $s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v$, where $x \neq y$ (though u could be s and z could be v). But then, $z.\pi = x$ and $z.\pi = y$, which implies the contradiction that $x = y$. We conclude that G_π contains a unique simple path from s to v, and thus G_π forms a rooted tree with root s. ■

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

Lemma 22.17 (Predecessor-subgraph property)
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \rightarrow \mathbb{R}$, let $s \in V$ be a source vertex, and assume that G contains no negative-weight cycles that are reachable from s. Then, after a call to INITIALIZE-SINGLE-SOURCE(G, s) followed by any sequence of relaxation steps on edges of G that produces $v.d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph G_π is a shortest-paths tree rooted at s.

Proof We must prove that the three properties of shortest-paths trees given on page 608 hold for G_π. To show the first property, we must show that V_π 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 $v.d$ if and only if $v.\pi \neq \text{NIL}$, since both assignments occur in RELAX. Thus, the vertices in V_π are exactly those reachable from s.

The second property, that G_π forms a rooted tree with root s, follows directly from Lemma 22.16.

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_π 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 $v_k = v$. Consider an edge (v_{i-1}, v_i) in path p. Because this edge belongs to G_π, the last relaxation that changed $v_i.d$ must have been of this edge. After that relaxation, we had $v_i.d = v_{i-1}.d + (v_{i-1}, v_i)$. Subsequently, an edge entering v_{i-1} could have been relaxed, causing $v_{i-1}.d$ to decrease further, but without changing $v_i.d$. Therefore, we have $v_i.d \geq v_{i-1}.d + w(v_{i-1}, v_i)$. Thus, for $i = 1, 2, \ldots, k$, we have both $v_i.d = \delta(s, v_i)$ and $v_i.d \geq v_{i-1}.d + w(v_{i-1}, v_i)$, which together imply $w(v_{i-1}, v_i) \leq \delta(s, v_i) - \delta(s, v_{i-1})$. Summing the weights along path p yields

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

Thus, we have $w(p) \leq \delta(s, v_k)$. 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 p is a shortest path from s to $v = v_k$. ■

Exercises

22.5-1

Give two shortest-paths trees for the directed graph of Figure 22.2 on page 609 other than the two shown.

22.5-2

Give an example of a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$ and source vertex 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).

22.5-3
Modify the proof of Lemma 22.10 to handle cases in which shortest-path weights are ∞ or $-\infty$.

22.5-4
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 $s.\pi$ to a non-NIL value, then G contains a negative-weight cycle.

22.5-5
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 $v.\pi$ is allowed 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 π values that produces a cycle in G_π. (By Lemma 22.16, such an assignment cannot be produced by a sequence of relaxation steps.)

22.5-6
Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$ and no negative-weight cycles. Let $s \in V$ be the source vertex, and let G be initialized by INITIALIZE-SINGLE-SOURCE(G, s). Use induction to prove that for every vertex $v \in V_\pi$, there exists a path from s to v in G_π and that this property is maintained as an invariant over any sequence of relaxations.

22.5-7
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 $v.d = \delta(s, v)$ for all $v \in V$.

22.5-8
Let G be an arbitrary weighted, directed graph with a negative-weight cycle reachable from the source vertex s. Show how to construct an infinite sequence of relaxations of the edges of G such that every relaxation causes a shortest-path estimate to change.

Problems

22-1 Yen's improvement to Bellman-Ford

The Bellman-Ford algorithm does not specify the order in which to relax edges in each pass. Consider the following method for deciding upon the order. Before the first pass, assign an arbitrary linear order $v_1, v_2, \ldots, v_{|V|}$ to the vertices of the input graph $G = (V, E)$. Then 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 belongs to 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 each pass of the Bellman-Ford algorithm relaxes edges in the following way. First, visit each vertex in the order $v_1, v_2, \ldots, v_{|V|}$, relaxing edges of E_f that leave the vertex. 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, $v.d = \delta(s, v)$ for all vertices $v \in V$.

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

22-2 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 π 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 one d-dimensional box nests inside another.

c. You are given a set of n d-dimensional boxes $\{B_1, B_2, \ldots, B_n\}$. Give an efficient algorithm to find 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.

22-3 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 one U.S. dollar buys 64 Indian rupees, one Indian rupee buys 1.8 Japanese yen, and one Japanese yen buys 0.009 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy $64 \times 1.8 \times 0.009 = 1.0368$ U.S. dollars, thus turning a profit of 3.68%.

Suppose that you are given n currencies $c_1, c_2, \ldots, c_n$ and an $n \times n$ table R of exchange rates, such that 1 unit of currency c_i buys $R[i, j]$ units of currency c_j.

a. Give an efficient algorithm to determine whether 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.

22-4 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, assuming that these values are nonnegative integers. The algorithm 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 it has examined all bits and computed the correct solution.

This problem examines an algorithm for computing the shortest paths from a single source by scaling edge weights. The input is a directed graph $G = (V, E)$ with nonnegative integer edge weights w. Let $W = \max\{w(u, v) : (u, v) \in E\}$ be the maximum weight of any edge. In this problem, you will develop an algorithm that runs in $O(E \lg W)$ time. Assume that all vertices are reachable from the source.

The scaling 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 $w_3(u, v) = \langle 110 \rangle = 6$. Also with $k = 5$, if $w(u, v) = \langle 00100 \rangle = 4$, then $w_4(u, v) = \langle 0010 \rangle = 2$. Define $\delta_i(u, v)$ as the shortest-path weight from vertex u

to vertex v using weight function w_i, so that $\delta_k(u, v) = \delta(u, v)$ 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 $v \in V$. Assume throughout that $|E| \geq |V| - 1$. You will show how to compute δ_i from δ_{i-1} in $O(E)$ 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 how to compute $\delta(s, v)$ for all $v \in V$ in $O(E)$ time.

b. Show how to compute $\delta_1(s, v)$ for all $v \in V$ in $O(E)$ time.

Now focus on computing δ_i from δ_{i-1}.

c. Prove that for $i = 2, 3, \ldots, k$, 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. Conclude that you can compute $\delta(s, v)$ for all $v \in V$ in $O(E \lg W)$ time.

22-5 Karp's minimum mean-weight cycle algorithm

Let $G = (V, E)$ be a directed graph with weight function $w : E \to \mathbb{R}$, 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 $\mu^* = \min \{\mu(c) : c \text{ is a directed cycle in } G\}$. We call a cycle c for which $\mu(c) = \mu^*$ a ***minimum mean-weight cycle***. This problem investigates an efficient algorithm for computing μ^*.

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 $\delta(s, v) = \min \{\delta_k(s, v) : 0 \le k \le n - 1\}$ for all vertices $v \in V$.

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

$$\max \left\{ \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} : 0 \le k \le n - 1 \right\} \ge 0$$

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 simple path from u to v along the cycle is x. Prove that $\delta(s, v) = \delta(s, u) + x$. (*Hint:* The weight of the simple 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

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

(*Hint:* Show how to extend a shortest path to any vertex on a minimum mean-weight cycle along the cycle to make a shortest path to the next vertex on the cycle.)

e. Show that if $\mu^* = 0$, then the minimum value of

$$\max \left\{ \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} : 0 \le k \le n - 1 \right\} \, ,$$

taken over all vertices $v \in V$, equals 0.

f. Show that if you add a constant t to the weight of each edge of G, then μ^* increases by t. Use this fact to show that μ^* equals the minimum value of

$$\max\left\{\frac{\delta_n(s,v)-\delta_k(s,v)}{n-k} : 0 \le k \le n-1\right\},$$

taken over all vertices $v \in V$.

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

22-6 Bitonic shortest paths

A sequence is ***bitonic*** if it monotonically increases and then monotonically decreases, or if by a circular shift it monotonically increases and then monotonically decreases. 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 Problem 14-3 on page 407 for the bitonic euclidean traveling-salesperson problem.)

Suppose that you are given a directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, where all edge weights are unique, and you wish to find single-source shortest paths from a source vertex s. You 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

The shortest-path problem has a long history that is nicely desribed in an article by Schrijver [400]. He credits the general idea of repeatedly executing edge relaxations to Ford [148]. Dijkstra's algorithm [116] appeared in 1959, but it contained no mention of a priority queue. The Bellman-Ford algorithm is based on separate algorithms by Bellman [45] and Ford [149]. The same algorithm is also attributed to Moore [334]. Bellman describes the relation of shortest paths to difference constraints. Lawler [276] 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 result from using min-priority queues that require integer keys and rely on the sequence of values returned by the EXTRACT-MIN calls in Dijkstra's algorithm monotonically increasing over time. 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 [436], who gives an algorithm that runs in $O(E \lg\lg V)$ time, and by Raman [375], who gives an algorithm that runs in $O\left(E + V \min\left\{(\lg V)^{1/3+\epsilon}, (\lg W)^{1/4+\epsilon}\right\}\right)$ 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 [435] gives an algorithm that runs in $O(V + E)$ time for single-source shortest paths. In contrast to the algorithms mentioned in the previous paragraph, the sequence of values returned by EXTRACT-MIN calls does not monotonically increase over time, and so this algorithm is not an implementation of Dijkstra's algorithm. Pettie and Ramachandran [357] remove the restriction of integer weights on undirected graphs. Their algorithm entails a preprocessing phase, followed by queries for specific source vertices. Preprocessing takes $O(MST(V, E) + \min\{V \lg V, V \lg\lg r\})$ time, where $MST(V, E)$ is the time to compute a minimum spanning tree and r is the ratio of the maximum edge weight to the minimum edge weight. After preprocessing, each query takes $O(E \lg \widehat{\alpha}(E, V))$ time, where $\widehat{\alpha}(E, V)$ is the inverse of Ackermann's function. (See the chapter notes for Chapter 19 for a brief discussion of Ackermann's function and its inverse.)

For graphs with negative edge weights, an algorithm due to Gabow and Tarjan [167] runs in $O(\sqrt{V}E \lg(VW))$ time, and one by Goldberg [186] runs in $O(\sqrt{V}E \lg W)$ time, where $W = \max\{|w(u, v)| : (u, v) \in E\}$. There has also been some progress based on methods that use continuous optimization and electrical flows. Cohen et al. [98] give such an algorithm, which is randomized and runs in $\widetilde{O}(E^{10/7} \lg W)$ expected time (see Problem 3-6 on page 73 for the defintion of $\widetilde{O}$-notation). There is also a pseudopolyomial-time algorithm based on fast matrix multiplication. Sankowski [394] and Yuster and Zwick [465] designed an algorithm for shortest paths that runs in $\widetilde{O}(WV^{\omega})$ time, where two $n \times n$ matrices can be multiplied in $O(n^{\omega})$ time, giving a faster algorithm than the previously mentioned algorithms for small values of W on dense graphs.

Cherkassky, Goldberg, and Radzik [89] conducted extensive experiments comparing various shortest-path algorithms. Shortest-path algorithms are widely used in real-time navigation and route-planning applications. Typically based on Dijkstra's algorithm, these algorithms use many clever ideas to be able to compute shortest paths on networks with many millions of vertices and edges in fractions of a second. Bast et al. [36] survey many of these developments.

23 All-Pairs Shortest Paths

In this chapter, we turn to the problem of finding shortest paths between all pairs of vertices in a graph. A classic application of this problem occurs in computing a table of distances between all pairs of cities for a road atlas. Classic perhaps, but not a true application of finding shortest paths between *all* pairs of vertices. After all, a road map modeled as a graph has one vertex for *every* road intersection and one edge wherever a road connects intersections. A table of intercity distances in an atlas might include distances for 100 cities, but the United States has approximately 300,000 signal-controlled intersections[1] and many more uncontrolled intersections.

A legitimate application of all-pairs shortest paths is to determine the ***diameter*** of a network: the longest of all shortest paths. If a directed graph models a communication network, with the weight of an edge indicating the time required for a message to traverse a communication link, then the diameter gives the longest possible transit time for a message in the network.

As in Chapter 22, the input is a weighted, directed graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$ that maps edges to real-valued weights. Now the goal is 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. For the all-pairs problem, the output typically takes a tabular form in which the entry in u's row and v's column is the weight of a shortest path from u to v.

You can solve an all-pairs shortest-paths problem by running a single-source shortest-paths algorithm $|V|$ times, once with each vertex as the source. If all edge weights are nonnegative, you can use Dijkstra's algorithm. If you implement the min-priority queue with a linear array, the running time is $O(V^3 + VE)$ which is $O(V^3)$. The binary min-heap implementation of the min-priority queue

---

[1] According to a report cited by U.S. Department of Transportation Federal Highway Administration, "a reasonable 'rule of thumb' is one signalized intersection per 1,000 population."

yields a running time of $O(V(V + E) \lg V)$. If $|E| = \Omega(V)$, the running time becomes $O(VE \lg V)$, which is faster than $O(V^3)$ if the graph is sparse. Alternatively, you can implement the min-priority queue with a Fibonacci heap, yielding a running time of $O(V^2 \lg V + VE)$.

If the graph contains negative-weight edges, Dijkstra's algorithm doesn't work, but you can run the slower Bellman-Ford algorithm once from each vertex. The resulting running time is $O(V^2 E)$, which on a dense graph is $O(V^4)$. This chapter shows how to guarantee a much better asymptotic running time. It also investigates the relation of the all-pairs shortest-paths problem to matrix multiplication.

Unlike the single-source algorithms, which assume an adjacency-list representation of the graph, most of the algorithms in this chapter represent the graph by an adjacency matrix. (Johnson's algorithm for sparse graphs, in Section 23.3, 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 $W = (w_{ij})$ representing the edge weights of an n-vertex directed graph $G = (V, E)$, where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j \ , \\ \text{the weight of directed edge } (i, j) & \text{if } i \neq j \text{ and } (i, j) \in E \ , \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E \ . \end{cases} \tag{23.1}$$

The graph may contain negative-weight edges, but we assume for the time being that the input graph contains no negative-weight cycles.

The tabular output of each of the all-pairs shortest-paths algorithms presented in this chapter is an $n \times n$ matrix. The (i, j) entry of the output matrix contains $\delta(i, j)$, the shortest-path weight from vertex i to vertex j, as in Chapter 22.

A full solution to the all-pairs shortest-paths problem includes not only the shortest-path weights but also a ***predecessor matrix*** $\Pi = (\pi_{ij})$, where π_{ij} is NIL if either $i = j$ or there is no path from i to j, and otherwise π_{ij} is the predecessor of j on some shortest path from i. Just as the predecessor subgraph G_π from Chapter 22 is a shortest-paths tree for a given source vertex, the subgraph induced by the ith row of the Π matrix should be a shortest-paths tree with root i. For each vertex $i \in V$, the ***predecessor subgraph*** of G for i is $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$, where

$$V_{\pi,i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\} \ ,$$
$$E_{\pi,i} = \{(\pi_{ij}, j) : j \in V_{\pi,i} - \{i\}\} \ .$$

If $G_{\pi,i}$ is a shortest-paths tree, then PRINT-ALL-PAIRS-SHORTEST-PATH on the following page, which is a modified version of the PRINT-PATH procedure from Chapter 20, prints a shortest path from vertex i to vertex j.

In order to highlight the essential features of the all-pairs algorithms in this chapter, we won't cover how to compute predecessor matrices and their properties as extensively as we dealt with predecessor subgraphs in Chapter 22. Some of the exercises cover the basics.

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

Chapter outline

Section 23.1 presents a dynamic-programming algorithm based on matrix multiplication to solve the all-pairs shortest-paths problem. The technique of "repeated squaring" yields a running time of $\Theta(V^3 \lg V)$. Section 23.2 gives another dynamic-programming algorithm, the Floyd-Warshall algorithm, which runs in $\Theta(V^3)$ time. Section 23.2 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 23.3 presents Johnson's algorithm, which solves the all-pairs shortest-paths problem in $O(V^2 \lg V + VE)$ time and is a good choice for large, sparse graphs.

Before proceeding, we need to establish some conventions for adjacency-matrix representations. First, we generally assume that the input graph $G = (V, E)$ has n vertices, so that $n = |V|$. Second, we 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}. Finally, some matrices have parenthesized superscripts, as in $L^{(r)} = \left(l_{ij}^{(r)}\right)$ or $D^{(r)} = \left(d_{ij}^{(r)}\right)$, to indicate iterates.

23.1 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 invokes an operation similar to matrix multiplication, so that the algorithm looks like repeated matrix multiplication. We'll start by developing a $\Theta(V^4)$-time algorithm for the all-pairs shortest-paths problem, and then we'll improve its running time to $\Theta(V^3 \lg V)$.

Before proceeding, let's briefly recap the steps given in Chapter 14 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.

We reserve the fourth step—constructing an optimal solution from computed information—for the exercises.

The structure of a shortest path

Let's start by characterizing the structure of an optimal solution. Lemma 22.1 tells us that all subpaths of a shortest path are shortest paths. Consider a shortest path p from vertex i to vertex j, and suppose that p contains at most r edges. Assuming that there are no negative-weight cycles, r is finite. If $i = j$, then p has weight 0 and no edges. If vertices i and j are distinct, then decompose path p into $i \overset{p'}{\rightsquigarrow} k \to j$, where path p' now contains at most $r-1$ edges. Lemma 22.1 says that 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 $l_{ij}^{(r)}$ be the minimum weight of any path from vertex i to vertex j that contains at most r edges. When $r = 0$, there is a shortest path from i to j with no edges if and only if $i = j$, yielding

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j\ , \\ \infty & \text{if } i \neq j\ . \end{cases} \tag{23.2}$$

For $r \geq 1$, one way to achieve a minimum-weight path from i to j with at most r edges is by taking a path containing at most $r-1$ edges, so that $l_{ij}^{(r)} = l_{ij}^{(r-1)}$. Another way is by taking a path of at most $r-1$ edges from i to some vertex k and then taking the edge (k,j), so that $l_{ij}^{(r)} = l_{ik}^{(r-1)} + w(k,j)$. Therefore, to examine paths from i to j consisting of at most r edges, try all possible predecessors k of j, giving the recursive definition

$$\begin{aligned} l_{ij}^{(r)} &= \min\left(l_{ij}^{(r-1)},\ \min\left\{l_{ik}^{(r-1)} + w_{kj} : 1 \leq k \leq n\right\}\right) \\ &= \min\left\{l_{ik}^{(r-1)} + w_{kj} : 1 \leq k \leq n\right\}\ . \end{aligned} \tag{23.3}$$

The last equality follows from the observation that $w_{jj} = 0$ for all j.

What are the actual shortest-path weights $\delta(i,j)$? If the graph contains no negative-weight cycles, then whenever $\delta(i,j) < \infty$, there is a shortest path from vertex i to vertex j that is simple. (A path p from i to j that is not simple contains a cycle. Since each cycle's weight is nonnegative, removing all cycles from the path leaves a simple path with weight no greater than p's weight.) Because any simple path 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

$$\delta(i,j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots . \tag{23.4}$$

Computing the shortest-path weights bottom up

Taking as input the matrix $W = (w_{ij})$, let's see how to compute a series of matrices $L^{(0)}, L^{(1)}, \ldots, L^{(n-1)}$, where $L^{(r)} = \left(l_{ij}^{(r)}\right)$ for $r = 0, 1, \ldots, n-1$. The initial matrix is $L^{(0)}$ given by equation (23.2). The final matrix $L^{(n-1)}$ contains the actual shortest-path weights.

The heart of the algorithm is the procedure EXTEND-SHORTEST-PATHS, which implements equation (23.3) for all i and j. The four inputs are the matrix $L^{(r-1)}$ computed so far; the edge-weight matrix W; the output matrix $L^{(r)}$, which will hold the computed result and whose elements are all initialized to ∞ before invoking the procedure; and the number n of vertices. The superscripts r and $r-1$ help to make the correspondence of the pseudocode with equation (23.3) plain, but they play no actual role in the pseudocode. The procedure extends the shortest paths computed so far by one more edge, producing the matrix $L^{(r)}$ of shortest-path weights from the matrix $L^{(r-1)}$ computed so far. Its running time is $\Theta(n^3)$ due to the three nested **for** loops.

```
EXTEND-SHORTEST-PATHS(L^(r-1), W, L^(r), n)
1  // Assume that the elements of L^(r) are initialized to ∞.
2  for i = 1 to n
3      for j = 1 to n
4          for k = 1 to n
5              l_ij^(r) = min{l_ij^(r), l_ik^(r-1) + w_kj}
```

Let's now understand the relation of this computation to matrix multiplication. Consider how to compute the matrix product $C = A \cdot B$ of two $n \times n$ matrices A and B. The straightforward method used by MATRIX-MULTIPLY on page 81 uses a triply nested loop to implement equation (4.1), which we repeat here for convenience:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} , \tag{23.5}$$

for $i, j = 1, 2, \ldots, n$. Now make the substitutions

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

in equation (23.3). You get equation (23.5)! Making these changes to EXTEND-SHORTEST-PATHS, and also replacing ∞ (the identity for min) by 0 (the identity for $+$), yields the procedure MATRIX-MULTIPLY. We can see that the procedure EXTEND-SHORTEST-PATHS$(L^{(r-1)}, W, L^{(r)}, n)$ computes the matrix "product" $L^{(r)} = L^{(r-1)} \cdot W$ using this unusual definition of matrix multiplication.[2]

Thus, we can solve the all-pairs shortest-paths problem by repeatedly multiplying matrices. Each step extends the shortest-path weights computed so far by one more edge using EXTEND-SHORTEST-PATHS$(L^{(r-1)}, W, L^{(r)}, n)$ to perform the matrix multiplication. Starting with the matrix $L^{(0)}$, we produce the following sequence of $n-1$ matrices corresponding to powers of W:

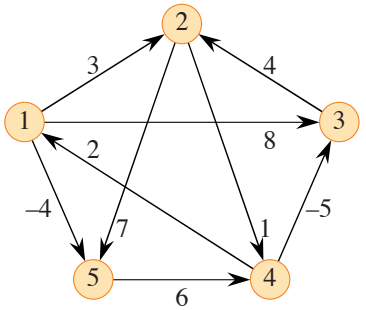
$$
\begin{aligned}
L^{(1)} &= L^{(0)} \cdot W &&= W^1\,,\\
L^{(2)} &= L^{(1)} \cdot W &&= W^2\,,\\
L^{(3)} &= L^{(2)} \cdot W &&= W^3\,,\\
&\;\;\vdots\\
L^{(n-1)} &= L^{(n-2)} \cdot W &&= W^{n-1}\,.
\end{aligned}
$$

At the end, the matrix $L^{(n-1)} = W^{n-1}$ contains the shortest-path weights.

The procedure SLOW-APSP on the next page computes this sequence in $\Theta(n^4)$ time. The procedure takes the $n \times n$ matrices W and $L^{(0)}$ as inputs, along with n. Figure 23.1 illustrates its operation. The pseudocode uses two $n \times n$ matrices L and M to store powers of W, computing $M = L \cdot W$ on each iteration. Line 2 initializes $L = L^{(0)}$. For each iteration r, line 4 initializes $M = \infty$, where ∞ in this context is a matrix of scalar ∞ values. The rth iteration starts with the invariant $L = L^{(r-1)} = W^{r-1}$. Line 6 computes $M = L \cdot W = L^{(r-1)} \cdot W = W^{r-1} \cdot W = W^r = L^{(r)}$ so that the invariant can be restored for the next iteration by line 7, which sets $L = M$. At the end, the matrix $L = L^{(n-1)} = W^{n-1}$ of shortest-path weights is returned. The assignments to $n \times n$ matrices in lines 2, 4, and 7 implicitly run doubly nested loops that take $\Theta(n^2)$ time for each assignment.

---

[2] An algebraic ***semiring*** contains operations $\oplus$, which is commutative with identity $I_\oplus$, and $\otimes$, with identity $I_\otimes$, where $\otimes$ distributes over $\oplus$ on both the left and right, and where $I_\oplus \otimes x = x \otimes I_\oplus = I_\oplus$ for all x. Standard matrix multiplication, as in MATRIX-MULTIPLY, uses the semiring with $+$ for $\oplus$, $\cdot$ for $\otimes$, 0 for $I_\oplus$, and 1 for $I_\otimes$. The procedure EXTEND-SHORTEST-PATHS uses another semiring, known as the ***tropical semiring***, with min for $\oplus$, $+$ for $\otimes$, ∞ for $I_\oplus$, and 0 for $I_\otimes$.



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

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

Figure 23.1 A directed graph and the sequence of matrices $L^{(r)}$ computed by SLOW-APSP. You might want to verify that $L^{(5)}$, defined as $L^{(4)} \cdot W$, equals $L^{(4)}$, and thus $L^{(r)} = L^{(4)}$ for all $r \geq 4$.

The $n - 1$ invocations of EXTEND-SHORTEST-PATHS, each of which takes $\Theta(n^3)$ time, dominate the computation, yielding a total running time of $\Theta(n^4)$.

```
SLOW-APSP(W, L^(0), n)
1  let L = (l_ij) and M = (m_ij) be new n × n matrices
2  L = L^(0)
3  for r = 1 to n − 1
4       M = ∞        // initialize M
5       // Compute the matrix "product" M = L · W.
6       EXTEND-SHORTEST-PATHS(L, W, M, n)
7       L = M
8  return L
```

Improving the running time

Bear in mind that the goal is not to compute *all* the $L^{(r)}$ matrices: only the matrix $L^{(n-1)}$ matters. Recall that in the absence of negative-weight cycles, equation (23.4) implies $L^{(r)} = L^{(n-1)}$ for all integers $r \geq n - 1$. Just as traditional matrix multiplication is associative, so is matrix multiplication defined by the EXTEND-SHORTEST-PATHS procedure (see Exercise 23.1-4). In fact, we can compute $L^{(n-1)}$ with only $\lceil \lg(n-1) \rceil$ matrix products by using the technique of ***repeated squaring***:

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

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

The procedure FASTER-APSP implements this idea. It takes just the $n \times n$ matrix W and the size n as inputs. Each iteration of the **while** loop of lines 4–8 starts with the invariant $L = W^r$, which it squares using EXTEND-SHORTEST-PATHS to obtain the matrix $M = L^2 = (W^r)^2 = W^{2r}$. At the end of each iteration, the value of r doubles, and L for the next iteration becomes M, restoring the invariant. Upon exiting the loop when $r \geq n-1$, the procedure returns $L = W^r = L^{(r)} = L^{(n-1)}$ by equation (23.4). As in SLOW-APSP, the assignments to $n \times n$ matrices in lines 2, 5, and 8 implicitly run doubly nested loops, taking $\Theta(n^2)$ time for each assignment.

```
FASTER-APSP(W, n)
1  let L and M be new n × n matrices
2  L = W
3  r = 1
4  while r < n − 1
5      M = ∞        // initialize M
6      EXTEND-SHORTEST-PATHS(L, L, M, n)      // compute M = L^2
7      r = 2r
8      L = M        // ready for the next iteration
9  return L
```

Because each of the $\lceil \lg(n-1) \rceil$ matrix products takes $\Theta(n^3)$ time, FASTER-APSP runs in $\Theta(n^3 \lg n)$ time. The code is tight, containing no elaborate data structures, and the constant hidden in the Θ-notation is therefore small.

Exercises

23.1-1

Run SLOW-APSP on the weighted, directed graph of Figure 23.2, showing the matrices that result for each iteration of the loop. Then do the same for FASTER-APSP.

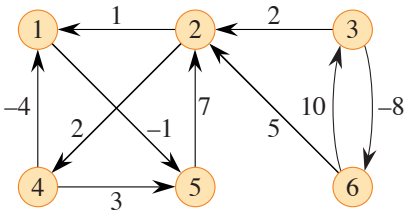


Figure 23.2 A weighted, directed graph for use in Exercises 23.1-1, 23.2-1, and 23.3-1.

23.1-2

Why is it convenient for both SLOW-APSP and FASTER-APSP that $w_{ii} = 0$ for $i = 1, 2, \ldots, n$?

23.1-3

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?

23.1-4

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

23.1-5

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 22.1).

23.1-6

Argue that we don't need the matrix M in SLOW-APSP because by substituting L for M and leaving out the initialization of M, the code still works correctly. (*Hint:* Relate line 5 of EXTEND-SHORTEST-PATHS to RELAX on page 610.) Do we need the matrix M in FASTER-APSP?

23.1-7
Suppose that you also want to compute the vertices on shortest paths in the algorithms of this section. Show how to compute the predecessor matrix Π from the completed matrix L of shortest-path weights in $O(n^3)$ time.

23.1-8
You can also compute the vertices on shortest paths along with computing the shortest-path weights. Define $\pi_{ij}^{(r)}$ as the predecessor of vertex j on any minimum-weight path from vertex i to vertex j that contains at most r edges. Modify the EXTEND-SHORTEST-PATHS and SLOW-APSP procedures to compute the matrices $\Pi^{(1)}, \Pi^{(2)}, \ldots, \Pi^{(n-1)}$ as they compute the matrices $L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$.

23.1-9
Modify FASTER-APSP so that it can determine whether the graph contains a negative-weight cycle.

23.1-10
Give an efficient algorithm to find the length (number of edges) of a minimum-length negative-weight cycle in a graph.

23.2 The Floyd-Warshall algorithm

Having already seen one dynamic-programming solution to the all-pairs shortest-paths problem, in this section we'll see another: the ***Floyd-Warshall algorithm***, which runs in $\Theta(V^3)$ time. As before, negative-weight edges may be present, but not negative-weight cycles. As in Section 23.1, we develop the algorithm by following the dynamic-programming process. After studying the resulting algorithm, we 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 characterize the structure of a shortest path differently from how we characterized it in Section 23.1. The Floyd-Warshall 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 relies on the following observation. Numbering the vertices of G by $V = \{1, 2, \ldots, n\}$, take a subset $\{1, 2, \ldots, k\}$ of vertices for some $1 \le k \le n$. 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

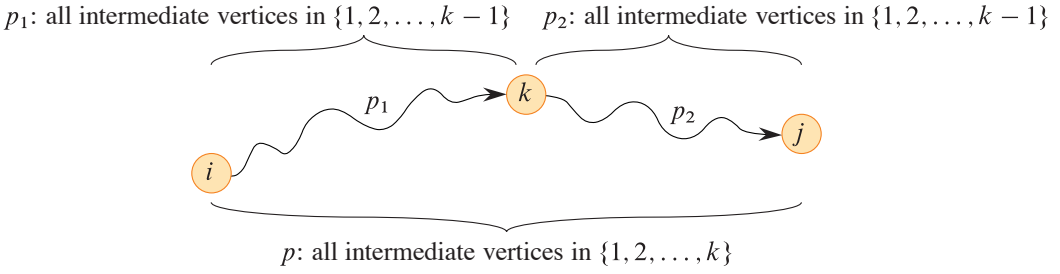


Figure 23.3 Optimal substructure used by the Floyd-Warshall algorithm. Path p is a shortest path from vertex i to vertex j, and k is the highest-numbered intermediate vertex of p. Path p_1, the portion of path p from vertex i to vertex k, has all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. The same holds for path p_2 from vertex k to vertex j.

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 details of the relationship depend on whether k is an intermediate vertex of path p or not.

- If k is not an intermediate vertex of path p, then all intermediate vertices of path p belong to 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 decompose p into $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$, as Figure 23.3 illustrates. By Lemma 22.1, p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k\}$. In fact, we can make a slightly stronger statement. Because vertex k is not an *intermediate* vertex of path p_1, all intermediate vertices of p_1 belong to the set $\{1, 2, \ldots, k-1\}$. Therefore p_1 is a shortest path from i to k with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Likewise, p_2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$.

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

The above observations suggest a recursive formulation of shortest-path estimates that differs from the one in Section 23.1. Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices belong to 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 $d_{ij}^{(0)} = w_{ij}$. Following the above discussion, define $d_{ij}^{(k)}$ recursively by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\} & \text{if } k \geq 1\,. \end{cases} \tag{23.6}$$

Because for any path, all intermediate vertices belong to 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 (23.6), the bottom-up procedure FLOYD-WARSHALL computes 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 (23.1). The procedure returns the matrix $D^{(n)}$ of shortest-path weights. Figure 23.4 shows the matrices $D^{(k)}$ computed by the Floyd-Warshall algorithm for the graph in Figure 23.1.

```
FLOYD-WARSHALL(W, n)
1  D^(0) = W
2  for k = 1 to n
3       let D^(k) = (d_ij^(k)) be a new n × n matrix
4       for i = 1 to n
5            for j = 1 to n
6                 d_ij^(k) = min{d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1)}
7  return D^(n)
```

The running time of the Floyd-Warshall algorithm is determined by the triply nested **for** loops of lines 2–6. Because each execution of line 6 takes $O(1)$ time, the algorithm runs in $\Theta(n^3)$ time. As in the final algorithm in Section 23.1, the code is tight, with no elaborate data structures, and so the constant hidden in the Θ-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 Π from the D matrix. Exercise 23.1-7 asks you to implement this method so that it runs in $O(n^3)$ time. Given the predecessor matrix Π, the PRINT-ALL-PAIRS-SHORTEST-PATH procedure prints the vertices on a given shortest path.

Alternatively, the predecessor matrix Π can be computed while the algorithm computes the matrices $D^{(0)}, D^{(1)}, \ldots, D^{(n)}$. Specifically, compute a sequence of

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

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

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

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

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

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

Figure 23.4 The sequence of matrices $D^{(k)}$ and $\Pi^{(k)}$ computed by the Floyd-Warshall algorithm for the graph in Figure 23.1.

matrices $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$, where $\Pi = \Pi^{(n)}$ and $\pi_{ij}^{(k)}$ is the predecessor of vertex j on a shortest path from vertex i with all intermediate vertices in the set $\{1, 2, \ldots, k\}$.

Here's a recursive formulation of $\pi_{ij}^{(k)}$. When $k = 0$, a shortest path from i to j has no intermediate vertices at all, and so

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty \,, \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty \,. \end{cases} \tag{23.7}$$

For $k \geq 1$, if the path has k as an intermediate vertex, so that it is $i \rightsquigarrow k \rightsquigarrow j$ where $k \neq j$, then choose as the predecessor of j on this path the same vertex as the predecessor of j chosen on a shortest path from k with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Otherwise, when the path from i to j does not have k as an intermediate vertex, choose the same predecessor of j as on a shortest path from i with all intermediate vertices in the set $\{1, 2, \ldots, k-1\}$. Formally, for $k \geq 1$,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \ (k \text{ is an intermediate vertex}) \,, \\ \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \ (k \text{ is not an intermediate vertex}) \,. \end{cases} \tag{23.8}$$

Exercise 23.2-3 asks you to show how to incorporate the $\Pi^{(k)}$ matrix computations into the FLOYD-WARSHALL procedure. Figure 23.4 shows the sequence of $\Pi^{(k)}$ matrices that the resulting algorithm computes for the graph of Figure 23.1. 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. Exercise 23.2-7 asks for yet another way to reconstruct shortest paths.

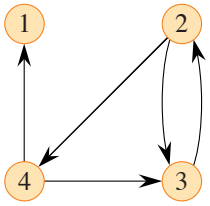
Transitive closure of a directed graph

Given a directed graph $G = (V, E)$ with vertex set $V = \{1, 2, \ldots, n\}$, you might wish to determine simply whether G contains a path from i to j for all vertex pairs $i, j \in V$, without regard to edge weights. We define the ***transitive closure*** of G 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, you get $d_{ij} < n$. Otherwise, you get $d_{ij} = \infty$.

There is another, similar way to compute the transitive closure of G in $\Theta(n^3)$ time, which can save time and space in practice. This method substitutes the logical operations $\vee$ (logical OR) and $\wedge$ (logical AND) for the arithmetic operations min and $+$ in the Floyd-Warshall algorithm. For $i, j, k = 1, 2, \ldots, n$, 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. To construct the transitive closure $G^* = (V, E^*)$, put edge (i, j) into E^* if and only if $t_{ij}^{(n)} = 1$. A recursive definition of $t_{ij}^{(k)}$, analogous to recurrence (23.6), is



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

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

Figure 23.5 A directed graph and the matrices $T^{(k)}$ computed by the transitive-closure algorithm.

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

and for $k \geq 1$,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)\,. \tag{23.9}$$

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

```
TRANSITIVE-CLOSURE(G, n)
 1  let T^(0) = (t_ij^(0)) be a new n × n matrix
 2  for i = 1 to n
 3      for j = 1 to n
 4          if i == j or (i, j) ∈ G.E
 5              t_ij^(0) = 1
 6          else t_ij^(0) = 0
 7  for k = 1 to n
 8      let T^(k) = (t_ij^(k)) be a new n × n matrix
 9      for i = 1 to n
10          for j = 1 to n
11              t_ij^(k) = t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
12  return T^(n)
```

Figure 23.5 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 Floyd-Warshall algorithm's by a factor corresponding to the size of a word of computer storage.

Exercises

23.2-1

Run the Floyd-Warshall algorithm on the weighted, directed graph of Figure 23.2. Show the matrix $D^{(k)}$ that results for each iteration of the outer loop.

23.2-2

Show how to compute the transitive closure using the technique of Section 23.1.

23.2-3

Modify the FLOYD-WARSHALL procedure to compute the $\Pi^{(k)}$ matrices according to equations (23.7) and (23.8). 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 22.16.)

23.2-4

As it appears on page 657, the Floyd-Warshall algorithm requires $\Theta(n^3)$ space, since it creates $d_{ij}^{(k)}$ for $i, j, k = 1, 2, \ldots, n$. Show that the procedure FLOYD-WARSHALL$'$, which simply drops all the superscripts, is correct, and thus only $\Theta(n^2)$ space is required.

```
FLOYD-WARSHALL'(W, n)
1  D = W
2  for k = 1 to n
3      for i = 1 to n
4          for j = 1 to n
5              d_ij = min {d_ij, d_ik + d_kj}
6  return D
```

23.2-5

Consider the following change to how equation (23.8) handles equality:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \geq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \ (k \text{ is an intermediate vertex}) , \\ \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \ (k \text{ is not an intermediate vertex}) . \end{cases}$$

Is this alternative definition of the predecessor matrix Π correct?

23.2-6

Show how to use the output of the Floyd-Warshall algorithm to detect the presence of a negative-weight cycle.

23.2-7

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 lie 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 Φ like the s table in the matrix-chain multiplication problem of Section 14.2?

23.2-8

Give an $O(VE)$-time algorithm for computing the transitive closure of a directed graph $G = (V, E)$. Assume that $|V| = O(E)$ and that the graph is represented with adjacency lists.

23.2-9

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

23.3 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 faster 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 Chapter 22 describes.

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, Dijkstra's algorithm can find shortest paths between all pairs of vertices by running it 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, first compute a new set of nonnegative edge weights so

that Dijkstra's algorithm applies. The new set of edge weights $\hat{w}$ 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 $\hat{w}$.
2. For all edges (u, v), the new weight $\hat{w}(u, v)$ is nonnegative.

As we'll see in a moment, preprocessing G to determine the new weight function $\hat{w}$ takes $O(VE)$ time.

Preserving shortest paths by reweighting

The following lemma shows how to reweight the edges to satisfy the first property above. We use δ to denote shortest-path weights derived from weight function w and $\hat{\delta}$ to denote shortest-path weights derived from weight function $\hat{w}$.

Lemma 23.1 (Reweighting does not change shortest paths)
Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, let $h : V \to \mathbb{R}$ be any function mapping vertices to real numbers. For each edge $(u, v) \in E$, define

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v) . \tag{23.10}$$

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 $\hat{w}$. That is, $w(p) = \delta(v_0, v_k)$ if and only if $\hat{w}(p) = \hat{\delta}(v_0, v_k)$. Furthermore, G has a negative-weight cycle using weight function w if and only if G has a negative-weight cycle using weight function $\hat{w}$.

Proof We start by showing that

$$\hat{w}(p) = w(p) + h(v_0) - h(v_k) . \tag{23.11}$$

We have

$$\begin{aligned}
\hat{w}(p) &= \sum_{i=1}^{k} \hat{w}(v_{i-1}, v_i) \\
&= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\
&= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) \qquad \text{(because the sum telescopes)} \\
&= w(p) + h(v_0) - h(v_k) .
\end{aligned}$$

Therefore, any path p from v_0 to v_k has $\hat{w}(p) = w(p) + h(v_0) - h(v_k)$. Because $h(v_0)$ and $h(v_k)$ do not depend on the path, if one path from v_0 to v_k is shorter than another using weight function w, then it is also shorter using $\hat{w}$. Thus, $w(p) = \delta(v_0, v_k)$ if and only if $\hat{w}(p) = \hat{\delta}(v_0, v_k)$.

Finally, we show that G has a negative-weight cycle using weight function w if and only if G has a negative-weight cycle using weight function $\hat{w}$. Consider any cycle $c = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = v_k$. By equation (23.11),

$$\begin{aligned} \hat{w}(c) &= w(c) + h(v_0) - h(v_k) \\ &= w(c) , \end{aligned}$$

and thus c has negative weight using w if and only if it has negative weight using $\hat{w}$. ■

Producing nonnegative weights by reweighting

Our next goal is to ensure that the second property holds: $\hat{w}(u, v)$ must be nonnegative for all edges $(u, v) \in E$. Given a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$, we'll see how to make a new graph $G' = (V', E')$, where $V' = V \cup \{s\}$ for some new vertex $s \notin V$ and $E' = E \cup \{(s, v) : v \in V\}$. To incorporate the new vertex s, extend the weight function w so that $w(s, v) = 0$ for all $v \in V$. Since no edges enter s, no shortest paths in G', other than those with source s, contain s. Moreover, G' has no negative-weight cycles if and only if G has no negative-weight cycles. Figure 23.6(a) shows the graph G' corresponding to the graph G of Figure 23.1.

Now suppose that G and G' have no negative-weight cycles. Define the function $h(v) = \delta(s, v)$ for all $v \in V'$. By the triangle inequality (Lemma 22.10 on page 633), we have $h(v) \leq h(u) + w(u, v)$ for all edges $(u, v) \in E'$. Thus, by defining reweighted edge weights $\hat{w}$ according to equation (23.10), we have $\hat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$, thereby satisfying the second property. Figure 23.6(b) shows the graph G' from Figure 23.6(a) with reweighted edges.

Computing all-pairs shortest paths

Johnson's algorithm to compute all-pairs shortest paths uses the Bellman-Ford algorithm (Section 22.1) and Dijkstra's algorithm (Section 22.3) as subroutines. The pseudocode appears in the procedure JOHNSON on page 666. It assumes implicitly that the edges are stored in adjacency lists. The algorithm returns the usual $|V| \times |V|$ matrix $D = (d_{ij})$, where $d_{ij} = \delta(i, j)$, or it reports that the input graph contains a negative-weight cycle. As is typical for an all-pairs shortest-paths algorithm, it assumes that the vertices are numbered from 1 to $|V|$.

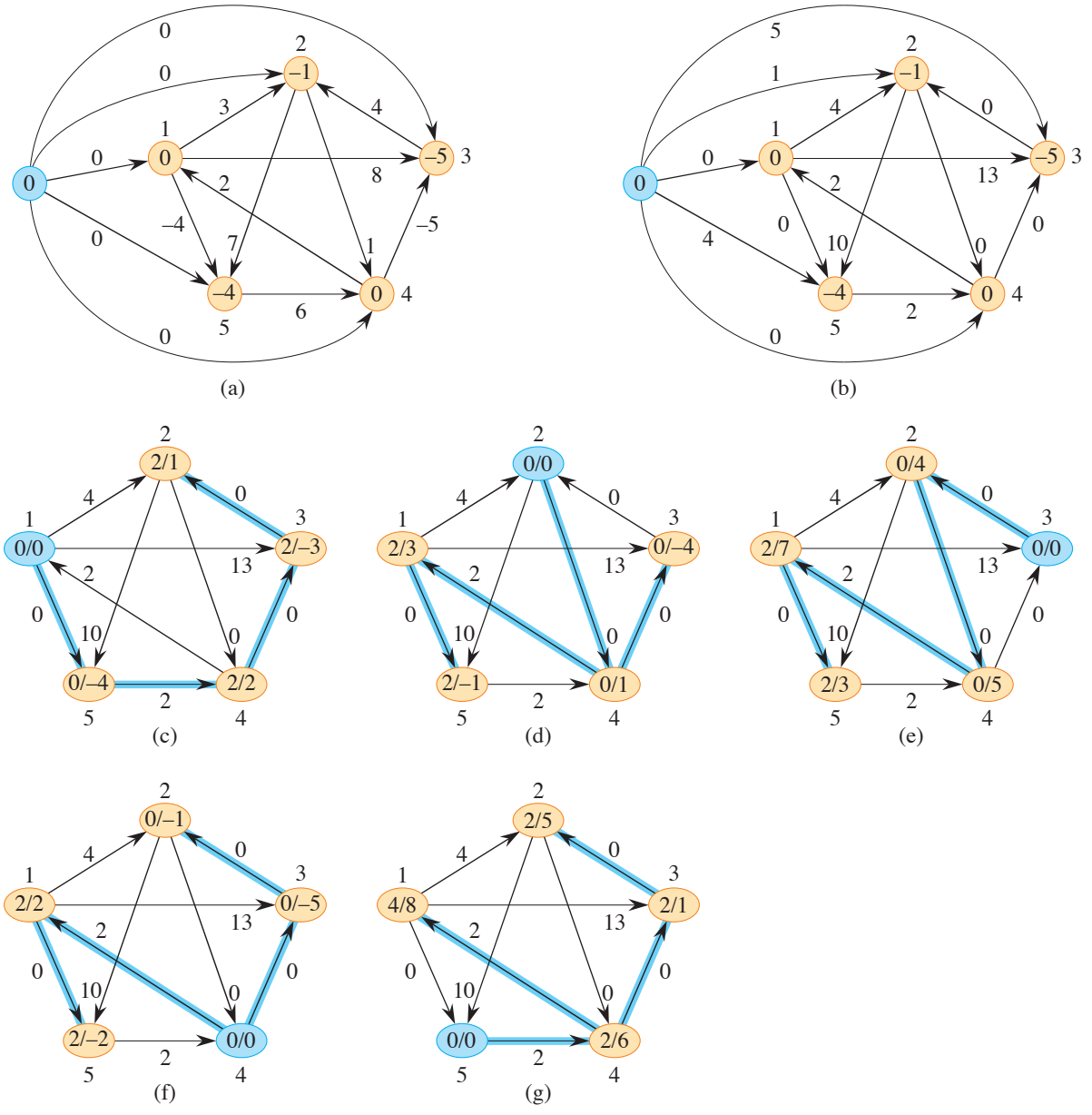


Figure 23.6 Johnson's all-pairs shortest-paths algorithm run on the graph of Figure 23.1. Vertex numbers appear outside the vertices. **(a)** The graph G' with the original weight function w. The new vertex s is blue. Within each vertex v is $h(v) = \delta(s, v)$. **(b)** After reweighting each edge (u, v) with weight function $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$. **(c)–(g)** The result of running Dijkstra's algorithm on each vertex of G using weight function $\hat{w}$. In each part, the source vertex u is blue, and blue edges belong to the shortest-paths tree computed by the algorithm. Within each vertex v are the values $\hat{\delta}(u, v)$ and $\delta(u, v)$, separated by a slash. The value $d_{uv} = \delta(u, v)$ is equal to $\hat{\delta}(u, v) + h(v) - h(u)$.

```
JOHNSON(G, w)
 1  compute G', where G'.V = G.V ∪ {s},
        G'.E = G.E ∪ {(s, v) : v ∈ G.V}, and
        w(s, v) = 0 for all v ∈ G.V
 2  if BELLMAN-FORD(G', w, s) == FALSE
 3      print "the input graph contains a negative-weight cycle"
 4  else for each vertex v ∈ G'.V
 5          set h(v) to the value of δ(s, v)
                computed by the Bellman-Ford algorithm
 6      for each edge (u, v) ∈ G'.E
 7          ŵ(u, v) = w(u, v) + h(u) − h(v)
 8      let D = (d_uv) be a new n × n matrix
 9      for each vertex u ∈ G.V
10          run DIJKSTRA(G, ŵ, u) to compute δ̂(u, v) for all v ∈ G.V
11          for each vertex v ∈ G.V
12              d_uv = δ̂(u, v) + h(v) − h(u)
13      return D
```

The JOHNSON procedure simply performs the actions specified earlier. Line 1 produces G'. Line 2 runs the Bellman-Ford algorithm on G' with weight function w and source vertex s. If G', and hence G, contains a negative-weight cycle, line 3 reports the problem. Lines 4–12 assume that G' contains no negative-weight cycles. Lines 4–5 set $h(v)$ to the shortest-path weight $\delta(s, v)$ computed by the Bellman-Ford algorithm for all $v \in V'$. Lines 6–7 compute the new weights $\widehat{w}$. For each pair of vertices $u, v \in V$, the **for** loop of lines 9–12 computes the shortest-path weight $\widehat{\delta}(u, v)$ by calling Dijkstra's algorithm once from each vertex in V. Line 12 stores in matrix entry d_{uv} the correct shortest-path weight $\delta(u, v)$, calculated using equation (23.11). Finally, line 13 returns the completed D matrix. Figure 23.6 depicts the execution of Johnson's algorithm.

If the min-priority queue in Dijkstra's algorithm is implemented by a Fibonacci heap, Johnson's algorithm runs in $O(V^2 \lg V + VE)$ time. The simpler binary min-heap implementation yields a running time of $O(VE \lg V)$, which is still asymptotically faster than the Floyd-Warshall algorithm if the graph is sparse.

Exercises

23.3-1

Use Johnson's algorithm to find the shortest paths between all pairs of vertices in the graph of Figure 23.2. Show the values of h and $\widehat{w}$ computed by the algorithm.

23.3-2
What is the purpose of adding the new vertex s to V, yielding V'?

23.3-3
Suppose that $w(u, v) \geq 0$ for all edges $(u, v) \in E$. What is the relationship between the weight functions w and $\hat{w}$?

23.3-4
Professor Greenstreet claims that there is a simpler way to reweight edges than the method used in Johnson's algorithm. Letting $w^* = \min\{w(u, v) : (u, v) \in E\}$, just define $\hat{w}(u, v) = w(u, v) - w^*$ for all edges $(u, v) \in E$. What is wrong with the professor's method of reweighting?

23.3-5
Show that if G contains a 0-weight cycle c, then $\hat{w}(u, v) = 0$ for every edge (u, v) in c.

23.3-6
Professor Michener claims that there is no need to create a new source vertex in line 1 of JOHNSON. He suggests using $G' = G$ instead and letting s be any vertex. Give an example of a weighted, directed graph G for which incorporating the professor's idea into JOHNSON causes incorrect answers. Assume that $\infty - \infty$ is undefined, and in particular, it is not 0. Then show that if G is strongly connected (every vertex is reachable from every other vertex), the results returned by JOHNSON with the professor's modification are correct.

Problems

23-1 Transitive closure of a dynamic graph
You wish to maintain the transitive closure of a directed graph $G = (V, E)$ as you insert edges into E. That is, after inserting an edge, you update the transitive closure of the edges inserted so far. Start with G having no edges initially, and represent the transitive closure by a boolean matrix.

a. Show how to update the transitive closure $G^* = (V, E^*)$ of a graph $G = (V, E)$ in $O(V^2)$ time when a new edge is added to G.

b. Give an example of a graph G and an edge e such that $\Omega(V^2)$ time is required to update the transitive closure after inserting e into G, no matter what algorithm is used.

c. Give an algorithm for updating the transitive closure as edges are inserted into the graph. For any sequence of r insertions, your algorithm should run in time $\sum_{i=1}^{r} t_i = O(V^3)$, where t_i is the time to update the transitive closure upon inserting the ith edge. Prove that your algorithm attains this time bound.

23-2 Shortest paths in ϵ-dense graphs

A graph $G = (V, E)$ is ***ϵ-dense*** if $|E| = \Theta(V^{1+\epsilon})$ for some constant ϵ in the range $0 < \epsilon \leq 1$. d-ary min-heaps (see Problem 6-2 on page 179) provide a way to match the running times of Fibonacci-heap-based shortest-path algorithms on ϵ-dense graphs without using as complicated a data structure.

a. What are the asymptotic running times for the operations INSERT, EXTRACT-MIN, and DECREASE-KEY, as a function of d and the number n of elements in a d-ary min-heap? What are these running times if you choose $d = \Theta(n^\alpha)$ for some constant $0 < \alpha \leq 1$? Compare these running times to the amortized costs of these operations for a Fibonacci heap.

b. Show how to compute shortest paths from a single source on an ϵ-dense directed graph $G = (V, E)$ with no negative-weight edges in $O(E)$ time. (*Hint:* Pick d as a function of ϵ.)

c. Show how to solve the all-pairs shortest-paths problem on an ϵ-dense directed graph $G = (V, E)$ with no negative-weight edges in $O(VE)$ time.

d. Show how to solve the all-pairs shortest-paths problem in $O(VE)$ time on an ϵ-dense directed graph $G = (V, E)$ that may have negative-weight edges but has no negative-weight cycles.

Chapter notes

Lawler [276] has a good discussion of the all-pairs shortest-paths problem. He attributes the matrix-multiplication algorithm to the folklore. The Floyd-Warshall algorithm is due to Floyd [144], who based it on a theorem of Warshall [450] that describes how to compute the transitive closure of boolean matrices. Johnson's algorithm is taken from [238].

Several researchers have given improved algorithms for computing shortest paths via matrix multiplication. Fredman [153] shows how to solve the all-pairs shortest paths problem using $O(V^{5/2})$ comparisons between sums of edge weights and obtains an algorithm that runs in $O(V^3(\lg\lg V / \lg V)^{1/3})$ time, which is slightly better than the running time of the Floyd-Warshall algorithm. This bound

has been improved several times, and the fastest algorithm is now by Williams [457], with a running time of $O(V^3/2^{\Omega(\lg^{1/2} V)})$.

Another line of research demonstrates how to apply algorithms for fast matrix multiplication (see the chapter notes for Chapter 4) to the all-pairs shortest paths problem. Let $O(n^\omega)$ be the running time of the fastest algorithm for multiplying two $n \times n$ matrices. Galil and Margalit [170, 171] and Seidel [403] designed algorithms that solve the all-pairs shortest paths problem in undirected, unweighted graphs in $(V^\omega p(V))$ time, where $p(n)$ denotes a particular function that is polylogarithmically bounded in n. In dense graphs, these algorithms are faster than the $O(VE)$ time needed to perform $|V|$ breadth-first searches. Several researchers have extended these results to give algorithms for solving the all-pairs shortest paths problem in undirected graphs in which the edge weights are integers in the range $\{1, 2, \ldots, W\}$. The asymptotically fastest such algorithm, by Shoshan and Zwick [410], runs in $O(WV^\omega p(VW))$ time. In directed graphs, the best algorithm to date is due to Zwick [467] and runs in $\widetilde{O}(W^{1/(4-\omega)} V^{2+1/(4-\omega)})$ time.

Karger, Koller, and Phillips [244] and independently McGeoch [320] have given a time bound that depends on E^*, the set of edges in E that participate in some shortest path. Given a graph with nonnegative edge weights, their algorithms run in $O(VE^* + V^2 \lg V)$ time and improve upon running Dijkstra's algorithm $|V|$ times when $|E^*| = o(E)$. Pettie [355] uses an approach based on component hierarchies to achieve a running time of $O(VE + V^2 \lg\lg V)$, and the same running time is also achieved by Hagerup [205].

Baswana, Hariharan, and Sen [37] examined decremental algorithms, which allow a sequence of intermixed edge deletions and queries, for maintaining all-pairs shortest paths and transitive-closure information. When a path exists, their randomized transitive-closure algorithm can fail to report it with probability $1/n^c$ for an arbitrary $c > 0$. The query times are $O(1)$ with high probability. For transitive closure, the amortized time for each update is $O(V^{4/3} \lg^{1/3} V)$. By comparison, Problem 23-1, in which edges are inserted, asks for an incremental algorithm. For all-pairs shortest paths, the update times depend on the queries. For queries just giving the shortest-path weights, the amortized time per update is $O(V^3/E \lg^2 V)$. To report the actual shortest path, the amortized update time is $\min\left\{O(V^{3/2}\sqrt{\lg V}), O(V^3/E \lg^2 V)\right\}$. Demetrescu and Italiano [111] showed how to handle update and query operations when edges are both inserted and deleted, as long as the range of edge weights is bounded.

Aho, Hopcroft, and Ullman [5] defined an algebraic structure known as a "closed semiring," which serves as a general framework for solving path problems in directed graphs. Both the Floyd-Warshall algorithm and the transitive-closure algorithm from Section 23.2 are instantiations of an all-pairs algorithm based on closed semirings. Maggs and Plotkin [309] showed how to find minimum spanning trees using a closed semiring.

24 Maximum Flow

Just as you can model a road map as a directed graph in order to find the shortest path from one point to another, you can also interpret a directed graph as a "flow network" and use it to answer questions about material flows. Imagine a material coursing through a system from a source, where the material is produced, to a sink, where it is consumed. The source produces the material at some steady rate, and the sink consumes the material at the same rate. The "flow" of the material at any point in the system is intuitively the rate at which the material moves. Flow networks can model many problems, including liquids flowing through pipes, parts through assembly lines, current through electrical networks, and information through communication networks.

You can think of each directed edge in a flow network as a conduit for the material. Each conduit has a stated capacity, given as a maximum rate at which the material can flow through the conduit, such as 200 gallons of liquid per hour through a pipe or 20 amperes of electrical current through a wire. Vertices are conduit junctions, and other than the source and sink, material flows through the vertices without collecting in them. In other words, the rate at which material enters a vertex must equal the rate at which it leaves the vertex. We call this property "flow conservation," and it is equivalent to Kirchhoff's current law when the material is electrical current.

The goal of the maximum-flow problem is to compute the greatest rate for shipping material from the source to the sink without violating any capacity constraints. It is one of the simplest problems concerning flow networks and, as we shall see in this chapter, this problem can be solved by efficient algorithms. Moreover, other network-flow problems are solvable by adapting the basic techniques used in maximum-flow algorithms.

This chapter presents two general methods for solving the maximum-flow problem. Section 24.1 formalizes the notions of flow networks and flows, formally defining the maximum-flow problem. Section 24.2 describes the classical method

of Ford and Fulkerson for finding maximum flows. We finish up with a simple application of this method, finding a maximum matching in an undirected bipartite graph, in Section 24.3. (Section 25.1 will give a more efficient algorithm that is specifically designed to find a maximum matching in a bipartite graph.)

24.1 Flow networks

This section gives a graph-theoretic definition of flow networks, discusses their properties, and defines the maximum-flow problem precisely. It also introduces some helpful notation.

Flow networks and flows

A ***flow network*** $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a nonnegative ***capacity*** $c(u, v) \geq 0$. We further require that if E contains an edge (u, v), then there is no edge (v, u) in the reverse direction. (We'll see shortly how to work around this restriction.) If $(u, v) \notin E$, then for convenience we define $c(u, v) = 0$, and we disallow self-loops. Each flow network contains two distinguished vertices: a ***source*** s and a ***sink*** t. For convenience, we assume that each vertex lies on some path from the source to the sink. That is, for each vertex $v \in V$, the flow network contains a path $s \rightsquigarrow v \rightsquigarrow t$. Because each vertex other than s has at least one entering edge, we have $|E| \geq |V| - 1$. Figure 24.1 shows an example of a flow network.

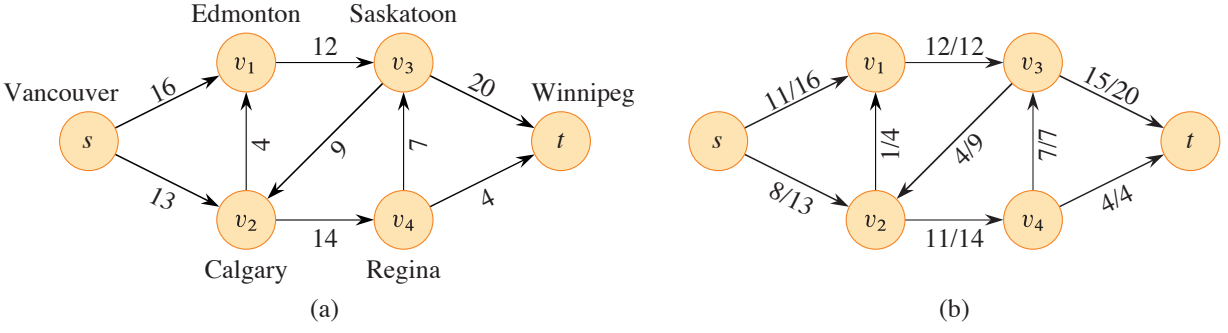


Figure 24.1 **(a)** A flow network $G = (V, E)$ for the Lucky Puck Company's trucking problem. The Vancouver factory is the source s, and the Winnipeg warehouse is the sink t. The company ships pucks through intermediate cities, but only $c(u, v)$ crates per day can go from city u to city v. Each edge is labeled with its capacity. **(b)** A flow f in G with value $|f| = 19$. Each edge (u, v) is labeled by $f(u, v)/c(u, v)$. The slash notation merely separates the flow and capacity and does not indicate division.

We are now ready to define flows more formally. Let $G = (V, E)$ be a flow network with a capacity function c. Let s be the source of the network, and let t be the sink. A ***flow*** in G is a real-valued function $f : V \times V \to \mathbb{R}$ that satisfies the following two properties:

Capacity constraint: For all $u, v \in V$, we require

$$0 \le f(u, v) \le c(u, v) .$$

The flow from one vertex to another must be nonnegative and must not exceed the given capacity.

Flow conservation: For all $u \in V - \{s, t\}$, we require

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v) .$$

The total flow into a vertex other than the source or sink must equal the total flow out of that vertex—informally, "flow in equals flow out."

When $(u, v) \notin E$, there can be no flow from u to v, and $f(u, v) = 0$.

We call the nonnegative quantity $f(u, v)$ the flow from vertex u to vertex v. The ***value*** $|f|$ of a flow f is defined as

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) , \tag{24.1}$$

that is, the total flow out of the source minus the flow into the source. (Here, the $|\cdot|$ notation denotes flow value, not absolute value or cardinality.) Typically, a flow network does not have any edges into the source, and the flow into the source, given by the summation $\sum_{v \in V} f(v, s)$, is 0. We include it, however, because when we introduce residual networks later in this chapter, the flow into the source can be positive. In the ***maximum-flow problem***, the input is a flow network G with source s and sink t, and the goal is to find a flow of maximum value.

An example of flow

A flow network can model the trucking problem shown in Figure 24.1(a). The Lucky Puck Company has a factory (source s) in Vancouver that manufactures hockey pucks, and it has a warehouse (sink t) in Winnipeg that stocks them. Lucky Puck leases space on trucks from another firm to ship the pucks from the factory to the warehouse. Because the trucks travel over specified routes (edges) between cities (vertices) and have a limited capacity, Lucky Puck can ship at most $c(u, v)$ crates per day between each pair of cities u and v in Figure 24.1(a). Lucky Puck

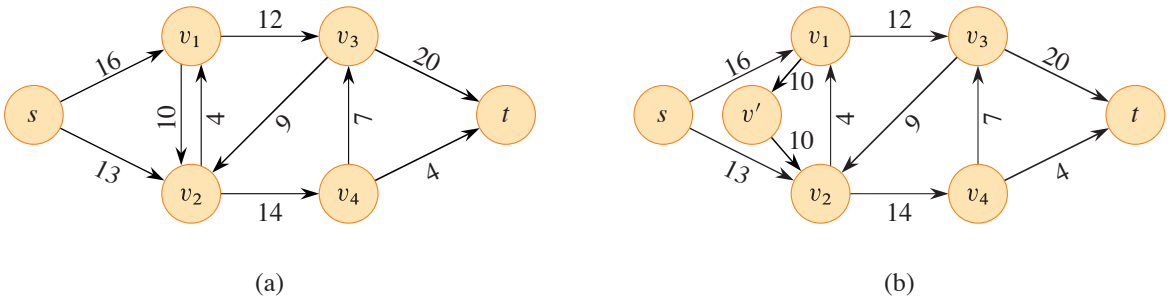


Figure 24.2 Converting a network with antiparallel edges to an equivalent one with no antiparallel edges. **(a)** A flow network containing both the edges (v_1, v_2) and (v_2, v_1). **(b)** An equivalent network with no antiparallel edges. A new vertex v' was added, and edge (v_1, v_2) was replaced by the pair of edges (v_1, v') and (v', v_2), both with the same capacity as (v_1, v_2).

has no control over these routes and capacities, and so the company cannot alter the flow network shown in Figure 24.1(a). They need to determine the largest number p of crates per day that they can ship and then to produce this amount, since there is no point in producing more pucks than they can ship to their warehouse. Lucky Puck is not concerned with how long it takes for a given puck to get from the factory to the warehouse. They care only that p crates per day leave the factory and p crates per day arrive at the warehouse.

A flow in this network models the "flow" of shipments because the number of crates shipped per day from one city to another is subject to a capacity constraint. Additionally, the model must obey flow conservation, for in a steady state, the rate at which pucks enter an intermediate city must equal the rate at which they leave. Otherwise, crates would accumulate at intermediate cities.

Modeling problems with antiparallel edges

Suppose that the trucking firm offers Lucky Puck the opportunity to lease space for 10 crates in trucks going from Edmonton to Calgary. It might seem natural to add this opportunity to our example and form the network shown in Figure 24.2(a). This network suffers from one problem, however: it violates the original assumption that if edge $(v_1, v_2) \in E$, then $(v_2, v_1) \notin E$. We call the two edges (v_1, v_2) and (v_2, v_1) ***antiparallel***. Thus, to model a flow problem with antiparallel edges, the network must be transformed into an equivalent one containing no antiparallel edges. Figure 24.2(b) displays this equivalent network. To transform the network, choose one of the two antiparallel edges, in this case (v_1, v_2), and split it by adding a new vertex v' and replacing edge (v_1, v_2) with the pair of edges (v_1, v') and (v', v_2). Also set the capacity of both new edges to the capacity of the original edge. The resulting network satisfies the property that if an edge belongs to

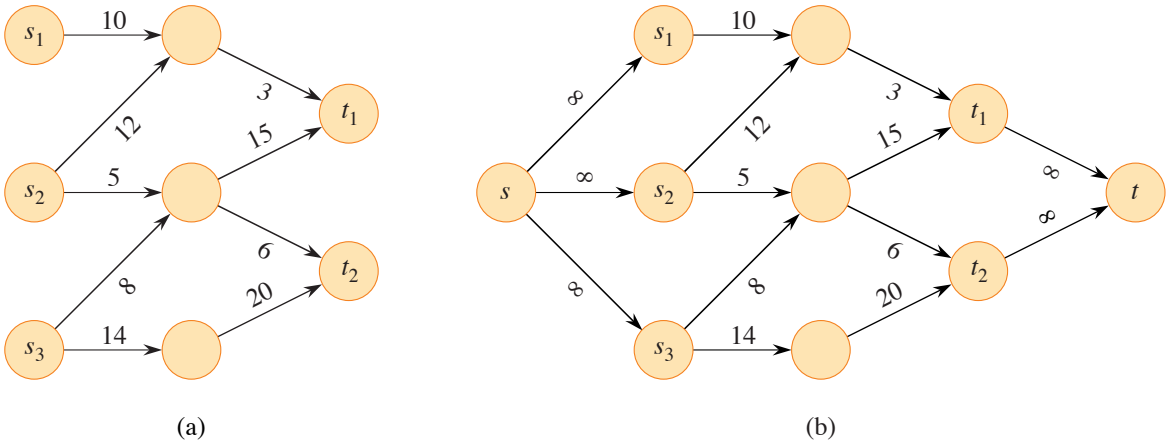


Figure 24.3 Converting a multiple-source, multiple-sink maximum-flow problem into a problem with a single source and a single sink. **(a)** A flow network with three sources $S = \{s_1, s_2, s_3\}$ and two sinks $T = \{t_1, t_2\}$. **(b)** An equivalent single-source, single-sink flow network. Add a supersource s and an edge with infinite capacity from s to each of the multiple sources. Also add a supersink t and an edge with infinite capacity from each of the multiple sinks to t.

the network, the reverse edge does not. As Exercise 24.1-1 asks you to prove, the resulting network is equivalent to the original one.

Networks with multiple sources and sinks

A maximum-flow problem may have several sources and sinks, rather than just one of each. The Lucky Puck Company, for example, might actually have a set of m factories $\{s_1, s_2, \ldots, s_m\}$ and a set of n warehouses $\{t_1, t_2, \ldots, t_n\}$, as shown in Figure 24.3(a). Fortunately, this problem is no harder than ordinary maximum flow.

The problem of determining a maximum flow in a network with multiple sources and multiple sinks reduces to an ordinary maximum-flow problem. Figure 24.3(b) shows how to convert the network from (a) to an ordinary flow network with only a single source and a single sink. Add a ***supersource*** s and add a directed edge (s, s_i) with capacity $c(s, s_i) = \infty$ for each $i = 1, 2, \ldots, m$. Similarly, create a new ***supersink*** t and add a directed edge (t_i, t) with capacity $c(t_i, t) = \infty$ for each $i = 1, 2, \ldots, n$. Intuitively, any flow in the network in (a) corresponds to a flow in the network in (b), and vice versa. The single supersource s provides as much flow as desired for the multiple sources s_i, and the single supersink t likewise consumes as much flow as desired for the multiple sinks t_i. Exercise 24.1-2 asks you to prove formally that the two problems are equivalent.

Exercises

24.1-1

Show that splitting an edge in a flow network yields an equivalent network. More formally, suppose that flow network G contains edge (u, v), and define a new flow network G' by creating a new vertex x and replacing (u, v) by new edges (u, x) and (x, v) with $c(u, x) = c(x, v) = c(u, v)$. Show that a maximum flow in G' has the same value as a maximum flow in G.

24.1-2

Extend the flow properties and definitions to the multiple-source, multiple-sink problem. Show that any flow in a multiple-source, multiple-sink flow network corresponds to a flow of identical value in the single-source, single-sink network obtained by adding a supersource and a supersink, and vice versa.

24.1-3

Suppose that a flow network $G = (V, E)$ violates the assumption that the network contains a path $s \rightsquigarrow v \rightsquigarrow t$ for all vertices $v \in V$. Let u be a vertex for which there is no path $s \rightsquigarrow u \rightsquigarrow t$. Show that there must exist a maximum flow f in G such that $f(u, v) = f(v, u) = 0$ for all vertices $v \in V$.

24.1-4

Let f be a flow in a network, and let α be a real number. The ***scalar flow product***, denoted αf, is a function from $V \times V$ to $\mathbb{R}$ defined by

$$(\alpha f)(u, v) = \alpha \cdot f(u, v) .$$

Prove that the flows in a network form a ***convex set***. That is, show that if f_1 and f_2 are flows, then so is $\alpha f_1 + (1 - \alpha) f_2$ for all α in the range $0 \le \alpha \le 1$.

24.1-5

State the maximum-flow problem as a linear-programming problem.

24.1-6

Professor Adam has two children who, unfortunately, dislike each other. The problem is so severe that not only do they refuse to walk to school together, but in fact each one refuses to walk on any block that the other child has stepped on that day. The children have no problem with their paths crossing at a corner. Fortunately both the professor's house and the school are on corners, but beyond that he is not sure if it is going to be possible to send both of his children to the same school. The professor has a map of his town. Show how to formulate the problem of determining whether both his children can go to the same school as a maximum-flow problem.

24.1-7

Suppose that, in addition to edge capacities, a flow network has ***vertex capacities***. That is each vertex v has a limit $l(v)$ on how much flow can pass through v. Show how to transform a flow network $G = (V, E)$ with vertex capacities into an equivalent flow network $G' = (V', E')$ without vertex capacities, such that a maximum flow in G' has the same value as a maximum flow in G. How many vertices and edges does G' have?

24.2 The Ford-Fulkerson method

This section presents the Ford-Fulkerson method for solving the maximum-flow problem. We call it a "method" rather than an "algorithm" because it encompasses several implementations with differing running times. The Ford-Fulkerson method depends on three important ideas that transcend the method and are relevant to many flow algorithms and problems: residual networks, augmenting paths, and cuts. These ideas are essential to the important max-flow min-cut theorem (Theorem 24.6), which characterizes the value of a maximum flow in terms of cuts of the flow network. We end this section by presenting one specific implementation of the Ford-Fulkerson method and analyzing its running time.

The Ford-Fulkerson method iteratively increases the value of the flow. It starts with $f(u, v) = 0$ for all $u, v \in V$, giving an initial flow of value 0. Each iteration increases the flow value in G by finding an "augmenting path" in an associated "residual network" G_f. The edges of the augmenting path in G_f indicate on which edges in G to update the flow in order to increase the flow value. Although each iteration of the Ford-Fulkerson method increases the value of the flow, we'll see that the flow on any particular edge of G may increase or decrease. Although it might seem counterintuitive to decrease the flow on an edge, doing so may enable flow to increase on other edges, allowing more flow to travel from the source to the sink. The Ford-Fulkerson method, given in the procedure FORD-FULKERSON-METHOD, repeatedly augments the flow until the residual network has no more augmenting paths. The max-flow min-cut theorem shows that upon termination, this process yields a maximum flow.

```
FORD-FULKERSON-METHOD(G, s, t)
1  initialize flow f to 0
2  while there exists an augmenting path p in the residual network G_f
3      augment flow f along p
4  return f
```

In order to implement and analyze the Ford-Fulkerson method, we need to introduce several additional concepts.

Residual networks

Intuitively, given a flow network G and a flow f, the residual network G_f consists of edges whose capacities represent how the flow can change on edges of G. An edge of the flow network can admit an amount of additional flow equal to the edge's capacity minus the flow on that edge. If that value is positive, that edge goes into G_f with a "residual capacity" of $c_f(u,v) = c(u,v) - f(u,v)$. The only edges of G that belong to G_f are those that can admit more flow. Those edges (u,v) whose flow equals their capacity have $c_f(u,v) = 0$, and they do not belong to G_f.

You might be surprised that the residual network G_f can also contain edges that are not in G. As an algorithm manipulates the flow, with the goal of increasing the total flow, it might need to decrease the flow on a particular edge in order to increase the flow elsewhere. In order to represent a possible decrease in the positive flow $f(u,v)$ on an edge in G, the residual network G_f contains an edge (v,u) with residual capacity $c_f(v,u) = f(u,v)$—that is, an edge that can admit flow in the opposite direction to (u,v), at most canceling out the flow on (u,v). These reverse edges in the residual network allow an algorithm to send back flow it has already sent along an edge. Sending flow back along an edge is equivalent to *decreasing* the flow on the edge, which is a necessary operation in many algorithms.

More formally, for a flow network $G = (V,E)$ with source s, sink t, and a flow f, consider a pair of vertices $u, v \in V$. We define the ***residual capacity*** $c_f(u,v)$ by

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E\ , \\ f(v,u) & \text{if } (v,u) \in E\ , \\ 0 & \text{otherwise}\ . \end{cases} \tag{24.2}$$

In a flow network, $(u,v) \in E$ implies $(v,u) \notin E$, and so exactly one case in equation (24.2) applies to each ordered pair of vertices.

As an example of equation (24.2), if $c(u,v) = 16$ and $f(u,v) = 11$, then $f(u,v)$ can increase by up to $c_f(u,v) = 5$ units before exceeding the capacity constraint on edge (u,v). Alternatively, up to 11 units of flow can return from v to u, so that $c_f(v,u) = 11$.

Given a flow network $G = (V,E)$ and a flow f, the ***residual network*** of G induced by f is $G_f = (V, E_f)$, where

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}\ . \tag{24.3}$$

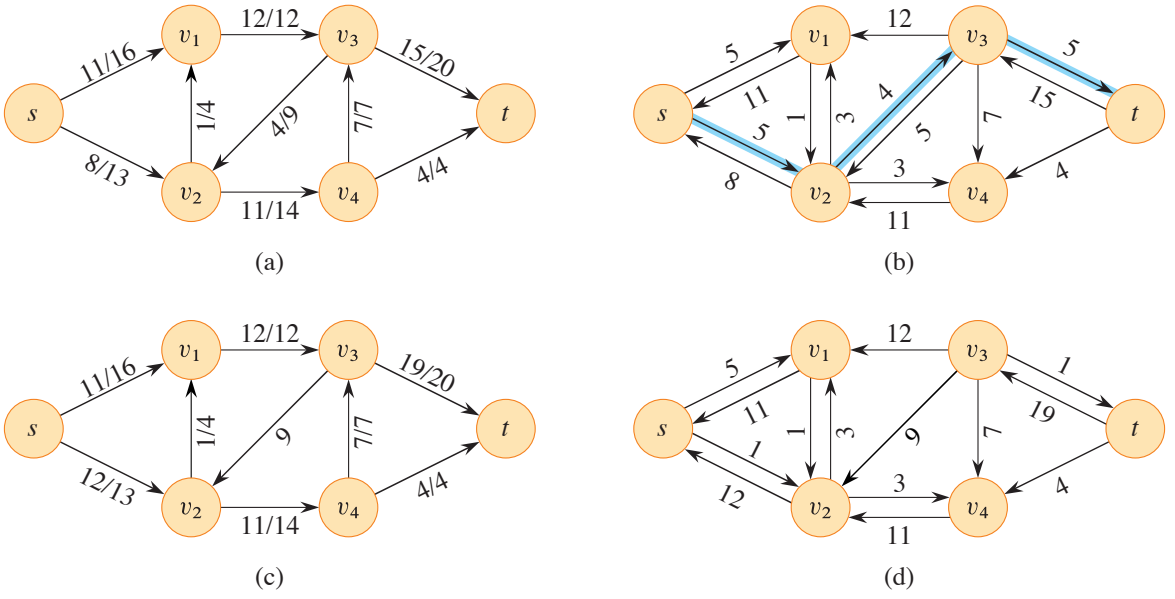


Figure 24.4 **(a)** The flow network G and flow f of Figure 24.1(b). **(b)** The residual network G_f with augmenting path p, having residual capacity $c_f(p) = c_f(v_2, v_3) = 4$, in blue. Edges with residual capacity equal to 0, such as (v_1, v_3), are not shown, a convention we follow in the remainder of this section. **(c)** The flow in G that results from augmenting along path p by its residual capacity 4. Edges carrying no flow, such as (v_3, v_2), are labeled only by their capacity, another convention we follow throughout. **(d)** The residual network induced by the flow in (c).

That is, as promised above, each edge of the residual network, or ***residual edge***, can admit a flow that is greater than 0. Figure 24.4(a) repeats the flow network G and flow f of Figure 24.1(b), and Figure 24.4(b) shows the corresponding residual network G_f. The edges in E_f are either edges in E or their reversals, and thus

$$|E_f| \le 2\,|E|\ .$$

Observe that the residual network G_f is similar to a flow network with capacities given by c_f. It does not satisfy the definition of a flow network, however, because it could contain antiparallel edges. Other than this difference, a residual network has the same properties as a flow network, and we can define a flow in the residual network as one that satisfies the definition of a flow, but with respect to capacities c_f in the residual network G_f.

A flow in a residual network provides a roadmap for adding flow to the original flow network. If f is a flow in G and f' is a flow in the corresponding residual network G_f, we define $f \uparrow f'$, the ***augmentation*** of flow f by f', to be a function from $V \times V$ to $\mathbb{R}$, defined by

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E\ , \\ 0 & \text{otherwise}\ . \end{cases} \tag{24.4}$$

The intuition behind this definition follows the definition of the residual network. The flow on (u,v) increases by $f'(u,v)$, but decreases by $f'(v,u)$ because pushing flow on the reverse edge in the residual network signifies decreasing the flow in the original network. Pushing flow on the reverse edge in the residual network is also known as ***cancellation***. For example, suppose that 5 crates of hockey pucks go from u to v and 2 crates go from v to u. That is equivalent (from the perspective of the final result) to sending 3 crates from u to v and none from v to u. Cancellation of this type is crucial for any maximum-flow algorithm.

The following lemma shows that augmenting a flow in G by a flow in G_f yields a new flow in G with a greater flow value.

Lemma 24.1

Let $G = (V,E)$ be a flow network with source s and sink t, and let f be a flow in G. Let G_f be the residual network of G induced by f, and let f' be a flow in G_f. Then the function $f \uparrow f'$ defined in equation (24.4) is a flow in G with value $|f \uparrow f'| = |f| + |f'|$.

Proof We first verify that $f \uparrow f'$ obeys the capacity constraint for each edge in E and flow conservation at each vertex in $V - \{s,t\}$.

For the capacity constraint, first observe that if $(u,v) \in E$, then $c_f(v,u) = f(u,v)$. Because f' is a flow in G_f, we have $f'(v,u) \le c_f(v,u)$, which gives $f'(v,u) \le f(u,v)$. Therefore,

$$\begin{aligned} (f \uparrow f')(u,v) &= f(u,v) + f'(u,v) - f'(v,u) && \text{(by equation (24.4))} \\ &\ge f(u,v) + f'(u,v) - f(u,v) && \text{(because } f'(v,u) \le f(u,v)\text{)} \\ &= f'(u,v) \\ &\ge 0\ . \end{aligned}$$

In addition,

$$\begin{aligned} &(f \uparrow f')(u,v) \\ &\qquad = f(u,v) + f'(u,v) - f'(v,u) && \text{(by equation (24.4))} \\ &\qquad \le f(u,v) + f'(u,v) && \text{(because flows are nonnegative)} \\ &\qquad \le f(u,v) + c_f(u,v) && \text{(capacity constraint)} \\ &\qquad = f(u,v) + c(u,v) - f(u,v) && \text{(definition of } c_f\text{)} \\ &\qquad = c(u,v)\ . \end{aligned}$$

To show that flow conservation holds and that $|f \uparrow f'| = |f| + |f'|$, we first prove the claim that for all $u \in V$, we have

$$\sum_{v \in V} (f \uparrow f')(u,v) - \sum_{v \in V} (f \uparrow f')(v,u)$$
$$= \sum_{v \in V} f(u,v) - \sum_{v \in V} f(v,u) + \sum_{v \in V} f'(u,v) - \sum_{v \in V} f'(v,u) \,. \tag{24.5}$$

Because we disallow antiparallel edges in G (but not in G_f), we know that for each vertex u, there can be an edge (u,v) or (v,u) in G, but never both. For a fixed vertex u, define $V_l(u) = \{v : (u,v) \in E\}$ to be the set of vertices with edges in G leaving u, and define $V_e(u) = \{v : (v,u) \in E\}$ to be the set of vertices with edges in G entering u. We have $V_l(u) \cup V_e(u) \subseteq V$ and, because G contains no antiparallel edges, $V_l(u) \cap V_e(u) = \emptyset$. By the definition of flow augmentation in equation (24.4), only vertices v in $V_l(u)$ can have positive $(f \uparrow f')(u,v)$, and only vertices v in $V_e(u)$ can have positive $(f \uparrow f')(v,u)$. Starting from the left-hand side of equation (24.5), we use this fact and then reorder and group terms, giving

$$\begin{aligned}
&\sum_{v \in V} (f \uparrow f')(u,v) - \sum_{v \in V} (f \uparrow f')(v,u) \\
&\quad = \sum_{v \in V_l(u)} (f \uparrow f')(u,v) - \sum_{v \in V_e(u)} (f \uparrow f')(v,u) \\
&\quad = \sum_{v \in V_l(u)} (f(u,v) + f'(u,v) - f'(v,u)) - \sum_{v \in V_e(u)} (f(v,u) + f'(v,u) - f'(u,v)) \\
&\quad = \sum_{v \in V_l(u)} f(u,v) + \sum_{v \in V_l(u)} f'(u,v) - \sum_{v \in V_l(u)} f'(v,u) \\
&\qquad\qquad - \sum_{v \in V_e(u)} f(v,u) - \sum_{v \in V_e(u)} f'(v,u) + \sum_{v \in V_e(u)} f'(u,v) \\
&\quad = \sum_{v \in V_l(u)} f(u,v) - \sum_{v \in V_e(u)} f(v,u) \\
&\qquad\qquad + \sum_{v \in V_l(u)} f'(u,v) + \sum_{v \in V_e(u)} f'(u,v) - \sum_{v \in V_l(u)} f'(v,u) - \sum_{v \in V_e(u)} f'(v,u) \\
&\quad = \sum_{v \in V_l(u)} f(u,v) - \sum_{v \in V_e(u)} f(v,u) + \sum_{v \in V_l(u) \cup V_e(u)} f'(u,v) - \sum_{v \in V_l(u) \cup V_e(u)} f'(v,u) \,.
\end{aligned} \tag{24.6}$$

In equation (24.6), all four summations can extend to sum over V, since each additional term has value 0. (Exercise 24.2-1 asks you to prove this formally.) Taking all four summations over V, instead of just subsets of V, proves the claim in equation (24.5).

Now we are ready to prove flow conservation for $f \uparrow f'$ and that $|f \uparrow f'| = |f| + |f'|$. For the latter property, let $u = s$ in equation (24.5). Then, we have

$$\begin{aligned}|f \uparrow f'| &= \sum_{v \in V}(f \uparrow f')(s,v) - \sum_{v \in V}(f \uparrow f')(v,s) \\ &= \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) + \sum_{v \in V} f'(s,v) - \sum_{v \in V} f'(v,s) \\ &= |f| + |f'| \ .\end{aligned}$$

For flow conservation, observe that for any vertex u that is neither s nor t, flow conservation for f and f' means that the right-hand side of equation (24.5) is 0, and thus $\sum_{v \in V}(f \uparrow f')(u,v) = \sum_{v \in V}(f \uparrow f')(v,u)$. ■

Augmenting paths

Given a flow network $G = (V, E)$ and a flow f, an ***augmenting path*** p is a simple path from s to t in the residual network G_f. By the definition of the residual network, the flow on an edge (u, v) of an augmenting path may increase by up to $c_f(u, v)$ without violating the capacity constraint on whichever of (u, v) and (v, u) belongs to the original flow network G.

The blue path in Figure 24.4(b) is an augmenting path. Treating the residual network G_f in the figure as a flow network, the flow through each edge of this path can increase by up to 4 units without violating a capacity constraint, since the smallest residual capacity on this path is $c_f(v_2, v_3) = 4$. We call the maximum amount by which we can increase the flow on each edge in an augmenting path p the ***residual capacity*** of p, given by

$$c_f(p) = \min\{c_f(u,v) : (u,v) \text{ is in } p\} \ .$$

The following lemma, which Exercise 24.2-7 asks you to prove, makes the above argument more precise.

Lemma 24.2

Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Define a function $f_p : V \times V \to \mathbb{R}$ by

$$f_p(u,v) = \begin{cases} c_f(p) & \text{if } (u,v) \text{ is on } p \ , \\ 0 & \text{otherwise} \ . \end{cases} \tag{24.7}$$

Then, f_p is a flow in G_f with value $|f_p| = c_f(p) > 0$. ■

The following corollary shows that augmenting f by f_p produces another flow in G whose value is closer to the maximum. Figure 24.4(c) shows the result of augmenting the flow f from Figure 24.4(a) by the flow f_p in Figure 24.4(b), and Figure 24.4(d) shows the ensuing residual network.

Corollary 24.3

Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Let f_p be defined as in equation (24.7), and suppose that f is augmented by f_p. Then the function $f \uparrow f_p$ is a flow in G with value $|f \uparrow f_p| = |f| + |f_p| > |f|$.

Proof Immediate from Lemmas 24.1 and 24.2. ■

Cuts of flow networks

The Ford-Fulkerson method repeatedly augments the flow along augmenting paths until it has found a maximum flow. How do we know that when the algorithm terminates, it has actually found a maximum flow? The max-flow min-cut theorem, which we will prove shortly, tells us that a flow is maximum if and only if its residual network contains no augmenting path. To prove this theorem, though, we must first explore the notion of a cut of a flow network.

A ***cut*** (S, T) of flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$. (This definition is similar to the definition of "cut" that we used for minimum spanning trees in Chapter 21, except that here we are cutting a directed graph rather than an undirected graph, and we insist that $s \in S$ and $t \in T$.) If f is a flow, then the ***net flow*** $f(S, T)$ across the cut (S, T) is defined to be

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u) . \tag{24.8}$$

The ***capacity*** of the cut (S, T) is

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v) . \tag{24.9}$$

A ***minimum cut*** of a network is a cut whose capacity is minimum over all cuts of the network.

You probably noticed that the definitions of flow across a cut and capacity of a cut differ in that flow counts edges going in both directions across the cut, but capacity counts only edges going from the source side of the cut toward the sink side. This asymmetry is intentional and important. The reason for this difference will become apparent later in this section.

Figure 24.5 shows the cut $(\{s, v_1, v_2\}, \{v_3, v_4, t\})$ in the flow network of Figure 24.1(b). The net flow across this cut is

$$\begin{aligned} f(v_1, v_3) + f(v_2, v_4) - f(v_3, v_2) &= 12 + 11 - 4 \\ &= 19 , \end{aligned}$$

and the capacity of this cut is

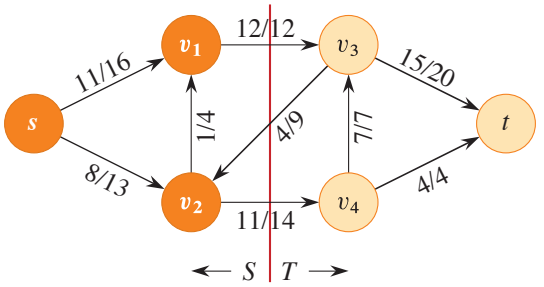


Figure 24.5 A cut (S, T) in the flow network of Figure 24.1(b), where $S = \{s, v_1, v_2\}$ and $T = \{v_3, v_4, t\}$. The vertices in S are orange, and the vertices in T are tan. The net flow across (S, T) is $f(S, T) = 19$, and the capacity is $c(S, T) = 26$.

$$\begin{aligned} c(v_1, v_3) + c(v_2, v_4) &= 12 + 14 \\ &= 26 . \end{aligned}$$

The following lemma shows that, for a given flow f, the net flow across any cut is the same, and it equals $|f|$, the value of the flow.

Lemma 24.4
Let f be a flow in a flow network G with source s and sink t, and let (S, T) be any cut of G. Then the net flow across (S, T) is $f(S, T) = |f|$.

Proof For any vertex $u \in V - \{s, t\}$, rewrite the flow-conservation condition as

$$\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) = 0 . \tag{24.10}$$

Taking the definition of $|f|$ from equation (24.1) and adding the left-hand side of equation (24.10), which equals 0, summed over all vertices in $S - \{s\}$, gives

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{u \in S - \{s\}} \left(\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) \right) .$$

Expanding the right-hand summation and regrouping terms yields

$$\begin{aligned} |f| &= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{u \in S - \{s\}} \sum_{v \in V} f(u, v) - \sum_{u \in S - \{s\}} \sum_{v \in V} f(v, u) \\ &= \sum_{v \in V} \left(f(s, v) + \sum_{u \in S - \{s\}} f(u, v) \right) - \sum_{v \in V} \left(f(v, s) + \sum_{u \in S - \{s\}} f(v, u) \right) \\ &= \sum_{v \in V} \sum_{u \in S} f(u, v) - \sum_{v \in V} \sum_{u \in S} f(v, u) . \end{aligned}$$

Because $V = S \cup T$ and $S \cap T = \emptyset$, splitting each summation over V into summations over S and T gives

$$\begin{aligned}
|f| &= \sum_{v \in S} \sum_{u \in S} f(u,v) + \sum_{v \in T} \sum_{u \in S} f(u,v) - \sum_{v \in S} \sum_{u \in S} f(v,u) - \sum_{v \in T} \sum_{u \in S} f(v,u) \\
&= \sum_{v \in T} \sum_{u \in S} f(u,v) - \sum_{v \in T} \sum_{u \in S} f(v,u) \\
&\qquad + \left(\sum_{v \in S} \sum_{u \in S} f(u,v) - \sum_{v \in S} \sum_{u \in S} f(v,u) \right) .
\end{aligned}$$

The two summations within the parentheses are actually the same, since for all vertices $x, y \in S$, the term $f(x,y)$ appears once in each summation. Hence, these summations cancel, yielding

$$\begin{aligned}
|f| &= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) \\
&= f(S,T) . \qquad \blacksquare
\end{aligned}$$

A corollary to Lemma 24.4 shows how cut capacities bound the value of a flow.

Corollary 24.5
The value of any flow f in a flow network G is bounded from above by the capacity of any cut of G.

Proof Let (S,T) be any cut of G and let f be any flow. By Lemma 24.4 and the capacity constraint,

$$\begin{aligned}
|f| &= f(S,T) \\
&= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) \\
&\le \sum_{u \in S} \sum_{v \in T} f(u,v) \\
&\le \sum_{u \in S} \sum_{v \in T} c(u,v) \\
&= c(S,T) . \qquad \blacksquare
\end{aligned}$$

Corollary 24.5 yields the immediate consequence that the value of a maximum flow in a network is bounded from above by the capacity of a minimum cut of the network. The important max-flow min-cut theorem, which we now state and prove, says that the value of a maximum flow is in fact equal to the capacity of a minimum cut.

Theorem 24.6 (Max-flow min-cut theorem)
If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

1. f is a maximum flow in G.
2. The residual network G_f contains no augmenting paths.
3. $|f| = c(S, T)$ for some cut (S, T) of G.

Proof (1) $\Rightarrow$ (2): Suppose for the sake of contradiction that f is a maximum flow in G but that G_f has an augmenting path p. Then, by Corollary 24.3, the flow found by augmenting f by f_p, where f_p is given by equation (24.7), is a flow in G with value strictly greater than $|f|$, contradicting the assumption that f is a maximum flow.

(2) $\Rightarrow$ (3): Suppose that G_f has no augmenting path, that is, that G_f contains no path from s to t. Define

$$S = \{v \in V : \text{there exists a path from } s \text{ to } v \text{ in } G_f\}$$

and $T = V - S$. The partition (S, T) is a cut: we have $s \in S$ trivially and $t \notin S$ because there is no path from s to t in G_f. Now consider a pair of vertices $u \in S$ and $v \in T$. If $(u, v) \in E$, we must have $f(u, v) = c(u, v)$, since otherwise $(u, v) \in E_f$, which would place v in set S. If $(v, u) \in E$, we must have $f(v, u) = 0$, because otherwise $c_f(u, v) = f(v, u)$ would be positive and we would have $(u, v) \in E_f$, which again would place v in S. Of course, if neither (u, v) nor (v, u) belongs to E, then $f(u, v) = f(v, u) = 0$. We thus have

$$\begin{aligned} f(S, T) &= \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u) \\ &= \sum_{u \in S} \sum_{v \in T} c(u, v) - \sum_{v \in T} \sum_{u \in S} 0 \\ &= c(S, T) \, . \end{aligned}$$

By Lemma 24.4, therefore, $|f| = f(S, T) = c(S, T)$.

(3) $\Rightarrow$ (1): By Corollary 24.5, $|f| \leq c(S, T)$ for all cuts (S, T). The condition $|f| = c(S, T)$ thus implies that f is a maximum flow. ■

The basic Ford-Fulkerson algorithm

Each iteration of the Ford-Fulkerson method finds *some* augmenting path p and uses p to modify the flow f. As Lemma 24.2 and Corollary 24.3 suggest, replacing f by $f \uparrow f_p$ produces a new flow whose value is $|f| + |f_p|$. The procedure FORD-FULKERSON on the next page implements the method by updating the flow

attribute $(u, v).f$ for each edge $(u, v) \in E$.[1] It assumes implicitly that $(u, v).f = 0$ if $(u, v) \notin E$. The procedure also assumes that the capacities $c(u, v)$ come with the flow network, and that $c(u, v) = 0$ if $(u, v) \notin E$. The procedure computes the residual capacity $c_f(u, v)$ in accordance with the formula (24.2). The expression $c_f(p)$ in the code is just a temporary variable that stores the residual capacity of the path p.

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ G.E
2      (u, v).f = 0
3  while there exists a path p from s to t in the residual network G_f
4      c_f(p) = min {c_f(u, v) : (u, v) is in p}
5      for each edge (u, v) in p
6          if (u, v) ∈ G.E
7              (u, v).f = (u, v).f + c_f(p)
8          else (v, u).f = (v, u).f − c_f(p)
9  return f
```

The FORD-FULKERSON procedure simply expands on the FORD-FULKERSON-METHOD pseudocode given earlier. Figure 24.6 shows the result of each iteration in a sample run. Lines 1–2 initialize the flow f to 0. The **while** loop of lines 3–8 repeatedly finds an augmenting path p in G_f and augments flow f along p by the residual capacity $c_f(p)$. Each residual edge in path p is either an edge in the original network or the reversal of an edge in the original network. Lines 6–8 update the flow in each case appropriately, adding flow when the residual edge is an original edge and subtracting it otherwise. When no augmenting paths exist, the flow f is a maximum flow.

Analysis of Ford-Fulkerson

The running time of FORD-FULKERSON depends on the augmenting path p and how it's found in line 3. If the edge capacities are irrational numbers, it's possible to choose the augmenting path so that the algorithm never terminates: the value of the flow increases with successive augmentations, but never converges to the maximum flow value. The good news is that if the algorithm finds the augmenting path by using a breadth-first search (which we saw in Section 20.2), it runs in

[1] Recall from Section 20.1 that we represent an attribute f for edge (u, v) with the same style of notation—$(u, v).f$—that we use for an attribute of any other object.

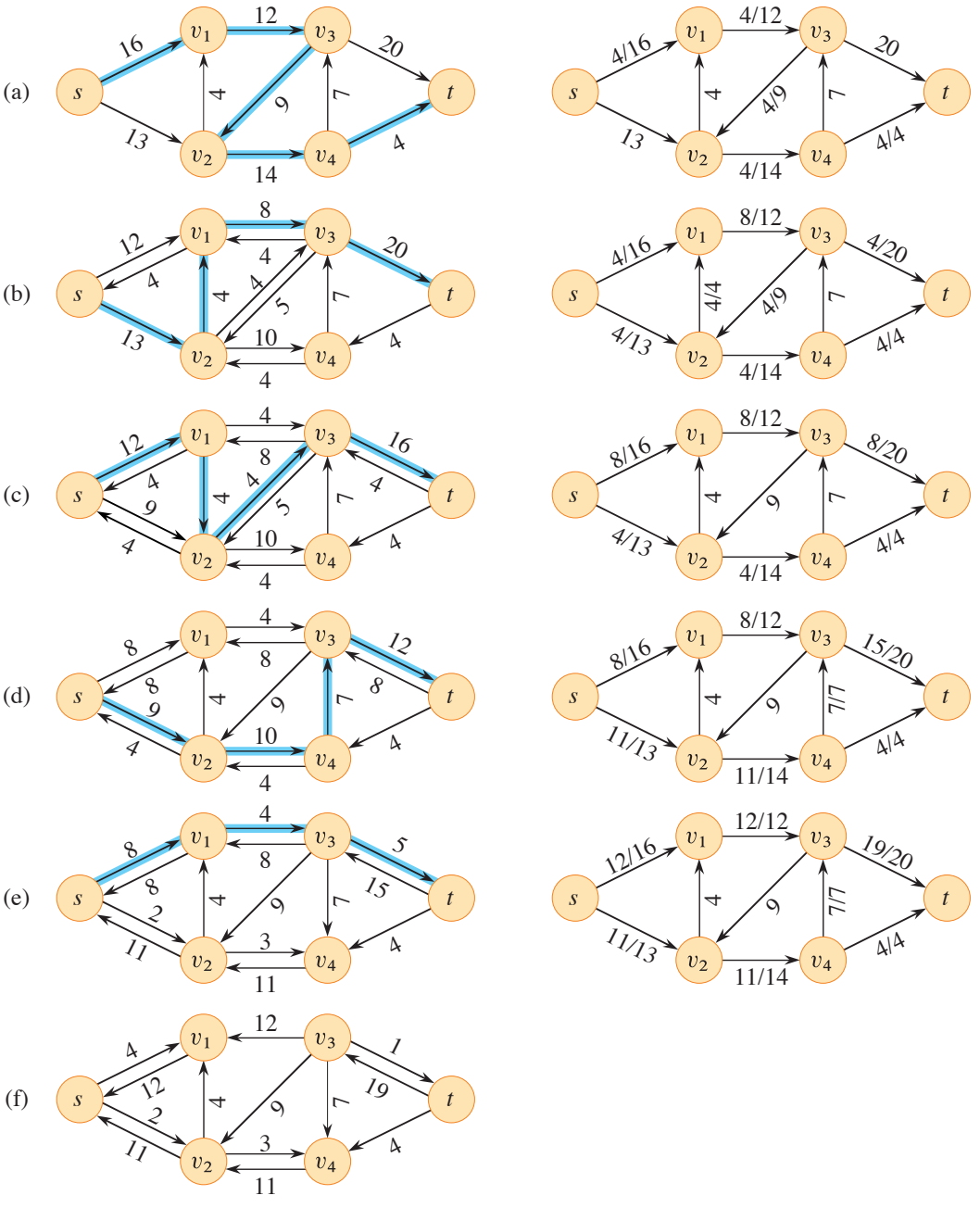


Figure 24.6 The execution of the basic Ford-Fulkerson algorithm. **(a)–(e)** Successive iterations of the **while** loop. The left side of each part shows the residual network G_f from line 3 with a blue augmenting path p. The right side of each part shows the new flow f that results from augmenting f by f_p. The residual network in (a) is the input flow network G. **(f)** The residual network at the last **while** loop test. It has no augmenting paths, and the flow f shown in (e) is therefore a maximum flow. The value of the maximum flow found is 23.

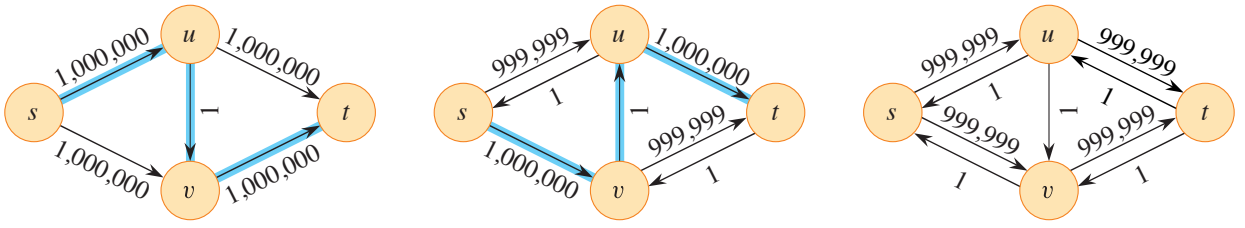


Figure 24.7 **(a)** A flow network for which FORD-FULKERSON can take $\Theta(E\ |f^*|)$ time, where f^* is a maximum flow, shown here with $|f^*| = 2{,}000{,}000$. The blue path is an augmenting path with residual capacity 1. **(b)** The resulting residual network, with another augmenting path whose residual capacity is 1. **(c)** The resulting residual network.

polynomial time. Before proving this result, we obtain a simple bound for the case in which all capacities are integers and the algorithm finds any augmenting path.

In practice, the maximum-flow problem often arises with integer capacities. If the capacities are rational numbers, an appropriate scaling transformation can make them all integers. If f^* denotes a maximum flow in the transformed network, then a straightforward implementation of FORD-FULKERSON executes the **while** loop of lines 3–8 at most $|f^*|$ times, since the flow value increases by at least 1 unit in each iteration.

A good implementation should perform the work done within the **while** loop efficiently. It should represent the flow network $G = (V, E)$ with the right data structure and find an augmenting path by a linear-time algorithm. Let's assume that the implementation keeps a data structure corresponding to a directed graph $G' = (V, E')$, where $E' = \{(u, v) : (u, v) \in E \text{ or } (v, u) \in E\}$. Edges in the network G are also edges in G', making it straightforward to maintain capacities and flows in this data structure. Given a flow f on G, the edges in the residual network G_f consist of all edges (u, v) of G' such that $c_f(u, v) > 0$, where c_f conforms to equation (24.2). The time to find a path in a residual network is therefore $O(V + E') = O(E)$ using either depth-first search or breadth-first search. Each iteration of the **while** loop thus takes $O(E)$ time, as does the initialization in lines 1–2, making the total running time of the FORD-FULKERSON algorithm $O(E\ |f^*|)$.

When the capacities are integers and the optimal flow value $|f^*|$ is small, the running time of the Ford-Fulkerson algorithm is good. Figure 24.7(a) shows an example of what can happen on a simple flow network for which $|f^*|$ is large. A maximum flow in this network has value 2,000,000: 1,000,000 units of flow traverse the path $s \to u \to t$, and another 1,000,000 units traverse the path $s \to v \to t$. If the first augmenting path found by FORD-FULKERSON is $s \to u \to v \to t$, shown

in Figure 24.7(a), the flow has value 1 after the first iteration. The resulting residual network appears in Figure 24.7(b). If the second iteration finds the augmenting path $s \to v \to u \to t$, as shown in Figure 24.7(b), the flow then has value 2. Figure 24.7(c) shows the resulting residual network. If the algorithm continues alternately choosing the augmenting paths $s \to u \to v \to t$ and $s \to v \to u \to t$, it performs a total of 2,000,000 augmentations, increasing the flow value by only 1 unit in each.

The Edmonds-Karp algorithm

In the example of Figure 24.7, the algorithm never chooses the augmenting path with the fewest edges. It should have. By using breadth-first search to find an augmenting path in the residual network, the algorithm runs in polynomial time, independent of the maximum flow value. We call the Ford-Fulkerson method so implemented the ***Edmonds-Karp algorithm***.

Let's now prove that the Edmonds-Karp algorithm runs in $O(VE^2)$ time. The analysis depends on the distances to vertices in the residual network G_f. The notation $\delta_f(u, v)$ denotes the shortest-path distance from u to v in G_f, where each edge has unit distance.

Lemma 24.7
If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and sink t, then for all vertices $v \in V - \{s, t\}$, the shortest-path distance $\delta_f(s, v)$ in the residual network G_f increases monotonically with each flow augmentation.

Proof We'll suppose that a flow augmentation occurs that causes the shortest-path distance from s to some vertex $v \in V - \{s, t\}$ to decrease and then derive a contradiction. Let f be the flow just before an augmentation that decreases some shortest-path distance, and let f' be the flow just afterward. Let v be a vertex with the minimum $\delta_{f'}(s, v)$ whose distance was decreased by the augmentation, so that $\delta_{f'}(s, v) < \delta_f(s, v)$. Let $p = s \rightsquigarrow u \to v$ be a shortest path from s to v in $G_{f'}$, so that $(u, v) \in E_{f'}$ and

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1 \,. \tag{24.11}$$

Because of how we chose v, we know that the distance of vertex u from the source s did not decrease, that is,

$$\delta_{f'}(s, u) \geq \delta_f(s, u) \,. \tag{24.12}$$

We claim that $(u, v) \notin E_f$. Why? If we have $(u, v) \in E_f$, then we also have

$$\begin{aligned}
\delta_f(s,v) &\le \delta_f(s,u) + 1 && \text{(by Lemma 22.10, the triangle inequality)} \\
&\le \delta_{f'}(s,u) + 1 && \text{(by inequality (24.12))} \\
&= \delta_{f'}(s,v) && \text{(by equation (24.11))} ,
\end{aligned}$$

which contradicts our assumption that $\delta_{f'}(s,v) < \delta_f(s,v)$.

How can we have $(u,v) \notin E_f$ and $(u,v) \in E_{f'}$? The augmentation must have increased the flow from v to u, so that edge (v,u) was in the augmenting path. The augmenting path was a shortest path from s to t in G_f, and since any subpath of a shortest path is itself a shortest path, this augmenting path includes a shortest path from s to u in G_f that has (v,u) as its last edge. Therefore,

$$\begin{aligned}
\delta_f(s,v) &= \delta_f(s,u) - 1 \\
&\le \delta_{f'}(s,u) - 1 && \text{(by inequality (24.12))} \\
&= \delta_{f'}(s,v) - 2 && \text{(by equation (24.11))} ,
\end{aligned}$$

so that $\delta_{f'}(s,v) > \delta_f(s,v)$, contradicting our assumption that $\delta_{f'}(s,v) < \delta_f(s,v)$. We conclude that our assumption that such a vertex v exists is incorrect. ■

The next theorem bounds the number of iterations of the Edmonds-Karp algorithm.

Theorem 24.8

If the Edmonds-Karp algorithm is run on a flow network $G = (V,E)$ with source s and sink t, then the total number of flow augmentations performed by the algorithm is $O(VE)$.

Proof We say that an edge (u,v) in a residual network G_f is ***critical*** on an augmenting path p if the residual capacity of p is the residual capacity of (u,v), that is, if $c_f(p) = c_f(u,v)$. After flow is augmented along an augmenting path, any critical edge on the path disappears from the residual network. Moreover, at least one edge on any augmenting path must be critical. We'll show that each of the $|E|$ edges can become critical at most $|V|/2$ times.

Let u and v be vertices in V that are connected by an edge in E. Since augmenting paths are shortest paths, when (u,v) is critical for the first time, we have

$$\delta_f(s,v) = \delta_f(s,u) + 1 .$$

Once the flow is augmented, the edge (u,v) disappears from the residual network. It cannot reappear later on another augmenting path until after the flow from u to v is decreased, which occurs only if (v,u) appears on an augmenting path. If f' is the flow in G when this event occurs, then we have

$$\delta_{f'}(s,u) = \delta_{f'}(s,v) + 1 .$$

Since $\delta_f(s,v) \le \delta_{f'}(s,v)$ by Lemma 24.7, we have

$$\begin{aligned}\delta_{f'}(s,u) &= \delta_{f'}(s,v) + 1 \\ &\ge \delta_f(s,v) + 1 \\ &= \delta_f(s,u) + 2 \,.\end{aligned}$$

Consequently, from the time (u,v) becomes critical to the time when it next becomes critical, the distance of u from the source increases by at least 2. The distance of u from the source is initially at least 0. Because edge (u,v) is on an augmenting path, and augmenting paths end at t, we know that u cannot be t, so that in any residual network that has a path from s to u, the shortest such path has at most $|V| - 2$ edges. Thus, after the first time that (u,v) becomes critical, it can become critical at most $(|V| - 2)/2 = |V|/2 - 1$ times more, for a total of at most $|V|/2$ times. Since there are $O(E)$ pairs of vertices that can have an edge between them in a residual network, the total number of critical edges during the entire execution of the Edmonds-Karp algorithm is $O(VE)$. Each augmenting path has at least one critical edge, and hence the theorem follows. ■

Because each iteration of FORD-FULKERSON takes $O(E)$ time when it uses breadth-first search to find the augmenting path, the total running time of the Edmonds-Karp algorithm is $O(VE^2)$.

Exercises

24.2-1

Prove that the summations in equation (24.6) equal the summations on the right-hand side of equation (24.5).

24.2-2

In Figure 24.1(b), what is the net flow across the cut $(\{s, v_2, v_4\}, \{v_1, v_3, t\})$? What is the capacity of this cut?

24.2-3

Show the execution of the Edmonds-Karp algorithm on the flow network of Figure 24.1(a).

24.2-4

In the example of Figure 24.6, what is the minimum cut corresponding to the maximum flow shown? Of the augmenting paths appearing in the example, which one cancels flow?

24.2-5

The construction in Section 24.1 to convert a flow network with multiple sources and sinks into a single-source, single-sink network adds edges with infinite capacity. Prove that any flow in the resulting network has a finite value if the edges of the original network with multiple sources and sinks have finite capacity.

24.2-6

Suppose that each source s_i in a flow network with multiple sources and sinks produces exactly p_i units of flow, so that $\sum_{v \in V} f(s_i, v) = p_i$. Suppose also that each sink t_j consumes exactly q_j units, so that $\sum_{v \in V} f(v, t_j) = q_j$, where $\sum_i p_i = \sum_j q_j$. Show how to convert the problem of finding a flow f that obeys these additional constraints into the problem of finding a maximum flow in a single-source, single-sink flow network.

24.2-7

Prove Lemma 24.2.

24.2-8

Suppose that we redefine the residual network to disallow edges into s. Argue that the procedure FORD-FULKERSON still correctly computes a maximum flow.

24.2-9

Suppose that both f and f' are flows in a flow network. Does the augmented flow $f \uparrow f'$ satisfy the flow conservation property? Does it satisfy the capacity constraint?

24.2-10

Show how to find a maximum flow in a flow network $G = (V, E)$ by a sequence of at most $|E|$ augmenting paths. (*Hint:* Determine the paths *after* finding the maximum flow.)

24.2-11

The ***edge connectivity*** of an undirected graph is the minimum number k of edges that must be removed to disconnect the graph. For example, the edge connectivity of a tree is 1, and the edge connectivity of a cyclic chain of vertices is 2. Show how to determine the edge connectivity of an undirected graph $G = (V, E)$ by running a maximum-flow algorithm on at most $|V|$ flow networks, each having $O(V + E)$ vertices and $O(E)$ edges.

24.2-12

You are given a flow network G, where G contains edges entering the source s. Let f be a flow in G with $|f| \geq 0$ in which one of the edges (v, s) entering

the source has $f(v, s) = 1$. Prove that there must exist another flow f' with $f'(v, s) = 0$ such that $|f| = |f'|$. Give an $O(E)$-time algorithm to compute f', given f and assuming that all edge capacities are integers.

24.2-13

Suppose that you wish to find, among all minimum cuts in a flow network G with integer capacities, one that contains the smallest number of edges. Show how to modify the capacities of G to create a new flow network G' in which any minimum cut in G' is a minimum cut with the smallest number of edges in G.

24.3 Maximum bipartite matching

Some combinatorial problems can be cast as maximum-flow problems, such as the multiple-source, multiple-sink maximum-flow problem from Section 24.1. Other combinatorial problems seem on the surface to have little to do with flow networks, but they can in fact be reduced to maximum-flow problems. This section presents one such problem: finding a maximum matching in a bipartite graph. In order to solve this problem, we'll take advantage of an integrality property provided by the Ford-Fulkerson method. We'll also see how to use the Ford-Fulkerson method to solve the maximum-bipartite-matching problem on a graph $G = (V, E)$ in $O(VE)$ time. Section 25.1 will present an algorithm specifically designed to solve this problem.

The maximum-bipartite-matching problem

Given an undirected graph $G = (V, E)$, a ***matching*** is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v. We say that a vertex $v \in V$ is ***matched*** by the matching M if some edge in M is incident on v, and otherwise, v is ***unmatched***. A ***maximum matching*** is a matching of maximum cardinality, that is, a matching M such that for any matching M', we have $|M| \geq |M'|$. In this section, we restrict our attention to finding maximum matchings in bipartite graphs: graphs in which the vertex set can be partitioned into $V = L \cup R$, where L and R are disjoint and all edges in E go between L and R. We further assume that every vertex in V has at least one incident edge. Figure 24.8 illustrates the notion of a matching in a bipartite graph.

The problem of finding a maximum matching in a bipartite graph has many practical applications. As an example, consider matching a set L of machines with a set R of tasks to be performed simultaneously. An edge (u, v) in E signifies that

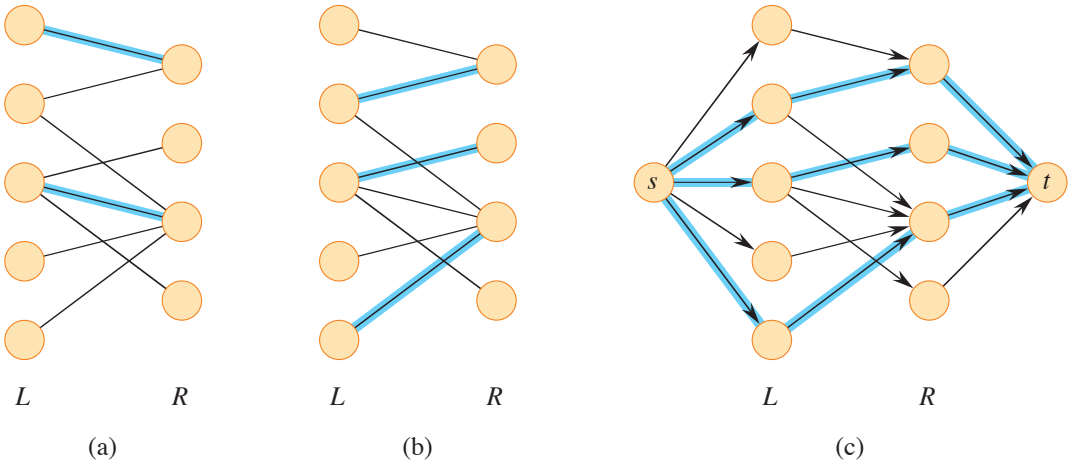


Figure 24.8 A bipartite graph $G = (V, E)$ with vertex partition $V = L \cup R$. **(a)** A matching with cardinality 2, indicated by blue edges. **(b)** A maximum matching with cardinality 3. **(c)** The corresponding flow network G' with a maximum flow shown. Each edge has unit capacity. Blue edges have a flow of 1, and all other edges carry no flow. The blue edges from L to R correspond to those in the maximum matching from (b).

a particular machine $u \in L$ is capable of performing a particular task $v \in R$. A maximum matching provides work for as many machines as possible.

Finding a maximum bipartite matching

The Ford-Fulkerson method provides a basis for finding a maximum matching in an undirected bipartite graph $G = (V, E)$ in time polynomial in $|V|$ and $|E|$. The trick is to construct a flow network in which flows correspond to matchings, as shown in Figure 24.8(c). We define the ***corresponding flow network*** $G' = (V', E')$ for the bipartite graph G as follows. Let the source s and sink t be new vertices not in V, and let $V' = V \cup \{s, t\}$. If the vertex partition of G is $V = L \cup R$, the directed edges of G' are the edges of E, directed from L to R, along with $|V|$ new directed edges:

$$\begin{aligned} E' = {} & \{(s, u) : u \in L\} \\ & \cup \{(u, v) : u \in L, v \in R, \text{ and } (u, v) \in E\} \\ & \cup \{(v, t) : v \in R\} \ . \end{aligned}$$

To complete the construction, assign unit capacity to each edge in E'. Since each vertex in V has at least one incident edge, $|E| \geq |V|/2$. Thus, $|E| \leq |E'| = |E| + |V| \leq 3|E|$, and so $|E'| = \Theta(E)$.

The following lemma shows that a matching in G corresponds directly to a flow in G's corresponding flow network G'. We say that a flow f on a flow network $G = (V, E)$ is ***integer-valued*** if $f(u, v)$ is an integer for all $(u, v) \in V \times V$.

Lemma 24.9
Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$, and let $G' = (V', E')$ be its corresponding flow network. If M is a matching in G, then there is an integer-valued flow f in G' with value $|f| = |M|$. Conversely, if f is an integer-valued flow in G', then there is a matching M in G with cardinality $|M| = |f|$ consisting of edges $(u, v) \in E$ such that $f(u, v) > 0$.

Proof We first show that a matching M in G corresponds to an integer-valued flow f in G'. Define f as follows. If $(u, v) \in M$, then $f(s, u) = f(u, v) = f(v, t) = 1$. For all other edges $(u, v) \in E'$, define $f(u, v) = 0$. It is simple to verify that f satisfies the capacity constraint and flow conservation.

Intuitively, each edge $(u, v) \in M$ corresponds to 1 unit of flow in G' that traverses the path $s \to u \to v \to t$. Moreover, the paths induced by edges in M are vertex-disjoint, except for s and t. The net flow across cut $(L \cup \{s\}, R \cup \{t\})$ is equal to $|M|$, and thus, by Lemma 24.4, the value of the flow is $|f| = |M|$.

To prove the converse, let f be an integer-valued flow in G' and, as in the statement of the lemma, let

$$M = \{(u, v) : u \in L,\ v \in R,\ \text{and}\ f(u, v) > 0\}\ .$$

Each vertex $u \in L$ has only one entering edge, namely (s, u), and its capacity is 1. Thus, each $u \in L$ has at most 1 unit of flow entering it, and if 1 unit of flow does enter, by flow conservation, 1 unit of flow must leave. Furthermore, since the flow f is integer-valued, for each $u \in L$, the 1 unit of flow can enter on at most one edge and can leave on at most one edge. Thus, 1 unit of flow enters u if and only if there is exactly one vertex $v \in R$ such that $f(u, v) = 1$, and at most one edge leaving each $u \in L$ carries positive flow. A symmetric argument applies to each $v \in R$. The set M is therefore a matching.

To see that $|M| = |f|$, observe that of the edges $(u, v) \in E'$ such that $u \in L$ and $v \in R$,

$$f(u, v) = \begin{cases} 1 & \text{if } (u, v) \in M\ , \\ 0 & \text{if } (u, v) \notin M\ . \end{cases}$$

Consequently, $f(L \cup \{s\}, R \cup \{t\})$, the net flow across cut $(L \cup \{s\}, R \cup \{t\})$, is equal to $|M|$. Lemma 24.4 gives that $|f| = f(L \cup \{s\}, R \cup \{t\}) = |M|$. ■

Based on Lemma 24.9, we would like to conclude that a maximum matching in a bipartite graph G corresponds to a maximum flow in its corresponding flow

network G', and therefore running a maximum-flow algorithm on G' provides a maximum matching in G. The only hitch in this reasoning is that the maximum-flow algorithm might return a flow in G' for which some $f(u, v)$ is not an integer, even though the flow value $|f|$ must be an integer. The following theorem shows that the Ford-Fulkerson method cannot produce a solution with this problem.

Theorem 24.10 (Integrality theorem)
If the capacity function c takes on only integer values, then the maximum flow f produced by the Ford-Fulkerson method has the property that $|f|$ is an integer. Moreover, for all vertices u and v, the value of $f(u, v)$ is an integer.

Proof Exercise 24.3-2 asks you to provide the proof by induction on the number of iterations. ■

We can now prove the following corollary to Lemma 24.9.

Corollary 24.11
The cardinality of a maximum matching M in a bipartite graph G equals the value of a maximum flow f in its corresponding flow network G'.

Proof We use the nomenclature from Lemma 24.9. Suppose that M is a maximum matching in G and that the corresponding flow f in G' is not maximum. Then there is a maximum flow f' in G' such that $|f'| > |f|$. Since the capacities in G' are integer-valued, by Theorem 24.10, we can assume that f' is integer-valued. Thus, f' corresponds to a matching M' in G with cardinality $|M'| = |f'| > |f| = |M|$, contradicting our assumption that M is a maximum matching. In a similar manner, we can show that if f is a maximum flow in G', its corresponding matching is a maximum matching on G. ■

Thus, to find a maximum matching in a bipartite undirected graph G, create the flow network G', run the Ford-Fulkerson method on G', and convert the integer-valued maximum flow found into a maximum matching for G. Since any matching in a bipartite graph has cardinality at most $\min\{|L|, |R|\} = O(V)$, the value of the maximum flow in G' is $O(V)$. Therefore, finding a maximum matching in a bipartite graph takes $O(VE') = O(VE)$ time, since $|E'| = \Theta(E)$.

Exercises

24.3-1
Run the Ford-Fulkerson algorithm on the flow network in Figure 24.8(c) and show the residual network after each flow augmentation. Number the vertices in L top

to bottom from 1 to 5 and in R top to bottom from 6 to 9. For each iteration, pick the augmenting path that is lexicographically smallest.

24.3-2

Prove Theorem 24.10. Use induction on the number of iterations of the Ford-Fulkerson method.

24.3-3

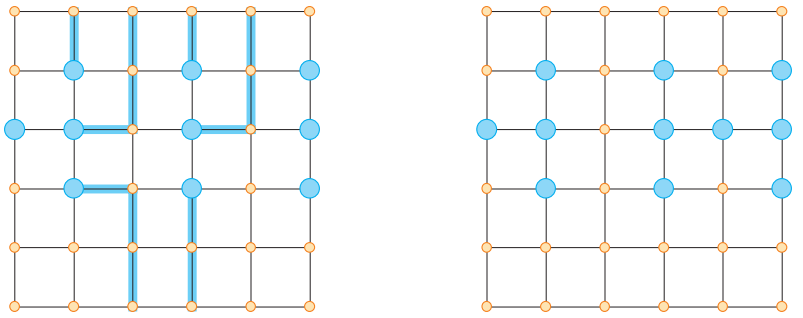
Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$, and let G' be its corresponding flow network. Give a good upper bound on the length of any augmenting path found in G' during the execution of FORD-FULKERSON.

Problems

24-1 Escape problem

An $n \times n$ ***grid*** is an undirected graph consisting of n rows and n columns of vertices, as shown in Figure 24.9. We denote the vertex in the ith row and the jth column by (i, j). All vertices in a grid have exactly four neighbors, except for the boundary vertices, which are the points (i, j) for which $i = 1$, $i = n$, $j = 1$, or $j = n$.

Given $m \leq n^2$ starting points $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ in the grid, the ***escape problem*** is to determine whether there are m vertex-disjoint paths from the starting points to any m different points on the boundary. For example, the grid in Figure 24.9(a) has an escape, but the grid in Figure 24.9(b) does not.



(a) (b)

Figure 24.9 Grids for the escape problem. Starting points are blue, and other grid vertices are tan. **(a)** A grid with an escape, shown by blue paths. **(b)** A grid with no escape.

a. Consider a flow network in which vertices, as well as edges, have capacities. That is, the total positive flow entering any given vertex is subject to a capacity constraint. Show how to reduce the problem of determining the maximum flow in a network with edge and vertex capacities to an ordinary maximum-flow problem on a flow network of comparable size.

b. Describe an efficient algorithm to solve the escape problem, and analyze its running time.

24-2 Minimum path cover

A ***path cover*** of a directed graph $G = (V, E)$ is a set P of vertex-disjoint paths such that every vertex in V is included in exactly one path in P. Paths may start and end anywhere, and they may be of any length, including 0. A ***minimum path cover*** of G is a path cover containing the fewest possible paths.

a. Give an efficient algorithm to find a minimum path cover of a directed acyclic graph $G = (V, E)$. (*Hint:* Assuming that $V = \{1, 2, \ldots, n\}$, construct a flow network based on the graph $G' = (V', E')$, where

$$V' = \{x_0, x_1, \ldots, x_n\} \cup \{y_0, y_1, \ldots, y_n\} ,$$
$$E' = \{(x_0, x_i) : i \in V\} \cup \{(y_i, y_0) : i \in V\} \cup \{(x_i, y_j) : (i, j) \in E\} ,$$

and run a maximum-flow algorithm.)

b. Does your algorithm work for directed graphs that contain cycles? Explain.

24-3 Hiring consulting experts

Professor Fieri wants to open a consulting company for the food industry. He has identified n important food categories, which he represents by the set $C = \{C_1, C_2, \ldots, C_n\}$. In each category C_k, he can hire an expert in that category for $e_k > 0$ dollars. The consulting company has lined up a set $J = \{J_1, J_2, \ldots, J_m\}$ of potential jobs. In order to perform job J_i, the company needs to have hired experts in a subset $R_i \subseteq C$ of categories. Each expert can work on multiple jobs simultaneously. If the company chooses to accept job J_i, it must have hired experts in all categories in R_i, and it takes in revenue of $p_i > 0$ dollars.

Professor Fieri's job is to determine which categories to hire experts in and which jobs to accept in order to maximize the net revenue, which is the total income from jobs accepted minus the total cost of employing the experts.

Consider the following flow network G. It contains a source vertex s, vertices $C_1, C_2, \ldots, C_n$, vertices $J_1, J_2, \ldots, J_m$, and a sink vertex t. For $k = 1, 2 \ldots, n$, the flow network contains an edge (s, C_k) with capacity $c(s, C_k) = e_k$, and for $i = 1, 2, \ldots, m$, the flow network contains an edge (J_i, t) with capacity

$c(J_i, t) = p_i$. For $k = 1, 2, \ldots, n$ and $i = 1, 2, \ldots, m$, if $C_k \in R_i$, then G contains an edge (C_k, J_i) with capacity $c(C_k, J_i) = \infty$.

a. Show that if $J_i \in T$ for a finite-capacity cut (S, T) of G, then $C_k \in T$ for each $C_k \in R_i$.

b. Show how to determine the maximum net revenue from the capacity of a minimum cut of G and the given p_i values.

c. Give an efficient algorithm to determine which jobs to accept and which experts to hire. Analyze the running time of your algorithm in terms of m, n, and $r = \sum_{i=1}^{m} |R_i|$.

24-4 Updating maximum flow

Let $G = (V, E)$ be a flow network with source s, sink t, and integer capacities. Suppose that you are given a maximum flow in G.

a. Suppose that the capacity of a single edge $(u, v) \in E$ increases by 1. Give an $O(V + E)$-time algorithm to update the maximum flow.

b. Suppose that the capacity of a single edge $(u, v) \in E$ decreases by 1. Give an $O(V + E)$-time algorithm to update the maximum flow.

24-5 Maximum flow by scaling

Let $G = (V, E)$ be a flow network with source s, sink t, and an integer capacity $c(u, v)$ on each edge $(u, v) \in E$. Let $C = \max\{c(u, v) : (u, v) \in E\}$.

a. Argue that a minimum cut of G has capacity at most $C\,|E|$.

b. For a given number K, show how to find an augmenting path of capacity at least K in $O(E)$ time, if such a path exists.

The procedure MAX-FLOW-BY-SCALING appearing on the following page modifies the basic FORD-FULKERSON-METHOD procedure to compute a maximum flow in G.

c. Argue that MAX-FLOW-BY-SCALING returns a maximum flow.

d. Show that the capacity of a minimum cut of the residual network G_f is less than $2K\,|E|$ each time line 4 executes.

e. Argue that the inner **while** loop of lines 5–6 executes $O(E)$ times for each value of K.

```
MAX-FLOW-BY-SCALING(G, s, t)
1  C = max{c(u, v) : (u, v) ∈ E}
2  initialize flow f to 0
3  K = 2^⌊lg C⌋
4  while K ≥ 1
5      while there exists an augmenting path p of capacity at least K
6          augment flow f along p
7      K = K/2
8  return f
```

f. Conclude that MAX-FLOW-BY-SCALING can be implemented so that it runs in $O(E^2 \lg C)$ time.

24-6 Widest augmenting path

The Edmonds-Karp algorithm implements the Ford-Fulkerson algorithm by always choosing a shortest augmenting path in the residual network. Suppose instead that the Ford-Fulkerson algorithm chooses a ***widest augmenting path***: an augmenting path with the greatest residual capacity. Assume that $G = (V, E)$ is a flow network with source s and sink t, that all capacities are integer, and that the largest capacity is C. In this problem, you will show that choosing a widest augmenting path results in at most $|E| \ln |f^*|$ augmentations to find a maximum flow f^*.

a. Show how to adjust Dijkstra's algorithm to find the widest augmenting path in the residual network.

b. Show that a maximum flow in G can be formed by successive flow augmentations along at most $|E|$ paths from s to t.

c. Given a flow f, argue that the residual network G_f has an augmenting path p with residual capacity $c_f(p) \geq (|f^*| - |f|)/|E|$.

d. Assuming that each augmenting path is a widest augmenting path, let f_i be the flow after augmenting the flow by the ith augmenting path, where f_0 has $f(u, v) = 0$ for all edges (u, v). Show that $|f^*| - |f_i| \leq |f^*|(1 - 1/|E|)^i$.

e. Show that $|f^*| - |f_i| < |f^*| e^{-i/|E|}$.

f. Conclude that after the flow is augmented at most $|E| \ln |f^*|$ times, the flow is a maximum flow.

24-7 Global minimum cut

A ***global cut*** in an undirected graph $G = (V, E)$ is a partition (see page 1156) of V into two nonempty sets V_1 and V_2. This definition is like the definition of cut that we have used in this chapter, except that we no longer have distinguished vertices s and t. Any edge (u, v) with $u \in V_1$ and $v \in V_2$ is said to ***cross*** the cut.

We can extend this definition of a cut to a multigraph $G = (V, E)$ (see page 1167), and we denote by $c(u, v)$ the number of edges in the multigraph with endpoints u and v. A global cut in a multigraph is still a partition of the vertices, and the value of a global cut (V_1, V_2) is $c(V_1, V_2) = \sum_{u \in V_1, v \in V_2} c(u, v)$. A solution to the ***global-minimum-cut problem*** is a cut (V_1, V_2) such that $c(V_1, V_2)$ is minimum. Let $\mu(G)$ denote the value of a global minimum cut in a graph or multigraph G.

a. Show how to find a global minimum cut of a graph $G = (V, E)$ by solving $\binom{|V|}{2}$ maximum-flow problems, each with a different pair of vertices as the source and sink, and taking the mininum value of the cuts found.

b. Give an algorithm to find a global minimum cut by solving only $\Theta(V)$ maximum-flow problems. What is the running time of your algorithm?

The remainder of this problem develops an algorithm for the global-minimum-cut problem that does not use any maximum-flow computations. It uses the notion of an edge contraction, defined on page 1168, with one crucial difference. The algorithm maintains a multigraph, so that upon contracting an edge (u, v), it creates a new vertex x, and for any other vertex $y \in V$, the number of edges between x and y is $c(u, y) + c(v, y)$. The algorithm does not maintain self-loops, and so it sets $c(x, x)$ to 0. Denote by $G/(u, v)$ the multigraph that results from contracting edge (u, v) in multigraph G.

Consider what can happen to the minimum cut when an edge is contracted. Assume that, at all points, the minimum cut in a multigraph G is unique. We'll remove this assumption later.

c. Show that for any edge (u, v), we have $\mu(G/(uv)) \geq \mu(G)$. Under what conditions is $\mu(G/(uv)) < \mu(G)$?

Next, you will show that if you pick an edge uniformly at random, the probability that it belongs to the minimum cut is small.

d. Show that for any multigraph $G = (V, E)$, the value of the global minimum cut is at most the average degree of a vertex: that $\mu(G) \leq 2|E|/|V|$, where $|E|$ denotes the total number of edges in the multigraph.

e. Using the results from parts (c) and (d), show that, if we pick an edge (u, v) uniformly at random, then the probability that (u, v) belongs to the minimum cut is at most $2/V$.

Consider the algorithm that repeatedly chooses an edge at random and contracts it until the multigraph has exactly two vertices, say u and v. At that point, the multigraph corresponds to a cut in the original graph, with vertex u representing all the nodes in one side of the original graph, and v representing all the vertices on the other side. The number of edges given by $c(u, v)$ corresponds exactly to the number of edges crossing the corresponding cut in the original graph. We call this algorithm the ***contraction algorithm***.

f. Suppose that the contraction algorithm terminates with a multigraph whose only vertices are u and v. Show that $\Pr\{c(u, v) = \mu(G)\} = \Omega\left(1/\binom{|V|}{2}\right)$.

g. Prove that if the contraction algorithm repeats $\binom{|V|}{2} \ln |V|$ times, then the probability that at least one of the runs returns the minimum cut is at least $1 - 1/|V|$.

h. Give a detailed implementation of the contraction algorithm that runs in $O(V^2)$ time.

i. Combine the previous parts and remove the assumption that the minimum cut must be unique, to conclude that running the contraction algorithm $\binom{|V|}{2} \ln |V|$ times yields an algorithm that runs in $O(V^4 \lg V)$ time and returns a minimum cut with probability at least $1 - 1/V$.

Chapter notes

Ahuja, Magnanti, and Orlin [7], Even [137], Lawler [276], Papadimitriou and Steiglitz [353], Tarjan [429], and Williamson [458] are good references for network flows and related algorithms. Schrijver [399] has written an interesting review of historical developments in the field of network flows.

The Ford-Fulkerson method is due to Ford and Fulkerson [149], who originated the formal study of many of the problems in the area of network flow, including the maximum-flow and bipartite-matching problems. Many early implementations of the Ford-Fulkerson method found augmenting paths using breadth-first search. Edmonds and Karp [132], and independently Dinic [119], proved that this strategy yields a polynomial-time algorithm. A related idea, that of using "blocking flows," was also first developed by Dinic [119].

A class of algorithms known as ***push-relabel algorithms***, due to Goldberg [185] and Goldberg and Tarjan [188], takes a different approach from the Ford-Fulkerson

method. Push-relabel algorithms allow flow conservation to be violated at vertices other than the source and sink as they execute. Using an idea first developed by Karzonov [251], they allow a ***preflow*** in which the flow into a vertex may exceed the flow out of the vertex. Such a vertex is said to be ***overflowing***. Initially, every edge leaving the source is filled to capacity, so that all neighbors of the source are overflowing. In a push-relabel algorithm, each vertex is assigned an integer height. An overflowing vertex may push flow to a neighboring vertex to which it has a residual edge provided that it is higher than the neighbor. If all residual edges from an overflowing vertex go to neighbors with equal or greater heights, then the vertex may increase its height. Once all vertices other than the sink are no longer overflowing, the preflow is not only a legal flow, but also a maximum flow.

Goldberg and Tarjan [188] gave an $O(V^3)$-time algorithm that uses a queue to maintain the set of overflowing vertices, as well as an algorithm that uses dynamic trees to achieve a running time of $O(VE \lg(V^2/E + 2))$. Several other researchers developed improved variants and implementations [9, 10, 15, 86, 87, 255, 358], the fastest of which, by King, Rao, and Tarjan [255], runs in $O(VE \log_{E/(V \lg V)} V)$ time.

Another efficient algorithm for maximum flow, by Goldberg and Rao [187], runs in $O\left(\min\left\{V^{2/3}, E^{1/2}\right\} E \lg(V^2/E + 2) \lg C\right)$ time, where C is the maximum capacity of any edge. Orlin [350] gave an algorithm in the same spirit as this algorithm that runs in $O(VE + E^{31/16} \lg^2 V)$ time. Combining it with the algorithm of King, Rao, and Tarjan results in an $O(VE)$-time algorithm.

A different approach to maximum flows and related problems is to use techniques from continuous optimization including electrical flows and interior-point methods. The first breakthrough in this line of work is due to Madry [308], who gave an $\widetilde{O}(E^{10/7})$-time algorithm for unit-capacity maximum flow and bipartite maximum matching. (See Problem 3-6 on page 73 for a definition of $\widetilde{O}$.) There has been a series of papers in this area for matchings, maximum flows, and minimum-cost flows. The fastest algorithm to date in this line of work for maximum flow is due to Lee and Sidford [285], taking $\widetilde{O}(\sqrt{V}E \lg^{O(1)} C)$ time. If the capacities are not too large, this algorithm is faster than the $O(VE)$-time algorithm mentioned above. Another algorithm, due to Liu and Sidford [303] runs in $\widetilde{O}(E^{11/8}C^{1/4})$ time, where C is the maximum capacity of any edge. This algorithm does not run in polynomial time, but for small enough capacities, it is faster than the previous ones.

In practice, push-relabel algorithms currently dominate algorithms based on augmenting paths, continuous-optimization, and linear programming for the maximum-flow problem [88] .

25 Matchings in Bipartite Graphs

Many real-world problems can be modeled as finding matchings in an undirected graph. For an undirected graph $G = (V, E)$, a ***matching*** is a subset of edges $M \subseteq E$ such that every vertex in V has at most one incident edge in M.

For example, consider the following scenario. You have one or more positions to fill and several candidates to interview. According to your schedule, you are able to interview candidates at certain time slots. You ask the candidates to indicate the subsets of time slots at which they are available. How can you schedule the interviews so that each time slot has at most one candidate scheduled, while maximizing the number of candidates that you can interview? You can model this scenario as a matching problem on a bipartite graph in which each vertex represents either a candidate or a time slot, with an edge between a candidate and a time slot if the candidate is available then. If an edge is included in the matching, that means you are scheduling a particular candidate for a particular time slot. Your goal is to find a ***maximum matching***: a matching of maximum cardinality. One of the authors of this book was faced with exactly this situation when hiring teaching assistants for a large class. He used the Hopcroft-Karp algorithm in Section 25.1 to schedule the interviews.

Another application of matching is the U.S. National Resident Matching Program, in which medical students are matched to hospitals where they will be stationed as medical residents. Each student ranks the hospitals by preference, and each hospital ranks the students. The goal is to assign students to hospitals so that there is never a student and a hospital that both have regrets because the student was not assigned to the hospital, yet each ranked the other higher than who or where they were assigned. This scenario is perhaps the best-known real-world example of the "stable-marriage problem," which Section 25.2 examines.

Yet another instance where matching comes into play occurs when workers must be assigned to tasks in order to maximize the overall effectiveness of the assignment. For each worker and each task, the worker has some quantified effectiveness

for that task. Assuming that there are equal numbers of workers and tasks, the goal is to find a matching with the maximum total effectiveness. Such a situation is an example of an assignment problem, which Section 25.3 shows how to solve.

The algorithms in this chapter find matchings in ***bipartite*** graphs. As in Section 24.3, the input is an undirected graph $G = (V, E)$, where $V = L \cup R$, the vertex sets L and R are disjoint, and every edge in E is incident on one vertex in L and one vertex in R. A matching, therefore, matches vertices in L with vertices in R. In some applications, the sets L and R have equal cardinality, and in other applications they need not be the same size.

An undirected graph need not be bipartite for the concept of matching to apply. Matching in general undirected graphs has applications in areas such as scheduling and computational chemistry. It models problems in which you want to pair up entities, represented by vertices. Two vertices are adjacent if they represent compatible entities, and you need to find a large set of compatible pairs. Maximum-matching and maximum-weight matching problems on general graphs can be solved by polynomial-time algorithms whose running times are similar to those for bipartite matching, but the algorithms are significantly more complicated. Exercise 25.2-5 discusses the general version of the stable-marriage problem, known as the "stable-roommates problem." Although matching applies to general undirected graphs, this chapter deals only with bipartite graphs.

25.1 Maximum bipartite matching (revisited)

Section 24.3 demonstrated one way to find a maximum matching in a bipartite graph, by finding a maximum flow. This section provides a more efficient method, the Hopcroft-Karp algorithm, which runs in $O(\sqrt{V}E)$ time. Figure 25.1(a) shows a matching in an undirected bipartite graph. A vertex that has an incident edge in matching M is ***matched*** under M, and otherwise, it is ***unmatched***. A ***maximal matching*** is a matching M to which no other edges can be added, that is, for every edge $e \in E - M$, the edge set $M \cup \{e\}$ fails to be a matching. A maximum matching is always maximal, but the reverse does not always hold.

Many algorithms to find maximum matchings, the Hopcroft-Karp algorithm included, work by incrementally increasing the size of a matching. Given a matching M in an undirected graph $G = (V, E)$, an ***M-alternating path*** is a simple path whose edges alternate between being in M and being in $E - M$. Figure 25.1(b) depicts an ***M-augmenting path*** (sometimes called an augmenting path with respect to M): an M-alternating path whose first and last edges belong to $E - M$. Since an M-augmenting path contains one more edge in $E - M$ than in M, it must consist of an odd number of edges.

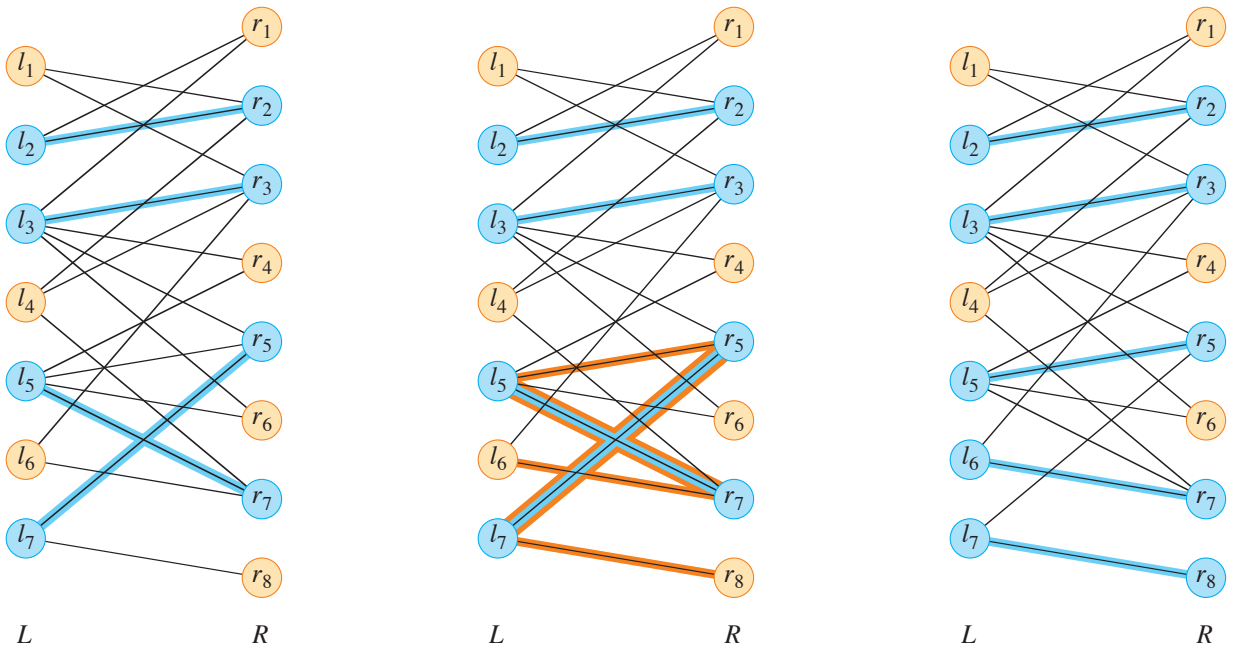


Figure 25.1 A bipartite graph, where $V = L \cup R$, $L = \{l_1, l_2, \ldots, l_7\}$, and $R = \{r_1, r_2, \ldots, r_8\}$. **(a)** A matching M with cardinality 4, highlighted in blue. Matched vertices are blue, and unmatched vertices are tan. **(b)** The five edges highlighted in orange form an M-augmenting path P going between vertices l_6 and r_8. **(c)** The set of edges $M' = M \oplus P$ highlighted in blue is a matching containing one more edge than M and adding l_6 and r_8 to the matched vertices. This matching is not a maximum matching (see Exercise 25.1-1).

Figure 25.1(c) demonstrates the following lemma, which shows that by removing from matching M the edges in an M-augmenting path that belong to M and adding to M the edges in the M-augmenting path that are not in M, the result is a new matching with one more edge than M. Since a matching is a set of edges, the lemma relies on the notion of the ***symmetric difference*** of two sets: $X \oplus Y = (X - Y) \cup (Y - X)$, that is, the elements that belong to X or Y, but not both. Alternatively, you can think of $X \oplus Y$ as $(X \cup Y) - (X \cap Y)$. The operator $\oplus$ is commutative and associative. Furthermore, $X \oplus X = \emptyset$ and $X \oplus \emptyset = \emptyset \oplus X = X$ for any set X, so that the empty set is the identity for $\oplus$.

Lemma 25.1

Let M be a matching in any undirected graph $G = (V, E)$, and let P be an M-augmenting path. Then the set of edges $M' = M \oplus P$ is also a matching in G with $|M'| = |M| + 1$.

Proof Let P contain q edges, so that $\lceil q/2 \rceil$ edges belong to $E - M$ and $\lfloor q/2 \rfloor$ edges belong to M, and let these q edges be $(v_1, v_2), (v_2, v_3), \ldots, (v_q, v_{q+1})$. Because P is an M-augmenting path, vertices v_1 and v_{q+1} are unmatched under M and all other vertices in P are matched. Edges $(v_1, v_2), (v_3, v_4), \ldots, (v_q, v_{q+1})$ belong to $E - M$, and edges $(v_2, v_3), (v_4, v_5), \ldots, (v_{q-1}, v_q)$ belong to M. The symmetric difference $M' = M \oplus P$ reverses these roles, so that edges $(v_1, v_2), (v_3, v_4), \ldots, (v_q, v_{q+1})$ belong to M' and $(v_2, v_3), (v_4, v_5), \ldots, (v_{q-1}, v_q)$ belong to $E - M'$. Each vertex $v_1, v_2, \ldots, v_q, v_{q+1}$ is matched under M', which gains one additional edge relative to M, and no other vertices or edges in G are affected by the change from M to M'. Hence, M' is a matching in G, and $|M'| = |M| + 1$. ∎

Since taking the symmetric difference of a matching M with an M-augmenting path increases the size of the matching by 1, the following corollary shows that taking the symmetric difference of M with k vertex-disjoint M-augmenting paths increases the size of the matching by k.

Corollary 25.2
Let M be a matching in any undirected graph $G = (V, E)$ and $P_1, P_2, \ldots, P_k$ be vertex-disjoint M-augmenting paths. Then the set of edges $M' = M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)$ is a matching in G with $|M'| = |M| + k$.

Proof Since the M-augmenting paths $P_1, P_2, \ldots, P_k$ are vertex-disjoint, we have that $P_1 \cup P_2 \cup \cdots \cup P_k = P_1 \oplus P_2 \oplus \cdots \oplus P_k$. Because the operator $\oplus$ is associative, we have

$$\begin{aligned} M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k) &= M \oplus (P_1 \oplus P_2 \oplus \cdots \oplus P_k) \\ &= (\cdots((M \oplus P_1) \oplus P_2) \oplus \cdots \oplus P_{k-1}) \oplus P_k \ . \end{aligned}$$

A simple induction on i using Lemma 25.1 shows that $M \oplus (P_1 \cup P_2 \cup \cdots \cup P_{i-1})$ is a matching in G containing $|M| + i - 1$ edges and that path P_i is an augmenting path with respect to $M \oplus (P_1 \cup P_2 \cup \cdots \cup P_{i-1})$. Each of these augmenting paths increases the size of the matching by 1, and so $|M'| = |M| + k$. ∎

As the Hopcroft-Karp algorithm goes from matching to matching, it will be useful to consider the symmetric difference between two matchings.

Lemma 25.3
Let M and M^* be matchings in graph $G = (V, E)$, and consider the graph $G' = (V, E')$, where $E' = M \oplus M^*$. Then, G' is a disjoint union of simple paths, simple cycles, and/or isolated vertices. The edges in each such simple path or simple cycle

alternate between M and M^*. If $|M^*| > |M|$, then G' contains at least $|M^*| - |M|$ vertex-disjoint M-augmenting paths.

Proof Each vertex in G' has degree 0, 1, or 2, since at most two edges of E' can be incident on a vertex: at most one edge from M and at most one edge from M^*. Therefore, each connected component of G' is either a singleton vertex, an even-length simple cycle with edges alternately in M and M^*, or a simple path with edges alternately in M and M^*. Since

$$\begin{aligned} E' &= M \oplus M^* \\ &= (M \cup M^*) - (M \cap M^*) \end{aligned}$$

and $|M^*| > |M|$, the edge set E' must contain $|M^*| - |M|$ more edges from M^* than from M. Because each cycle in G' has an even number of edges drawn alternately from M and M^*, each cycle has an equal number of edges from M and M^*. Therefore, the simple paths in G' account for there being $|M^*| - |M|$ more edges from M^* than M. Each path containing a different number of edges from M and M^* either starts and ends with edges from M, containing one more edge from M than from M^*, or starts and ends with edges from M^*, containing one more edge from M^* than from M. Because E' contains $|M^*| - |M|$ more edges from M^* than from M, there are at least $|M^*| - |M|$ paths of the latter type, and each one is an M-augmenting path. Because each vertex has at most two incident edges from E', these paths must be vertex-disjoint. ■

If an algorithm finds a maximum matching by incrementally increasing the size of the matching, how does it determine when to stop? The following corollary gives the answer: when there are no augmenting paths.

Corollary 25.4
Matching M in graph $G = (V, E)$ is a maximum matching if and only if G contains no M-augmenting path.

Proof We prove the contrapositive of both directions of the lemma statement. The contrapositive of the forward direction is straightforward. If there is an M-augmenting path P in G, then by Lemma 25.1, the matching $M \oplus P$ contains one more edge than M, meaning that M could not be a maximum matching.

To show the contrapositive of the backward direction—if M is not a maximum matching, then G contains an M-augmenting path—let M^* be a maximum matching in Lemma 25.3, so that $|M^*| > |M|$. Then G contains at least $|M^*| - |M| > 0$ vertex-disjoint M-augmenting paths. ■

We already have learned enough to create a maximum-matching algorithm that runs in $O(VE)$ time. Start with the matching M empty. Then repeatedly run a variant of either breadth-first search or depth-first search from an unmatched vertex that takes alternating paths until you find another unmatched vertex. Use the resulting M-augmenting path to increase the size of M by 1.

The Hopcroft-Karp algorithm

The Hopcroft-Karp algorithm improves the running time to $O(\sqrt{V}E)$. The procedure HOPCROFT-KARP is given an undirected bipartite graph, and it uses Corollary 25.2 to repeatedly increase the size of the matching M it finds. Corollary 25.4 proves that the algorithm is correct, since it terminates once there are no M-augmenting paths. It remains to show that the algorithm does run in $O(\sqrt{V}E)$ time. We'll see that the **repeat** loop of lines 2–5 iterates $O(\sqrt{V})$ times and how to implement line 3 so that it runs in $O(E)$ time in each iteration.

```
HOPCROFT-KARP(G)
1  M = ∅
2  repeat
3       let P = {P_1, P_2, ..., P_k} be a maximal set of vertex-disjoint
              shortest M-augmenting paths
4       M = M ⊕ (P_1 ∪ P_2 ∪ ··· ∪ P_k)
5  until P == ∅
6  return M
```

Let's first see how to find a maximal set of vertex-disjoint shortest M-augmenting paths in $O(E)$ time. There are three phases. The first phase forms a directed version G_M of the undirected bipartite graph G. The second phase creates a directed acyclic graph H from G_M via a variant of breadth-first search. The third phase finds a maximal set of vertex-disjoint shortest M-augmenting paths by running a variant of depth-first search on the transpose H^{T} of H. (Recall that the transpose of a directed graph reverses the direction of each edge. Since H is acyclic, so is H^{T}.)

Given a matching M, you can think of an M-augmenting path P as starting at an unmatched vertex in L, traversing an odd number of edges, and ending at an unmatched vertex in R. The edges in P traversed from L to R must belong to $E - M$, and the edges in P traversed from R to L must belong to M. The first phase, therefore, creates the directed graph G_M by directing the edges accordingly: $G_M = (V, E_M)$, where

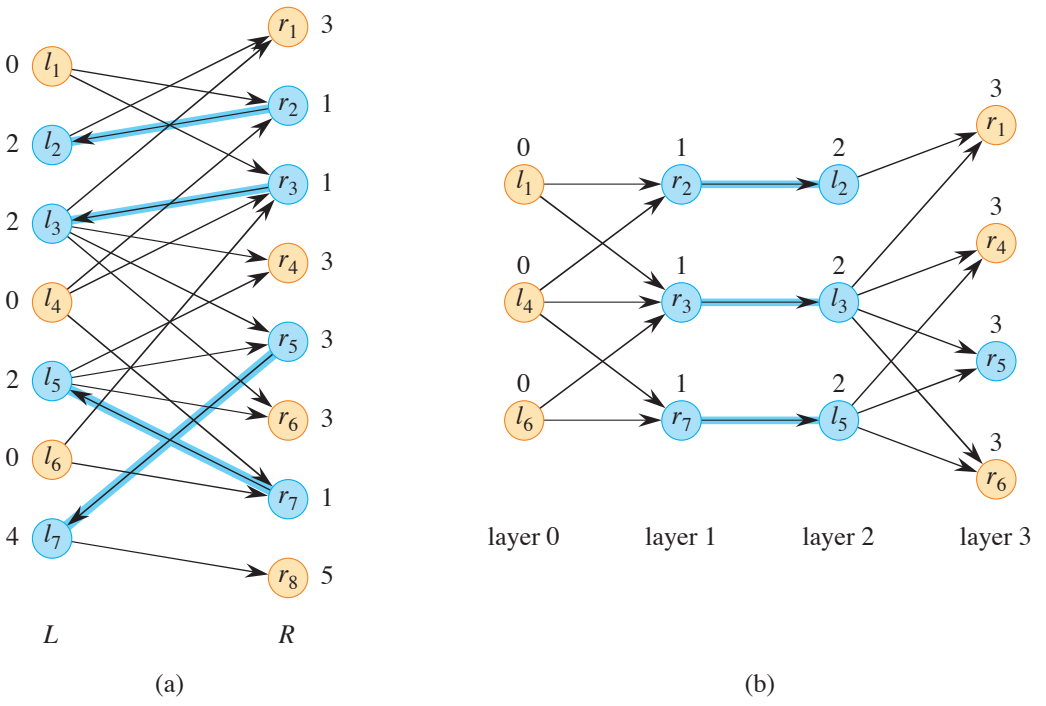


Figure 25.2 **(a)** The directed graph G_M created in the first phase for the undirected bipartite graph G and matching M in Figure 25.1(a). Breadth-first distances from any unmatched vertex in L appear next to each vertex. **(b)** The dag H created from G_M in the second phase. Because the smallest distance to an unmatched vertex in R is 3, vertices l_7 and r_8, with distances greater than 3, are not in H.

$$\begin{aligned} E_M = {} & \{(l,r) : l \in L, r \in R, \text{ and } (l,r) \in E - M\} \quad \text{(edges from } L \text{ to } R\text{)} \\ & \cup \{(r,l) : r \in R, l \in L, \text{ and } (l,r) \in M\} \quad \text{(edges from } R \text{ to } L\text{)} \,. \end{aligned}$$

Figure 25.2(a) shows the graph G_M for the graph G and matching M in Figure 25.1(a).

The dag $H = (V_H, E_H)$ created by the second phase has layers of vertices. Figure 25.2(b) shows the dag H corresponding to the directed graph G_M in part (a) of the figure. Each layer contains only vertices from L or only vertices from R, alternating from layer to layer. The layer that a vertex resides in is given by that vertex's minimum breadth-first distance in G_M from any unmatched vertex in L. Vertices in L appear in even-numbered layers, and vertices in R appear in odd-numbered layers. Let q denote the smallest distance in G_M of any unmatched vertex in R. Then, the last layer in H contains the vertices in R with distance q. Vertices whose distance exceeds q do not appear in V_H. (The graph H in Figure 25.2(b) omits vertices l_7 and r_8 because their distances from any unmatched vertex in L exceed $q = 3$.) The edges in E_H form a subset of E_M:

$$E_H = \{(l, r) \in E_M : r.d \le q \text{ and } r.d = l.d + 1\} \cup \{(r, l) \in E_M : l.d \le q\} \ ,$$

where the attribute d of a vertex gives the vertex's breadth-first distance in G_M from any unmatched vertex in L. Edges that do not go between two consecutive layers are omitted from E_H.

To determine the breadth-first distances of vertices, run breadth-first search on the graph G_M, but starting from all the unmatched vertices in L. (In the BFS procedure on page 556, replace the root vertex s by the set of unmatched vertices in L.) The predecessor attributes π computed by the BFS procedure are not needed here, since H is a dag and not necessarily a tree.

Every path in H from a vertex in layer 0 to an unmatched vertex in layer q corresponds to a shortest M-augmenting path in the original bipartite graph G. Just use the undirected versions of the directed edges in H. Moreover, every shortest M-augmenting path in G is present in H.

The third phase identifies a maximal set of vertex-disjoint shortest M-augmenting paths. As Figure 25.3 shows, it starts by creating the transpose H^{T} of H. Then, for each unmatched vertex r in layer q, it performs a depth-first search starting from r until it either reaches a vertex in layer 0 or has exhausted all possible paths without reaching a vertex in layer 0. Instead of maintaining discovery and finish times, the depth-first search just needs to keep track of the predecessor attributes π in the depth-first tree of each search. Upon reaching a vertex in layer 0, tracing back along the predecessors identifies an M-augmenting path. Each vertex is searched from only when it is first discovered in any search. If the search from a vertex r in layer q cannot find a path of undiscovered vertices to an undiscovered vertex in layer 0, then no M-augmenting path including r goes into the maximal set.

Figure 25.3 shows the result of the third phase. The first depth-first search starts from vertex r_1. It identifies the M-augmenting path $\langle(r_1, l_3), (l_3, r_3), (r_3, l_1)\rangle$, which is highlighted in orange, and discovers vertices r_1, l_3, r_3, and l_1. The next depth-first search starts from vertex r_4. This search first examines the edge (r_4, l_3), but because l_3 was already discovered, it backtracks and examines edge (r_4, l_5). From there, it continues and identifies the M-augmenting path $\langle(r_4, l_5), (l_5, r_7), (r_7, l_6)\rangle$, which is highlighted in yellow, and discovers vertices r_4, l_5, r_7, and l_6. The depth-first search from vertex r_6 gets stuck at vertices l_3 and l_5, which have already been discovered, and so this search fails to find a path of undiscovered vertices to a vertex in layer 0. There is no depth-first search from vertex r_5 because it is matched, and depth-first searches start from unmatched vertices. Therefore, the maximal set of vertex-disjoint shortest M-augmenting paths found contains just the two M-augmenting paths $(\langle r_1, l_3), (l_3, r_3), (r_3, l_1)\rangle$ and $\langle(r_4, l_5), (l_5, r_7), (r_7, l_6)\rangle$.

You might have noticed that in this example, this maximal set of two vertex-disjoint shortest M-augmenting paths is not a maximum set. The graph contains three vertex-disjoint shortest M-augmenting paths: $\langle(r_1, l_2), (l_2, r_2), (r_2, l_1)\rangle$, $\langle(r_4, l_3), (l_3, r_3), (r_3, l_4)\rangle$, and $\langle(r_6, l_5), (l_5, r_7), (r_7, l_6)\rangle$. No matter: the algorithm

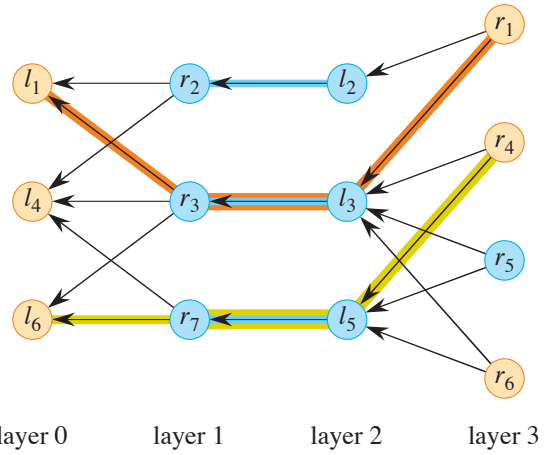


Figure 25.3 The transpose H^{T} of the dag H created in the third phase. The first depth-first search, starting from vertex r_1, identifies the M-augmenting path $\langle(r_1, l_3), (l_3, r_3), (r_3, l_1)\rangle$ highlighted in orange, and it discovers vertices r_1, l_3, r_3, l_1. The second depth-first search, starting from vertex r_4, identifies the M-augmenting path $\langle(r_4, l_5), (l_5, r_7), (r_7, l_6)\rangle$ highlighted in yellow, discovering vertices r_4, l_5, r_7, l_6.

requires the set of vertex-disjoint shortest M-augmenting paths found in line 3 of HOPCROFT-KARP to be only maximal, not necessarily maximum.

It remains to show that all three phases of line 3 take $O(E)$ time. We assume that in the original bipartite graph G, each vertex has at least one incident edge so that $|V| = O(E)$, which in turn implies that $|V| + |E| = O(E)$. The first phase creates the directed graph G_M by simply directing each edge of G, so that $|V_M| = |V|$ and $|E_M| = |E|$. The second phase performs a breadth-first search on G_M, taking $O(V_M + E_M) = O(E_M) = O(E)$ time. In fact, it can stop once the first distance in the queue within the breadth-first search exceeds the shortest distance q to an unmatched vertex in R. The dag H has $|V_H| \le |V_M|$ and $|E_H| \le |E_M|$, so that it takes $O(V_H + E_H) = O(E)$ time to construct. Finally, the third phase performs depth-first searches from the unmatched vertices in layer q. Once a vertex is discovered, it is not searched from again, and so the analysis of depth-first search from Section 20.3 applies here: $O(V_H + E_H) = O(E)$. Hence, all three phases take just $O(E)$ time.

Once the maximal set of vertex-disjoint shortest M-augmenting paths have been found in line 3, updating the matching in line 4 takes $O(E)$ time, as it is just a matter of going through the edges of the M-augmenting paths and adding edges to and removing edges from the matching M. Thus, each iteration of the **repeat** loop of lines 2–5 can run in $O(E)$ time.

It remains to show that the **repeat** loop iterates $O(\sqrt{V})$ times. We start with the following lemma, which shows that after each iteration of the **repeat** loop, the length of an augmenting path increases.

Lemma 25.5
Let $G = (V, E)$ be an undirected bipartite graph with matching M, and let q be the length of a shortest M-augmenting path. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ be a maximal set of vertex-disjoint M-augmenting paths of length q. Let $M' = M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)$, and suppose that P is a shortest M'-augmenting path. Then P has more than q edges.

Proof We consider separately the cases in which P is vertex-disjoint from the augmenting paths in $\mathcal{P}$ and in which it is not vertex-disjoint.

First, assume that P is vertex-disjoint from the augmenting paths in $\mathcal{P}$. Then, P contains edges that are in M but are not in any of $P_1, P_2, \ldots, P_k$, so that P is also an M-augmenting path. Since P is disjoint from $P_1, P_2, \ldots, P_k$ but is also an M-augmenting path, and since $\mathcal{P}$ is a maximal set of shortest M-augmenting paths, P must be longer than any of the augmenting paths in $\mathcal{P}$, each of which has length q. Therefore, P has more than q edges.

Now, assume that P visits at least one vertex from the M-augmenting paths in $\mathcal{P}$. By Corollary 25.2, M' is a matching in G with $|M'| = |M| + k$. Since P is an M'-augmenting path, by Lemma 25.1, $M' \oplus P$ is a matching with $|M' \oplus P| = |M'| + 1 = |M| + k + 1$. Now let $A = M \oplus M' \oplus P$. We claim that $A = (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P$:

$$
\begin{aligned}
A &= M \oplus M' \oplus P \\
&= M \oplus (M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)) \oplus P \\
&= (M \oplus M) \oplus (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P && \text{(associativity of } \oplus) \\
&= \emptyset \oplus (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P && (X \oplus X = \emptyset \text{ for all } X) \\
&= (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P && (\emptyset \oplus X = X \text{ for all } X) .
\end{aligned}
$$

Lemma 25.3 with $M^* = M' \oplus P$ gives that A contains at least $|M' \oplus P| - |M| = k + 1$ vertex-disjoint M-augmenting paths. Since each such M-augmenting path has at least q edges, we have $|A| \geq (k + 1)q = kq + q$.

Now we claim that P shares at least one edge with some M-augmenting path in $\mathcal{P}$. Under the matching M', every vertex in each M-augmenting path in $\mathcal{P}$ is matched. (Only the first and last vertex in each M-augmenting path P_i is unmatched under M, and under $M \oplus P_i$, all vertices in P_i are matched. Because the M-augmenting paths in $\mathcal{P}$ are vertex-disjoint, no other path in $\mathcal{P}$ can affect whether the vertices in P_i are matched. That is, the vertices in P_i are matched under $(M \oplus P_i) \oplus P_j$ if and only if they are matched under $M \oplus P_i$, for any other

path $P_j \in \mathcal{P}$.) Suppose that P shares a vertex v with some path $P_i \in \mathcal{P}$. Vertex v cannot be an endpoint of P, because the endpoints of P are unmatched under M'. Therefore, v has an incident edge in P that belongs to M'. Since any vertex has at most one incident edge in a matching, this edge must also belong to P_i, thus proving the claim.

Because $A = (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P$ and P shares at least one edge with some $P_i \in \mathcal{P}$, we have that $|A| < |P_1 \cup P_2 \cup \cdots \cup P_k| + |P|$. Thus, we have

$$\begin{aligned} kq + q &\le |A| \\ &< |P_1 \cup P_2 \cup \cdots \cup P_k| + |P| \\ &= kq + |P| \ , \end{aligned}$$

so that $q < |P|$. We conclude that P contains more than q edges. ■

The next lemma bounds the size of a maximum matching, based on the length of a shortest augmenting path.

Lemma 25.6
Let M be a matching in graph $G = (V, E)$, and let a shortest M-augmenting path in G contain q edges. Then the size of a maximum matching in G is at most $|M| + |V|/(q+1)$.

Proof Let M^* be a maximum matching in G. By Lemma 25.3, G contains at least $|M^*| - |M|$ vertex-disjoint M-augmenting paths. Each of these paths contains at least q edges, and hence at least $q+1$ vertices. Because these paths are vertex-disjoint, we have $(|M^*| - |M|)(q+1) \le |V|$, so that $|M^*| \le |M| + |V|/(q+1)$. ■

The final lemma bounds the number of iterations of the **repeat** loop of lines 2–5.

Lemma 25.7
When the HOPCROFT-KARP procedure runs on an undirected bipartite graph $G = (V, E)$, the **repeat** loop of lines 2–5 iterates $O(\sqrt{V})$ times.

Proof By Lemma 25.5, the length q of the shortest M-augmenting paths found in line 3 increases from iteration to iteration. After $\lceil \sqrt{|V|} \rceil$ iterations, therefore, we must have $q \ge \lceil \sqrt{|V|} \rceil$. Consider the situation after the first time line 4 executes with M-augmenting paths whose length is at least $\lceil \sqrt{|V|} \rceil$. Since the size of a matching increases by at least one edge per iteration, Lemma 25.6 implies that the number of additional iterations before achieving a maximum matching is at most

$$\frac{|V|}{\lceil \sqrt{|V|} \rceil + 1} < \frac{|V|}{\sqrt{|V|}} = \sqrt{|V|} \, .$$

Hence, the total number of loop iterations is less than $2\sqrt{|V|}$. ■

Thus, we have the following bound on the running time of the HOPCROFT-KARP procedure.

Theorem 25.8
The procedure HOPCROFT-KARP runs in $O(\sqrt{V}E)$ time on an undirected bipartite graph $G = (V, E)$.

Proof By Lemma 25.7 the **repeat** loop iterates $O(\sqrt{V})$ times, and we have seen how to implement each iteration in $O(E)$ time. ■

Exercises

25.1-1
Use the Hopcroft-Karp algorithm to find a maximum matching for the graph in Figure 25.1.

25.1-2
How are M-augmenting paths and augmenting paths in flow networks similar? How do they differ?

25.1-3
What is the advantage of searching in the transpose H^{T} from unmatched vertices in layer q (the first layer that contains an unmatched vertex in R) to layer 0 versus searching in the dag H from layer 0 to layer q?

25.1-4
Show how to bound the number of iterations of the the **repeat** loop of lines 2–5 of HOPCROFT-KARP by $\lceil 3\sqrt{|V|}/2 \rceil$.

★ ***25.1-5***
A ***perfect matching*** is a matching under which every vertex is matched. Let $G = (V, E)$ be an undirected bipartite graph with vertex partition $V = L \cup R$, where $|L| = |R|$. For any $X \subseteq V$, define the ***neighborhood*** of X as

$$N(X) = \{y \in V : (x, y) \in E \text{ for some } x \in X\} ,$$

that is, the set of vertices adjacent to some member of X. Prove ***Hall's theorem***: there exists a perfect matching in G if and only if $|A| \leq |N(A)|$ for every subset $A \subseteq L$.

25.1-6

In a ***d-regular*** graph, every vertex has degree d. If $G = (V, E)$ is bipartite with vertex partition $V = L \cup R$ and also d-regular, then $|L| = |R|$. Use Hall's theorem (see Exercise 25.1-5) to prove that every d-regular bipartite graph contains a perfect matching. Then use that result to prove that every d-regular bipartite graph contains d disjoint perfect matchings.

25.2 The stable-marriage problem

In Section 25.1, the goal was to find a maximum matching in an undirected bipartite graph. If you know that the graph $G = (V, E)$ with vertex partition $V = L \cup R$ is a ***complete bipartite graph***[1]—containing an edge from every vertex in L to every vertex in R—then you can find a maximum matching by a simple greedy algorithm.

When a graph can have several matchings, you might want to decide which matchings are most desirable. In Section 25.3, we'll add weights to the edges and find a matching of maximum weight. In this section, we will instead add some information to each vertex in a complete bipartite graph: a ranking of the vertices in the other side. That is, each vertex in L has an ordered list of all the vertices in R, and vice-versa. To keep things simple, let's assume that L and R each contain n vertices. The goal here is to match each vertex in L with a vertex in R in a "stable" way.

This problem derives its name, the ***stable-marriage problem***, from the notion of heterosexual marriage, viewing L as a set of women and R as a set of men.[2] Each woman ranks all the men in terms of desirability, and each man does the same with all the women. The goal is to pair up women and men (a matching) so that if a woman and a man are not matched to each other, then at least one of them prefers their assigned partner.

If a woman and a man are not matched to each other but each prefers the other over their assigned partner, they form a ***blocking pair***. A blocking pair has incentive to opt out of the assigned pairing and get together on their own. If that were to occur, then this pair would block the matching from being "stable." A ***stable***

[1] The definition of a complete bipartite graph differs from the definition of complete graph given on page 1167 because in a bipartite graph, there are no edges between vertices in L and no edges between vertices in R.

[2] Although marriage norms are changing, it's traditional to view the stable-marriage problem through the lens of heterosexual marriage.

matching, therefore, is a matching that has no blocking pair. If there is a blocking pair, then the matching is ***unstable***.

Let's look at an example with four women—Wanda, Emma, Lacey, and Karen—and four men—Oscar, Davis, Brent, and Hank—having the following preferences:

Wanda: Brent, Hank, Oscar, Davis
Emma: Davis, Hank, Oscar, Brent
Lacey: Brent, Davis, Hank, Oscar
Karen: Brent, Hank, Davis, Oscar

Oscar: Wanda, Karen, Lacey, Emma
Davis: Wanda, Lacey, Karen, Emma
Brent: Lacey, Karen, Wanda, Emma
Hank: Lacey, Wanda, Emma, Karen

A stable matching comprises the following pairs:

Lacey and Brent
Wanda and Hank
Karen and Davis
Emma and Oscar

You can verify that this matching has no blocking pair. For example, even though Karen prefers Brent and Hank to her partner Davis, Brent prefers his partner Lacey to Karen, and Hank prefers his partner Wanda to Karen, so that neither Karen and Brent nor Karen and Hank form a blocking pair. In fact, this stable matching is unique. Suppose instead that the last two pairs were

Emma and Davis
Karen and Oscar

Then Karen and Davis would be a blocking pair, because they were not paired together, Karen prefers Davis to Oscar, and Davis prefers Karen to Emma. Therefore, this matching is not stable.

Stable matchings need not be unique. For example, suppose that there are three women—Monica, Phoebe, and Rachel—and three men—Chandler, Joey, and Ross—with these preferences:

Monica: Chandler, Joey, Ross
Phoebe: Joey, Ross, Chandler
Rachel: Ross, Chandler, Joey

Chandler: Phoebe, Rachel, Monica
Joey: Rachel, Monica, Phoebe
Ross: Monica, Phoebe, Rachel

In this case, there are three stable matchings:

Matching 1	Matching 2	Matching 3
Monica and Chandler	Phoebe and Chandler	Rachel and Chandler
Phoebe and Joey	Rachel and Joey	Monica and Joey
Rachel and Ross	Monica and Ross	Phoebe and Ross

In matching 1, all women get their first choice and all men get their last choice. Matching 2 is the opposite, with all men getting their first choice and all women getting their last choice. When all the women or all the men get their first choice, there plainly cannot be a blocking pair. In matching 3, everyone gets their second choice. You can verify that there are no blocking pairs.

You might wonder whether it is always possible to come up with a stable matching no matter what rankings each participant provides. The answer is yes. (Exercise 25.2-3 asks you to show that even in the scenario of the National Resident Matching Program, where each hospital takes on multiple students, it is always possible to devise a stable assignment.) A simple algorithm known as the Gale-Shapley algorithm always finds a stable matching. The algorithm has two variants, which mirror each other: "woman-oriented" and "man-oriented." Let's examine the woman-oriented version. Each participant is either "free" or "engaged." Everyone starts out free. Engagements occur when a free woman proposes to a man. When a man is first proposed to, he goes from free to engaged, and he always stays engaged, though not necessarily to the same woman. If an engaged man receives a proposal from a woman whom he prefers to the woman he's currently engaged to, that engagement is broken, the woman to whom he had been engaged becomes free, and the man and the woman whom he prefers become engaged. Each woman proposes to the men in her preference list, in order, until the last time she becomes engaged. When a woman is engaged, she temporarily stops proposing, but if she becomes free again, she continues down her list. Once everyone is engaged, the algorithm terminates. The procedure GALE-SHAPLEY on the next page makes this process more concrete. The procedure allows for some choice: any free woman may be selected in line 2. We'll see that the procedure produces a stable matching regardless of the order in which line 2 chooses free women. For the man-oriented version, just reverse the roles of men and women in the procedure.

Let's see how the GALE-SHAPLEY procedure executes on the example with Wanda, Emma, Lacey, Karen, Oscar, Davis, Brent, and Hank. After everyone is initialized to free, here is one possible version of what can occur in successive iterations of the **while** loop of lines 2–9:

1. Wanda proposes to Brent. Brent is free, so that Wanda and Brent become engaged and no longer free.

```
GALE-SHAPLEY(men, women, rankings)
 1  assign each woman and man as free
 2  while some woman w is free
 3      let m be the first man on w's ranked list to whom she has not proposed
 4      if m is free
 5          w and m become engaged to each other (and not free)
 6      elseif m ranks w higher than the woman w' he is currently engaged to
 7          m breaks the engagement to w', who becomes free
 8          w and m become engaged to each other (and not free)
 9      else m rejects w, with w remaining free
10  return the stable matching consisting of the engaged pairs
```

2. Emma proposes to Davis. Davis is free, so that Emma and Davis become engaged and no longer free.
3. Lacey proposes to Brent. Brent is engaged to Wanda, but he prefers Lacey. Brent breaks the engagement to Wanda, who becomes free. Lacey and Brent become engaged, with Lacey no longer free.
4. Karen proposes to Brent. Brent is engaged to Lacey, whom he prefers to Karen. Brent rejects Karen, who remains free.
5. Karen proposes to Hank. Hank is free, so that Karen and Hank become engaged and no longer free.
6. Wanda proposes to Hank. Hank is engaged to Karen, but he prefers Wanda. Hank breaks the engagement with Karen, who becomes free. Wanda and Hank become engaged, with Wanda no longer free.
7. Karen proposes to Davis. Davis is engaged to Emma, but he prefers Karen. Davis breaks the engagement to Emma, who becomes free. Karen and Davis become engaged, with Karen no longer free.
8. Emma proposes to Hank. Hank is engaged to Wanda, whom he prefers to Emma. Hank rejects Emma, who remains free.
9. Emma proposes to Oscar. Oscar is free, so that Emma and Oscar become engaged and no longer free.

At this point, everyone is engaged and nobody is free, so the **while** loop terminates. The procedure returns the stable matching we saw earlier.

The following theorem shows that not only does GALE-SHAPLEY terminate, but that it always returns a stable matching, thereby proving that a stable matching always exists.

Theorem 25.9
The procedure GALE-SHAPLEY always terminates and returns a stable matching.

Proof Let's first show that the **while** loop of lines 2–9 always terminates, so that the procedure terminates. The proof is by contradiction. If the loop fails to terminate, it is because some woman remains free. In order for a woman to remain free, she must have proposed to all the men and been rejected by each one. In order for a man to reject a woman, he must be already engaged. Therefore, all the men are engaged. Once engaged, a man stays engaged (though not necessarily to the same woman). There are an equal number n of women and men, however, which means that every woman is engaged, leading to the contradiction that no women are free. We must also show that the **while** loop makes a bounded number of iterations. Since each of the n women goes through her ranking of the n men in order, possibly not reaching the end of her list, the total number of iterations is at most n^2. Therefore, the **while** loop always terminates, and the procedure returns a matching.

We need to show that there are no blocking pairs. We first observe that once a man m is engaged to a woman w, all subsequent actions for m occur in lines 6–8. Therefore, once a man is engaged, he stays engaged, and any time he breaks an engagement to a woman w, it's for a woman whom he prefers to w. Suppose that a woman w is matched with a man m, but she prefers man m'. We'll show that w and m' is not a blocking pair, because m' does not prefer w to his partner. Because w ranks m' higher than m, she must have proposed to m' before proposing to m, and m' either rejected her proposal or accepted it and later broke the engagement. If m' rejected the proposal from w, it is because he was already engaged to some woman he prefers to w. If m' accepted and later broke the engagement, he was at some point engaged to w but later accepted a proposal from a woman he prefers to w. In either case, he ultimately ends up with a partner whom he prefers to w. We conclude that even though w might prefer m' to her partner m, it is not also the case that m' prefers w to his partner. Therefore, the procedure returns a matching containing no blocking pairs. ■

Exercise 25.2-1 asks you to provide the proof of the following corollary.

Corollary 25.10
Given preference rankings for n women and n men, the Gale-Shapley algorithm can be implemented to run in $O(n^2)$ time. ■

Because line 2 can choose any free woman, you might wonder whether different choices can produce different stable matchings. The answer is no: as the following

theorem shows, every execution of the GALE-SHAPLEY produces exactly the same result. Moreover, the stable matching returned is optimal for the women.

Theorem 25.11
Regardless of how women are chosen in line 2 of GALE-SHAPLEY, the procedure always returns the same stable matching, and in this stable matching, each woman has the best partner possible in any stable matching.

Proof The proof that each woman has the best partner possible in any stable matching is by contradiction. Suppose that the GALE-SHAPLEY procedure returns a stable matching M, but that there is a different stable matching M' in which some woman w prefers her partner m' to the partner m she has in M. Because w ranks m' higher than m, she must have proposed to m' before proposing to m. Then there is a woman w' whom m' prefers to w, and m' was already engaged to w' when w proposed or m' accepted the proposal from w and later broke the engagement in favor of w'. Either way, there is a moment when m' decided against w in favor of w'. Now suppose, without loss of generality, that this moment was the first time that any man rejected a partner who belongs to some stable matching.

We claim that w' cannot have a partner m'' in a stable matching whom she prefers to m'. If there were such a man m'', then in order for w' to propose to m', she would have proposed to m'' and been rejected at some point before proposing to m'. If m' accepted the proposal from w and later broke it to accept w', then since this was the first rejection in a stable matching, we get the contradiction that m'' could not have rejected w' beforehand. If m' was already engaged to w' when w proposed, then again, m'' could not have rejected w' beforehand, thus proving the claim.

Since w' does not prefer anyone to m' in a stable matching and w' is not matched with m' in M' (because m' is matched with w in M'), w' prefers m' to her partner in M'. Since w' prefers m' over her partner in M' and m' prefers w' over his partner w in M', the pair w' and m' is a blocking pair in M'. Because M' has a blocking pair, it cannot be a stable matching, thereby contradicting the assumption that there exists some stable matching in which each woman has the best partner possible other than the matching M returned by GALE-SHAPLEY.

We put no condition on the execution of the procedure, which means that all possible orders in which line 2 selects women result in the same stable matching being returned. ■

Corollary 25.12
There can be stable matchings that the GALE-SHAPLEY procedure does not return.

Proof Theorem 25.11 says that for a given set of rankings, GALE-SHAPLEY returns just one matching, no matter how it chooses women in line 2. The earlier ex-

ample of three women and three men with three different stable matchings shows that there can be multiple stable matchings for a given set of rankings. A call of GALE-SHAPLEY is capable of returning only one of these stable matchings. ■

Although the GALE-SHAPLEY procedure gives the best possible outcome for the women, the following corollary shows that it also produces the worst possible outcome for the men.

Corollary 25.13
In the stable matching returned by the procedure GALE-SHAPLEY, each man has the worst partner possible in any stable matching.

Proof Let M be the matching returned by a call to GALE-SHAPLEY. Suppose that there is another stable matching M' and a man m who prefers his partner w in M to his partner w' in M'. Let the partner of w in M' be m'. By Theorem 25.11, m is the best partner that w can have in any stable matching, which means that w prefers m to m'. Since m prefers w to w', the pair w and m is a blocking pair in M', contradicting the assumption that M' is a stable matching. ■

Exercises

25.2-1
Describe how to implement the Gale-Shapley algorithm so that it runs in $O(n^2)$ time.

25.2-2
Is it possible to have an unstable matching with just two women and two men? If so, provide and justify an example. If not, argue why not.

25.2-3
The National Resident Matching Program differs from the scenario for the stable-marriage problem set out in this section in two ways. First, a hospital may be matched with more than one student, so that hospital h takes $r_h \geq 1$ students. Second, the number of students might not equal the number of hospitals. Describe how to modify the Gale-Shapley algorithm to fit the requirements of the National Resident Matching Program.

25.2-4
Prove the following property, which is known as ***weak Pareto optimality***:

> Let M be the stable matching produced by the GALE-SHAPLEY procedure, with women proposing to men. Then, for a given instance of the stable-marriage problem there is no matching—stable or unstable—such that every

woman has a partner whom she prefers to her partner in the stable matching M.

25.2-5

The ***stable-roommates problem*** is similar to the stable-marriage problem, except that the graph is a complete graph, not bipartite, with an even number of vertices. Each vertex represents a person, and each person ranks all the other people. The definitions of blocking pairs and stable matching extend in the natural way: a blocking pair comprises two people who both prefer each other to their current partner, and a matching is stable if there are no blocking pairs. For example, consider four people—Wendy, Xenia, Yolanda, and Zelda—with the following preference lists:

Wendy: Xenia, Yolanda, Zelda
Xenia: Wendy, Zelda, Yolanda
Yolanda: Wendy, Zelda, Xenia
Zelda: Xenia, Yolanda, Wendy

You can verify that the following matching is stable:

Wendy and Xenia
Yolanda and Zelda

Unlike the stable-marriage problem, the stable-roommates problem can have inputs for which no stable matching exists. Find such an input and explain why no stable matching exists.

25.3 The Hungarian algorithm for the assignment problem

Let us once again add some information to a complete bipartite graph $G = (V, E)$, where $V = L \cup R$. This time, instead of having the vertices of each side rank the vertices on the other side, we assign a weight to each edge. Again, let's assume that the vertex sets L and R each contain n vertices, so that the graph contains n^2 edges. For $l \in L$ and $r \in R$, denote the weight of edge (l, r) by $w(l, r)$, which represents the utility gained by matching vertex l with vertex r.

The goal is to find a perfect matching M^* (see Exercises 25.1-5 and 25.1-6) whose edges have the maximum total weight over all perfect matchings. That is, letting $w(M) = \sum_{(l,r)\in M} w(l, r)$ denote the total weight of the edges in matching M, we want to find a perfect matching M^* such that

$$w(M^*) = \max \{w(M) : M \text{ is a perfect matching}\} .$$

We call finding such a maximum-weight perfect matching the ***assignment problem***. A solution to the assignment problem is a perfect matching that maximizes the total utility. Like the stable-marriage problem, the assignment problem finds a matching that is "good," but with a different definition of good: maximizing total value rather than achieving stability.

Although you could enumerate all $n!$ perfect matchings to solve the assignment problem, an algorithm known as the ***Hungarian algorithm*** solves it much faster. This section will prove an $O(n^4)$ time bound, and Problem 25-2 asks you to refine the algorithm to reduce the running time to $O(n^3)$. Instead of working with the complete bipartite graph G, the Hungarian algorithm works with a subgraph of G called the "equality subgraph." The equality subgraph, which is defined below, changes over time and has the beneficial property that any perfect matching in the equality subgraph is also an optimal solution to the assignment problem.

The equality subgraph depends on assigning an attribute h to each vertex. We call h the ***label*** of a vertex, and we say that h is a ***feasible vertex labeling*** of G if

$$l.h + r.h \geq w(l, r) \text{ for all } l \in L \text{ and } r \in R \ .$$

A feasible vertex labeling always exists, such as the ***default vertex labeling*** given by

$$l.h = \max\{w(l, r) : r \in R\} \quad \text{for all } l \in L \ , \tag{25.1}$$

$$r.h = 0 \quad \text{for all } r \in R \ . \tag{25.2}$$

Given a feasible vertex labeling h, the ***equality subgraph*** $G_h = (V, E_h)$ of G consists of the same vertices as G and the subset of edges

$$E_h = \{(l, r) \in E : l.h + r.h = w(l, r)\} \ .$$

The following theorem ties together a perfect matching in an equality subgraph and an optimal solution to the assignment problem.

Theorem 25.14

Let $G = (V, E)$, where $V = L \cup R$, be a complete bipartite graph where each edge $(l, r) \in E$ has weight $w(l, r)$. Let h be a feasible vertex labeling of G and G_h be the equality subgraph of G. If G_h contains a perfect matching M^*, then M^* is an optimal solution to the assignment problem on G.

Proof If G_h contains a perfect matching M^*, then because G_h and G have the same sets of vertices, M^* is also a perfect matching in G. Because each edge of M^* belongs to G_h and each vertex has exactly one incident edge from any perfect matching, we have

$$\begin{aligned} w(M^*) &= \sum_{(l,r)\in M^*} w(l,r) \\ &= \sum_{(l,r)\in M^*} (l.h + r.h) \quad \text{(because all edges in } M^* \text{ belong to } G_h) \\ &= \sum_{l\in L} l.h + \sum_{r\in R} r.h \quad \text{(because } M^* \text{ is a perfect matching)} . \end{aligned}$$

Letting M be any perfect matching in G, we have

$$\begin{aligned} w(M) &= \sum_{(l,r)\in M} w(l,r) \\ &\le \sum_{(l,r)\in M} (l.h + r.h) \quad \text{(because } h \text{ is a feasible vertex labeling)} \\ &= \sum_{l\in L} l.h + \sum_{r\in R} r.h \quad \text{(because } M \text{ is a perfect matching)} . \end{aligned}$$

Thus, we have

$$w(M) \le \sum_{l\in L} l.h + \sum_{r\in R} r.h = w(M^*) , \tag{25.3}$$

so that M^* is a maximum-weight perfect matching in G. ■

The goal now becomes finding a perfect matching in an equality subgraph. Which equality subgraph? It does not matter! We have free rein to not only choose an equality subgraph, but to change which equality subgraph we choose as we go along. We just need to find *some* perfect matching in *some* equality subgraph.

To understand the equality subgraph better, consider again the proof of Theorem 25.14 and, in the second half, let M be any matching. The proof is still valid, in particular, inequality (25.3): the weight of any matching is always at most the sum of the vertex labels. If we choose any set of vertex labels that define an equality subgraph, then a maximum-cardinality matching in this equality subgraph has total value at most the sum of the vertex labels. If the set of vertex labels is the "right" one, then it will have total value equal to $w(M^*)$, and a maximum-cardinality matching in the equality subgraph is also a maximum-weight perfect matching. The Hungarian algorithm repeatedly modifies the matching and the vertex labels in order to achieve this goal.

The Hungarian algorithm starts with any feasible vertex labeling h and any matching M in the equality subgraph G_h. It repeatedly finds an M-augmenting path P in G_h and, using Lemma 25.1, updates the matching to be $M \oplus P$, thereby incrementing the size of the matching. As long as there is some equality subgraph that contains an M-augmenting path, the size of the matching can increase, until a perfect matching is achieved.

Four questions arise:

1. What initial feasible vertex labeling should the algorithm start with? Answer: the default vertex labeling given by equations (25.1) and (25.2).
2. What initial matching in G_h should the algorithm start with? Short answer: any matching, even an empty matching, but a greedy maximal matching works well.
3. If an M-augmenting path exists in G_h, how to find it? Short answer: use a variant of breadth-first search similar to the second phase of the procedure used in the Hopcroft-Karp algorithm to find a maximal set of shortest M-augmenting paths.
4. What if the search for an M-augmenting path fails? Short answer: update the feasible vertex labeling to bring in at least one new edge.

We'll elaborate on the short answers using the example that starts in Figure 25.4. Here, $L = \{l_1, l_2, \ldots, l_7\}$ and $R = \{r_1, r_2, \ldots, r_7\}$. The edge weights appear in the matrix shown in part (a), where the weight $w(l_i, r_j)$ appears in row i and column j. The feasible vertex labels, given by the default vertex labeling, appear to the left of and above the matrix. Matrix entries in red indicate edges (l_i, r_j) for which $l_i.h + r_j.h = w(l_i, r_j)$, that is, edges in the equality subgraph G_h appearing in part (b) of the figure.

Greedy maximal bipartite matching

There are several ways to implement a greedy method to find a maximal bipartite matching. The procedure GREEDY-BIPARTITE-MATCHING shows one. Edges in Figure 25.4(b) highlighted in blue indicate the initial greedy maximal matching in G_h. Exercise 25.3-2 asks you to show that the GREEDY-BIPARTITE-MATCHING procedure returns a matching that is at least half the size of a maximum matching.

```
GREEDY-BIPARTITE-MATCHING(G)
1  M = ∅
2  for each vertex l ∈ L
3      if l has an unmatched neighbor in R
4          choose any such unmatched neighbor r ∈ R
5          M = M ∪ {(l, r)}
6  return M
```

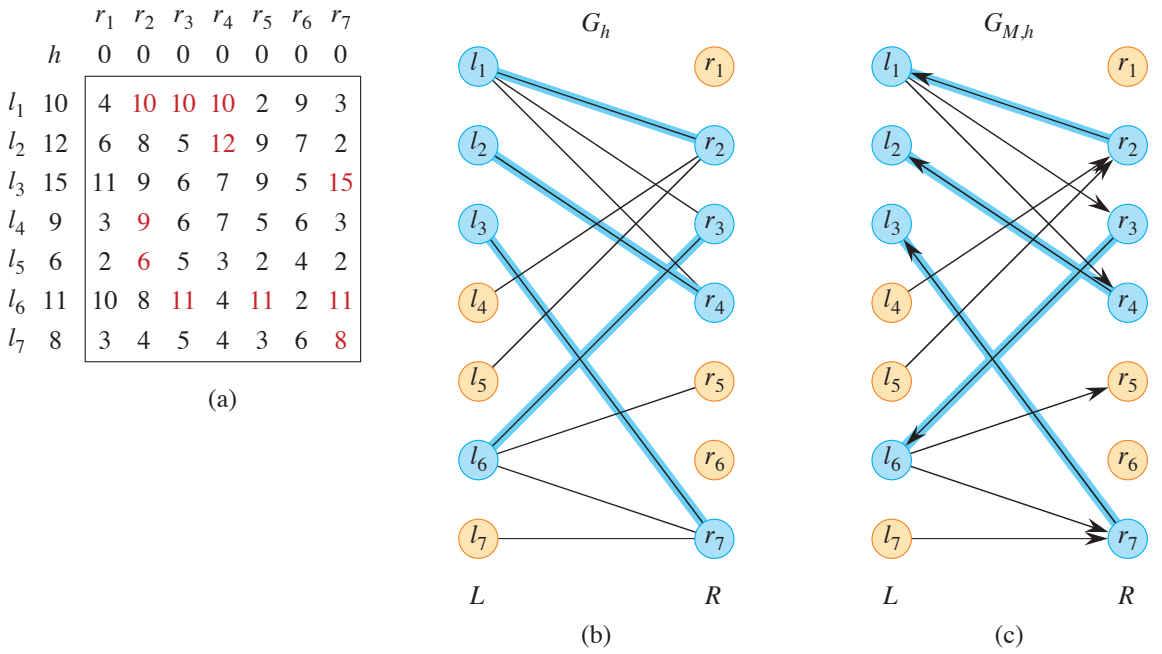


Figure 25.4 The start of the Hungarian algorithm. **(a)** The matrix of edge weights for a bipartite graph with $L = \{l_1, l_2, \ldots, l_7\}$ and $R = \{r_1, r_2, \ldots, r_7\}$. The value in row i and column j indicates $w(l_i, r_j)$. Feasible vertex labels appear above and next to the matrix. Red entries correspond to edges in the equality subgraph. **(b)** The equality subgraph G_h. Edges highlighted in blue belong to the initial greedy maximal matching M. Blue vertices are matched, and tan vertices are unmatched. **(c)** The directed equality subgraph $G_{M,h}$ created from G_h by directing edges in M from R to L and all other edges from L to R.

Finding an M-augmenting path in G_h

To find an M-augmenting path in the equality subgraph G_h with a matching M, the Hungarian algorithm first creates the ***directed equality subgraph*** $G_{M,h}$ from G_h, just as the Hopcroft-Karp algorithm creates G_M from G. As in the Hopcroft-Karp algorithm, you can think of an M-augmenting path as starting from an unmatched vertex in L, ending at an unmatched vertex in R, taking unmatched edges from L to R, and taking matched edges from R to L. Thus, $G_{M,h} = (V, E_{M,h})$, where

$$\begin{aligned} E_{M,h} = {} & \{(l,r) : l \in L, r \in R, \text{ and } (l,r) \in E_h - M\} && \text{(edges from } L \text{ to } R\text{)} \\ & \cup \{(r,l) : r \in R, l \in L, \text{ and } (l,r) \in M\} && \text{(edges from } R \text{ to } L\text{)} . \end{aligned}$$

Because an M-augmenting path in the directed equality subgraph $G_{M.h}$ is also an M-augmenting path in the equality subgraph G_h, it suffices to find M-augmenting paths in $G_{M.h}$. Figure 25.4(c) shows the directed equality subgraph $G_{M,h}$ corresponding to the equality subgraph G_h and matching M from part (b) of the figure.

With the directed equality subgraph $G_{M,h}$ in hand, the Hungarian algorithm searches for an M-augmenting path from any unmatched vertex in L to any unmatched vertex in R. Any exhaustive graph-search method suffices. Here, we'll use breadth-first search, starting from all the unmatched vertices in L (just as the Hopcroft-Karp algorithm does when creating the dag H), but stopping upon first discovering some unmatched vertex in R. Figure 25.5 shows the idea. To start from all the unmatched vertices in L, initialize the first-in, first-out queue with all the unmatched vertices in L, rather than just one source vertex. Unlike the dag H in the Hopcroft-Karp algorithm, here each vertex needs just one predecessor, so that the breadth-first search creates a ***breadth-first forest*** $F = (V_F, E_F)$. Each unmatched vertex in L is a root in F.

In Figure 25.5(g), the breadth-first search has found the M-augmenting path $\langle (l_4, r_2), (r_2, l_1), (l_1, r_3), (r_3, l_6), (l_6, r_5) \rangle$. Figure 25.6(a) shows the new matching created by taking the symmetric difference of the matching M in Figure 25.5(a) with this M-augmenting path.

When the search for an M-augmenting path fails

Having updated the matching M from an M-augmenting path, the Hungarian algorithm updates the directed equality subgraph $G_{M,h}$ according to the new matching and then starts a new breadth-first search from all the unmatched vertices in L. Figure 25.6 shows the start of this process, picking up from Figure 25.5.

In Figure 25.6(d), the queue contains vertices l_4 and l_3. Neither of these vertices has an edge that leaves it, however, so that once these vertices are removed from the queue, the queue becomes empty. The search terminates at this point, before discovering an unmatched vertex in R to yield an M-augmenting path. Whenever this situation occurs, the most recently discovered vertices must belong to L. Why? Whenever an unmatched vertex in R is discovered, the search has found an M-augmenting path, and when a matched vertex in R is discovered, it has an unvisited neighbor in L, which the search can then discover.

Recall that we have the freedom to work with any equality subgraph. We can change the directed equality subgraph "on the fly," as long we do not counteract the work already done. The Hungarian algorithm updates the feasible vertex labeling h to fulfill the following criteria:

1. No edge in the breadth-first forest F leaves the directed equality subgraph.
2. No edge in the matching M leaves the directed equality subgraph.
3. At least one edge (l, r), where $l \in L \cap V_F$ and $r \in R - V_F$ goes into E_h, and hence into $E_{M,h}$. Therefore, at least one vertex in R will be newly discovered.

Thus, at least one new edge enters the directed equality subgraph, and any edge that leaves the directed equality subgraph belongs to neither the matching M nor

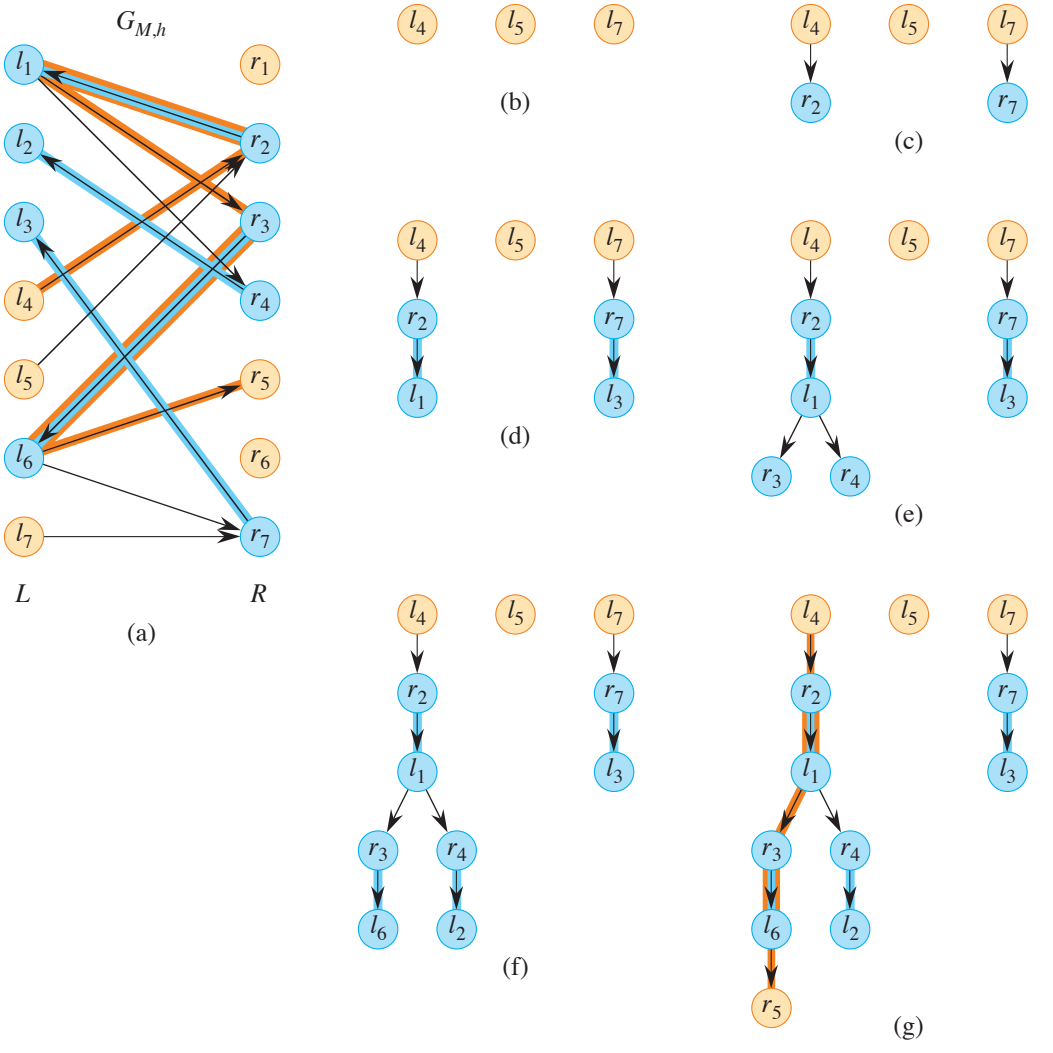


Figure 25.5 Finding an M-augmenting path in $G_{M,h}$ by breadth-first search. **(a)** The directed equality subgraph $G_{M,h}$ from Figure 25.4(c). **(b)–(g)** Successive versions of the breadth-first forest F, shown as the vertices at each distance from the roots—the unmatched vertices in L—are discovered. In parts (b)–(f), the layer of vertices closest to the bottom of the figure are those in the first-in, first-out queue. For example, in part (b), the queue contains the roots $\langle l_4, l_5, l_7 \rangle$, and in part (e), the queue contains $\langle r_3, r_4 \rangle$, at distance 3 from the roots. In part (g), the unmatched vertex r_5 is discovered, so the breadth-first search terminates. The path $\langle (l_4, r_2), (r_2, l_1), (l_1, r_3), (r_3, l_6), (l_6, r_5) \rangle$, highlighted in orange in parts (a) and (g), is an M-augmenting path. Taking its symmetric difference with the matching M yields a new matching with one more edge than M.

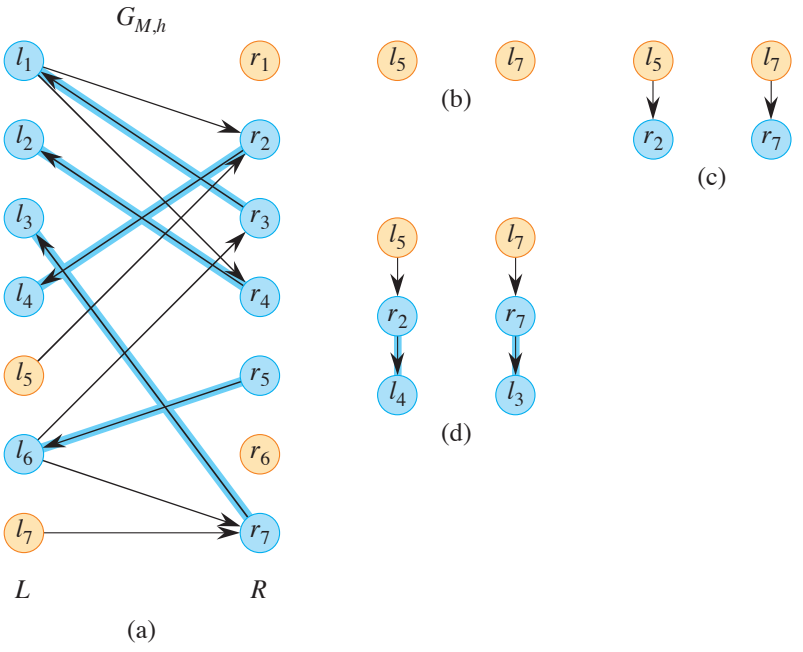


Figure 25.6 **(a)** The new matching M and the new directed equality subgraph $G_{M.h}$ after updating the matching in Figure 25.5(a) with the M-augmenting path in Figure 25.5(g). **(b)–(d)** Successive versions of the breadth-first forest F in a new breadth-first search with roots l_5 and l_7. After the vertices l_4 and l_3 in part (d) have been removed from the queue, the queue becomes empty before the search can discover an unmatched vertex in R.

the breadth-first forest F. Newly discovered vertices in R are enqueued, but their distances are not necessarily 1 greater than the distances of the most recently discovered vertices in L.

To update the feasible vertex labeling, the Hungarian algorithm first computes the value

$$\delta = \min \{l.h + r.h - w(l, r) : l \in F_L \text{ and } r \in R - F_R\} \ , \tag{25.4}$$

where $F_L = L \cap V_F$ and $F_R = R \cap V_F$ denote the vertices in the breadth-first forest F that belong to L and R, respectively. That is, δ is the smallest difference by which an edge incident on a vertex in F_L missed being in the current equality subgraph G_h. The Hungarian algorithm then creates a new feasible vertex labeling, say h', by subtracting δ from $l.h$ for all vertices $l \in F_L$ and adding δ to $r.h$ for all vertices $r \in F_R$:

$$v.h' = \begin{cases} v.h - \delta & \text{if } v \in F_L \ , \\ v.h + \delta & \text{if } v \in F_R \ , \\ v.h & \text{otherwise } (v \in V - V_F) \ . \end{cases} \tag{25.5}$$

The following lemma shows that these changes achieve the three criteria above.

Lemma 25.15
Let h be a feasible vertex labeling for the complete bipartite graph G with equality subgraph G_h, and let M be a matching for G_h and F be a breadth-first forest being constructed for the directed equality subgraph $G_{M,h}$. Then, the labeling h' in equation (25.5) is a feasible vertex labeling for G with the following properties:

1. If (u, v) is an edge in the breadth-first forest F for $G_{M,h}$, then $(u, v) \in E_{M,h'}$.
2. If (l, r) belongs to the matching M for G_h, then $(r, l) \in E_{M,h'}$.
3. There exist vertices $l \in F_L$ and $r \in R - F_R$ such that $(l, r) \notin E_{M,h}$ but $(l, r) \in E_{M,h'}$.

Proof We first show that h' is a feasible vertex labeling for G. Because h is a feasible vertex labeling, we have $l.h + r.h \geq w(l, r)$ for all $l \in L$ and $r \in R$. In order for h' to not be a feasible vertex labeling, we would need $l.h' + r.h' < l.h + r.h$ for some $l \in L$ and $r \in R$. The only way this could occur would be for some $l \in F_L$ and $r \in R - F_R$. In this instance, the amount of the decrease equals δ, so that $l.h' + r.h' = l.h - \delta + r.h$. By equation (25.4), we have that $l.h - \delta + r.h \geq w(l, r)$ for any $l \in F_L$ and $r \in R - F_R$, so that $l.h' + r.h' \geq w(l, r)$. For all other edges, we have $l.h' + r.h' \geq l.h + r.h \geq w(l, r)$. Thus, h' is a feasible vertex labeling.

Now we show that each of the three desired properties holds:

1. If $l \in F_L$ and $r \in F_R$, then we have $l.h' + r.h' = l.h + r.h$ because δ is added to the label of l and subtracted from the label of r. Therefore, if an edge belongs to F for the directed graph $G_{M,h}$, it also belongs to $G_{M,h'}$.
2. We claim that at the time the Hungarian algorithm computes the new feasible vertex labeling h', for every edge $(l, r) \in M$, we have $l \in F_L$ if and only if $r \in F_R$. To see why, consider a matched vertex r and let $(l, r) \in M$. First suppose that $r \in F_R$, so that the search discovered r and enqueued it. When r was removed from the queue, l was discovered, so $l \in F_L$. Now suppose that $r \notin F_R$, so r is undiscovered. We will show that $l \notin F_L$. The only edge in $G_{M,h}$ that enters l is (r, l), and since r is undiscovered, the search has not taken this edge; if $l \in F_L$, it is not because of the edge (r, l). The only other way that a vertex in L can be in F_L is if it is a root of the search, but only unmatched vertices in L are roots and l is matched. Thus, $l \notin F_L$, and the claim is proved.

   We already saw that $l \in F_L$ and $r \in F_R$ implies $l.h' + r.h' = l.h + r.h$. For the opposite case, when $l \in L - F_L$ and $R \in R - F_R$, we have that $l.h' = l.h$ and $r.h' = r.h$, so that again $l.h' + r.h' = l.h + r.h$. Thus, if edge (l, r) is in the matching M for the equality graph G_h, then $(r, l) \in E_{M,h'}$.

3. Let (l, r) be an edge not in E_h such that $l \in F_L$, $r \in R - F_R$, and $\delta = l.h + r.h - w(l, r)$. By the definition of δ, there is at least one such edge. Then, we have

$$\begin{aligned} l.h' + r.h' &= l.h - \delta + r.h \\ &= l.h - (l.h + r.h - w(l, r)) + r.h \\ &= w(l, r) , \end{aligned}$$

and thus $(l, r) \in E_{h'}$. Since (l, r) is not in E_h, it is not in the matching M, so that in $E_{M,h'}$ it must be directed from L to R. Thus, $(l, r) \in E_{M,h'}$. ■

It is possible for an edge to belong to $E_{M,h}$ but not to $E_{M,h'}$. By Lemma 25.15, any such edge belongs neither to the matching M nor to the breadth-first forest F at the time that the new feasible vertex labeling h' is computed. (See Exercise 25.3-3.)

Going back to Figure 25.6(d), the queue became empty before an M-augmenting path was found. Figure 25.7 shows the next steps taken by the algorithm. The value of $\delta = 1$ is achieved by the edge (l_5, r_3) because in Figure 25.4(a), $l_5.h + r_3.h - w(l_5, r_3) = 6 + 0 - 5 = 1$. In Figure 25.7(a), the values of $l_3.h$, $l_4.h$, $l_5.h$, and $l_7.h$ have decreased by 1 and the values of $r_2.h$ and $r_7.h$ have increased by 1 because these vertices are in F. As a result, the edges (l_1, r_2) and (l_6, r_7) leave $G_{M,h}$ and the edge (l_5, r_3) enters. Figure 25.7(b) shows the new directed equality subgraph $G_{M,h}$. With edge (l_5, r_3) now in $G_{M,h}$, Figure 25.7(c) shows that this edge is added to the breadth-first forest F, and r_3 is added to the queue. Parts (c)–(f) show the breadth-first forest continuing to be built until in part (f), the queue once again becomes empty after vertex l_2, which has no edges leaving, is removed. Again, the algorithm must update the feasible vertex labeling and the directed equality subgraph. Now the value of $\delta = 1$ is achieved by three edges: (l_1, r_6), (l_5, r_6), and (l_7, r_6).

As Figure 25.8 shows in parts (a) and (b), these edges enter $G_{M,h}$, and edge (l_6, r_3) leaves. Part (c) shows that edge (l_1, r_6) is added to the breadth-first forest. (Either of edges (l_5, r_6) or (l_7, r_6) could have been added instead.) Because r_6 is unmatched, the search has found the M-augmenting path $\langle (l_5, r_3), (r_3, l_1), (l_1, r_6) \rangle$, highlighted in orange.

Figure 25.9(a) shows $G_{M,h}$ after the matching M has been updated by taking its symmetric difference with the M-augmenting path. The Hungarian algorithm starts its last breadth-first search, with vertex l_7 as the only root. The search proceeds as shown in parts (b)–(h) of the figure, until the queue becomes empty after removing l_4. This time, we find that $\delta = 2$, achieved by the five edges (l_2, r_5), (l_3, r_1), (l_4, r_5), (l_5, r_1), and (l_5, r_5), each of which enters $G_{M,h}$. Figure 25.10(a) shows the results of decreasing the feasible vertex label of each vertex in F_L by 2 and increasing the feasible vertex label of each vertex in F_R

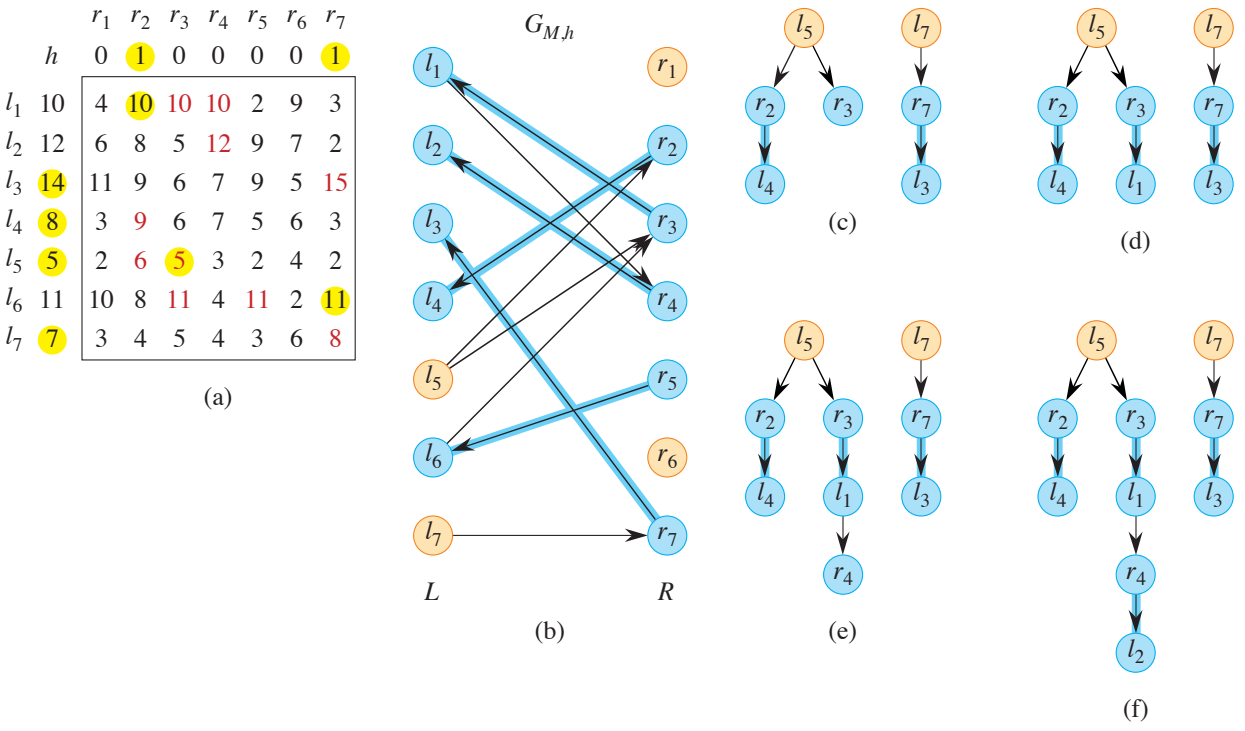


Figure 25.7 Updating the feasible vertex labeling and the directed equality subgraph $G_{M,h}$ when the queue becomes empty before finding an M-augmenting path. **(a)** With $\delta = 1$, the values of $l_3.h$, $l_4.h$, $l_5.h$, and $l_7.h$ decreased by 1 and $r_2.h$ and $r_7.h$ increased by 1. Edges (l_1, r_2) and (l_6, r_7) leave $G_{M,h}$, and edge (l_5, r_3) enters. These changes are highlighted in yellow. **(b)** The resulting directed equality subgraph $G_{M,h}$. **(c)–(f)** With edge (l_5, r_3) added to the breadth-first forest and r_3 added to the queue, the breadth-first search continues until the queue once again becomes empty in part (f).

by 2, and Figure 25.10(b) shows the resulting directed equality subgraph $G_{M,h}$. Part (c) shows that edge (l_3, r_1) is added to the breadth-first forest. Since r_1 is an unmatched vertex, the search terminates, having found the M-augmenting path $\langle (l_7, r_7), (r_7, l_3), (l_3, r_1) \rangle$, highlighted in orange. If r_1 had been matched, vertex r_5 would also have been added to the breadth-first forest, with any of l_2, l_4, or l_5 as its parent.

After updating the matching M, the algorithm arrives at the perfect matching shown for the equality subgraph G_h in Figure 25.11. By Theorem 25.14, the edges in M form an optimal solution to the original assignment problem given in the matrix. Here, the weights of edges (l_1, r_6), (l_2, r_4), (l_3, r_1), (l_4, r_2), (l_5, r_3), (l_6, r_5), and (l_7, r_7) sum to 65, which is the maximum weight of any matching.

The weight of the maximum-weight matching equals the sum of all the feasible vertex labels. These problems—maximizing the weight of a matching and mini-

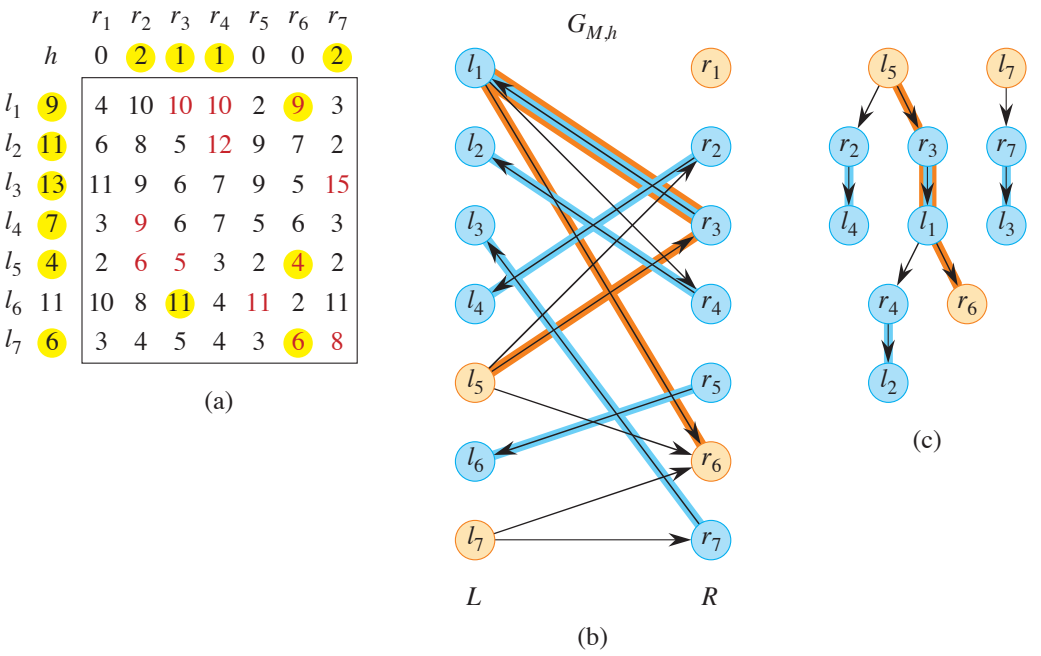


Figure 25.8 Another update to the feasible vertex labeling and directed equality subgraph $G_{M,h}$ because the queue became empty before finding an M-augmenting path. **(a)** With $\delta = 1$, the values of $l_1.h$, $l_2.h$, $l_3.h$, $l_4.h$, $l_5.h$, and $l_7.h$ decrease by 1, and $r_2.h$, $r_3.h$, $r_4.h$, and $r_7.h$ increase by 1. Edge (l_6, r_3) leaves $G_{M,h}$, and edges (l_1, r_6), (l_5, r_6) and (l_7, r_6) enter. **(b)** The resulting directed equality subgraph $G_{M,h}$. **(c)** With edge (l_1, r_6) added to the breadth-first forest and r_6 unmatched, the search terminates, having found the M-augmenting path $\langle (l_5, r_3), (r_3, l_1), (l_1, r_6) \rangle$, highlighted in orange in parts (b) and (c).

mizing the sum of the feasible vertex labels—are "duals" of each other, in a similar vein to how the value of a maximum flow equals the capacity of a minimum cut. Section 29.3 explores duality in more depth.

The Hungarian algorithm

The procedure HUNGARIAN on page 737 and its subroutine FIND-AUGMENTING-PATH on page 738 follow the steps we have just seen. The third property in Lemma 25.15 ensures that in line 23 of FIND-AUGMENTING-PATH the queue Q is nonempty. The pseudocode uses the attribute π to indicate predecessor vertices in the breadth-first forest. Instead of coloring vertices, as in the BFS procedure on page 556, the search puts the discovered vertices into the sets F_L and F_R. Because the Hungarian algorithm does not need breadth-first distances, the pseudocode omits the d attribute computed by the BFS procedure.

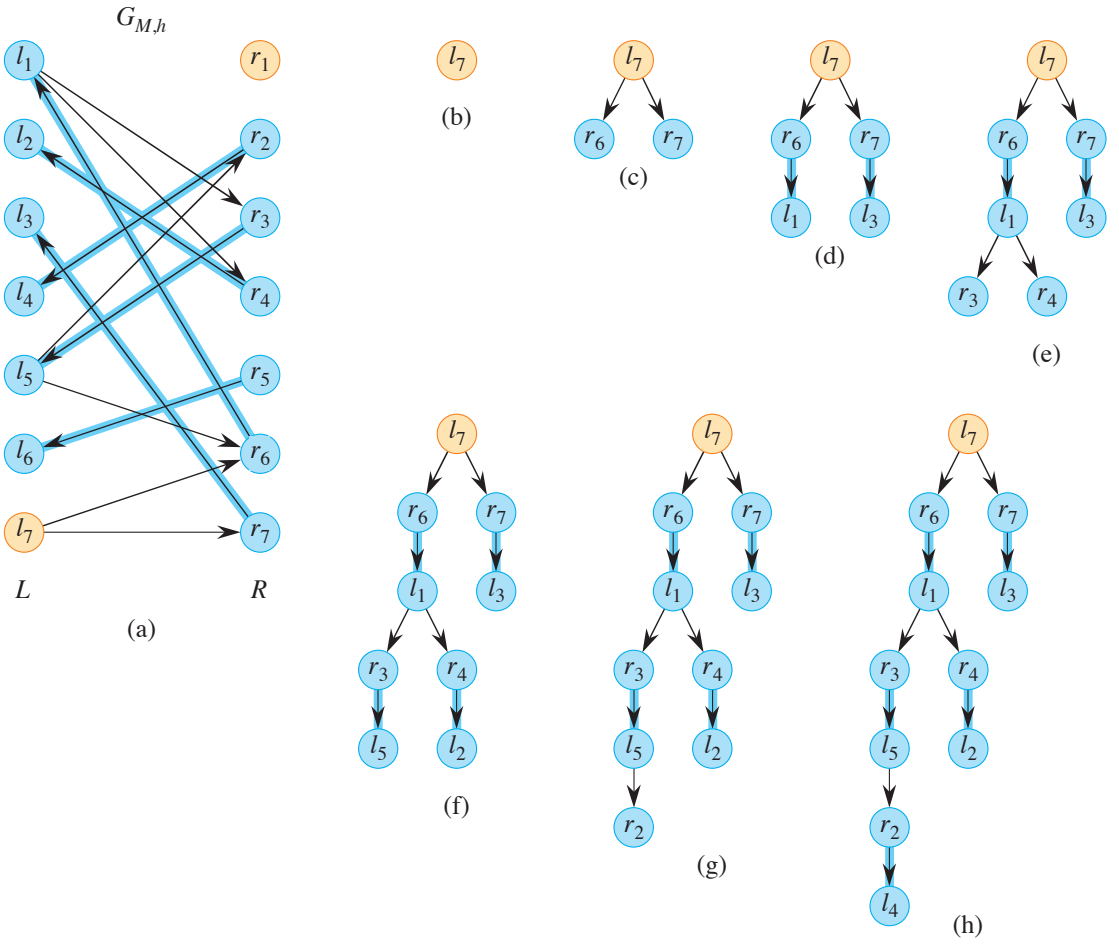


Figure 25.9 **(a)** The new matching M and the new directed equality subgraph $G_{M,h}$ after updating the matching in Figure 25.8 with the M-augmenting path in Figure 25.8 parts (b) and (c). **(b)–(h)** Successive versions of the breadth-first forest F in a new breadth-first search with root l_7. After the vertex l_4 in part (h) has been removed from the queue, the queue becomes empty before the search discovers an unmatched vertex in R.

Now, let's see why the Hungarian algorithm runs in $O(n^4)$ time, where $|V| = n/2$ and $|E| = n^2$ in the original graph G. (Below we outline how to reduce the running time to $O(n^3)$.) You can go through the pseudocode of HUNGARIAN to verify that lines 1–6 and 11 take $O(n^2)$ time. The **while** loop of lines 7–10 iterates at most n times, since each iteration increases the size of the matching M by 1. Each test in line 7 can take constant time by just checking whether $|M| < n$, each update of M in line 9 takes $O(n)$ time, and the updates in line 10 take $O(n^2)$ time.

To achieve the $O(n^4)$ time bound, it remains to show that each call of FIND-AUGMENTING-PATH runs in $O(n^3)$ time. Let's call each execution of lines 10–22

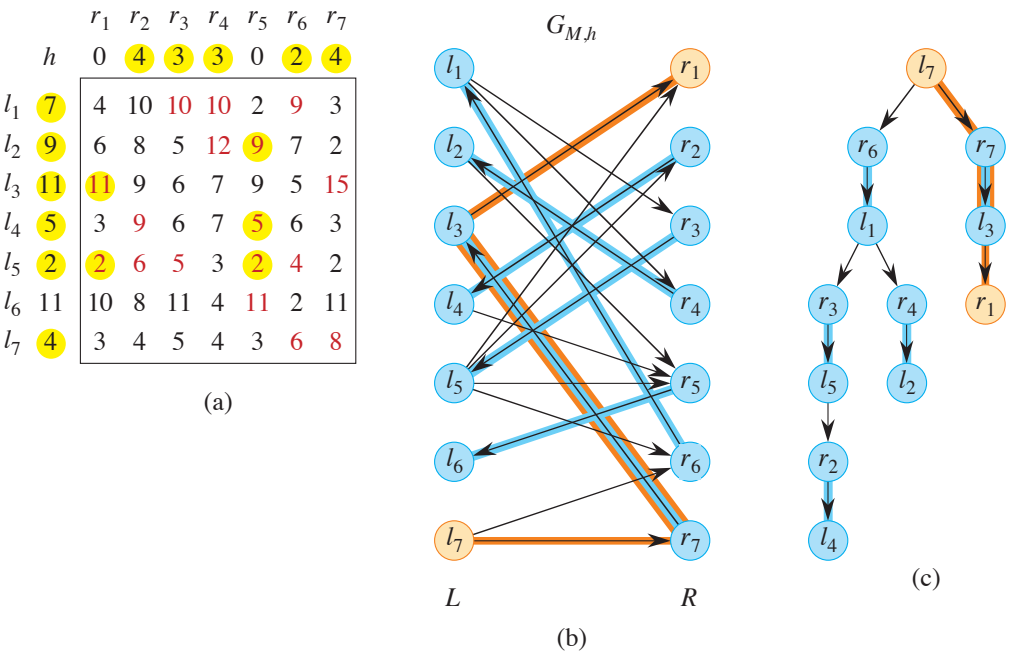


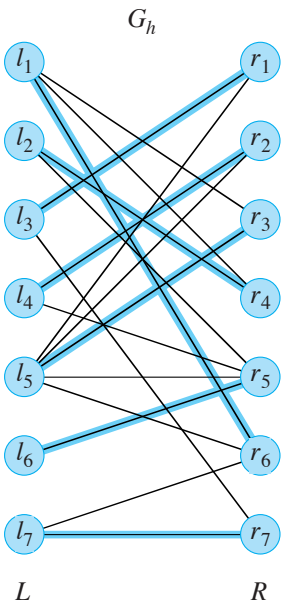
Figure 25.10 Updating the feasible vertex labeling and directed equality subgraph $G_{M,h}$. **(a)** Here, $\delta = 2$, so the values of $l_1.h$, $l_2.h$, $l_3.h$, $l_4.h$, $l_5.h$, and $l_7.h$ decreased by 2, and the values of $r_2.h$, $r_3.h$, $r_4.h$, $r_6.h$, and $r_7.h$ increased by 2. Edges (l_2, r_5), (l_3, r_1), (l_4, r_5), (l_5, r_1), and (l_5, r_5) enter $G_{M,h}$. **(b)** The resulting directed graph $G_{M,h}$. **(c)** With edge (l_3, r_1) added to the breadth-first forest and r_1 unmatched, the search terminates, having found the M-augmenting path $\langle (l_7, r_7), (r_7, l_3), (l_3, r_1) \rangle$, highlighted in orange in parts (b) and (c).

a ***growth step***. Ignoring the growth steps, you can verify that FIND-AUGMENTING-PATH is a breadth-first search. With the sets F_L and F_R represented appropriately, the breadth-first search takes $O(V + E) = O(n^2)$ time. Within a call of FIND-AUGMENTING-PATH, at most n growth steps can occur, since each growth step is guaranteed to discover at least one vertex in R. Since there are at most n^2 edges in $G_{M,h}$, the **for** loop of lines 16–22 iterates at most n^2 times per call of FIND-AUGMENTING-PATH. The bottleneck is lines 10 and 15, which take $O(n^2)$ time, so that FIND-AUGMENTING-PATH takes $O(n^3)$ time.

Exercise 25.3-5 asks you to show that reconstructing the directed equality subgraph $G_{M,h}$ in line 15 is actually unnecessary, so that its cost can be eliminated. Reducing the cost of computing δ in line 10 to $O(n)$ takes a little more effort and is the subject of Problem 25-2. With these changes, each call of FIND-AUGMENTING-PATH takes $O(n^2)$ time, so that the Hungarian algorithm runs in $O(n^3)$ time.

		r_1	r_2	r_3	r_4	r_5	r_6	r_7
	h	0	4	3	3	0	2	4
l_1	7	4	10	10	10	2	**9**	3
l_2	9	6	8	5	**12**	9	7	2
l_3	11	**11**	9	6	7	9	5	15
l_4	5	3	**9**	6	7	5	6	3
l_5	2	2	6	**5**	3	2	4	2
l_6	11	10	8	11	4	**11**	2	11
l_7	4	3	4	5	4	3	6	**8**

(a)



(b)

Figure 25.11 The final matching, shown for the equality subgraph G_h with blue edges and blue entries in the matrix. The weights of the edges in the matching sum to 65, which is the maximum for any matching in the original complete bipartite graph G, as well as the sum of all the final feasible vertex labels.

```
HUNGARIAN(G)
 1  for each vertex l ∈ L
 2      l.h = max {w(l, r) : r ∈ R}     // from equation (25.1)
 3  for each vertex r ∈ R
 4      r.h = 0                         // from equation (25.2)
 5  let M be any matching in G_h (such as the matching returned by
        GREEDY-BIPARTITE-MATCHING)
 6  from G, M, and h, form the equality subgraph G_h
        and the directed equality subgraph G_{M,h}
 7  while M is not a perfect matching in G_h
 8      P = FIND-AUGMENTING-PATH(G_{M,h})
 9      M = M ⊕ P
10      update the equality subgraph G_h
            and the directed equality subgraph G_{M,h}
11  return M
```

```
FIND-AUGMENTING-PATH(G_{M,h})
 1  Q = ∅
 2  F_L = ∅
 3  F_R = ∅
 4  for each unmatched vertex l ∈ L
 5      l.π = NIL
 6      ENQUEUE(Q, l)
 7      F_L = F_L ∪ {l}          // forest F starts with unmatched vertices in L
 8  repeat
 9      if Q is empty            // ran out of vertices to search from?
10          δ = min {l.h + r.h − w(l, r) : l ∈ F_L and r ∈ R − F_R}
11          for each vertex l ∈ F_L
12              l.h = l.h − δ    // relabel according to equation (25.5)
13          for each vertex r ∈ F_R
14              r.h = r.h + δ    // relabel according to equation (25.5)
15          from G, M, and h, form a new directed equality graph G_{M,h}
16          for each new edge (l, r) in G_{M,h}    // continue search with new edges
17              if r ∉ F_R
18                  r.π = l                        // discover r, add it to F
19                  if r is unmatched
20                      an M-augmenting path has been found
                            (exit the repeat loop)
21                  else ENQUEUE(Q, r)             // can search from r later
22                      F_R = F_R ∪ {r}
23      u = DEQUEUE(Q)                             // search from u
24      for each neighbor v of u in G_{M,h}
25          if v ∈ L
26              v.π = u
27              F_L = F_L ∪ {v}                    // discover v, add it to F
28              ENQUEUE(Q, v)                      // can search from v later
29          elseif v ∉ F_R                         // v ∈ R, do same as lines 18–22
30              v.π = u
31              if v is unmatched
32                  an M-augmenting path has been found
                        (exit the repeat loop)
33              else ENQUEUE(Q, v)
34                  F_R = F_R ∪ {v}
35  until an M-augmenting path has been found
36  using the predecessor attributes π, construct an M-augmenting path P
        by tracing back from the unmatched vertex in R
37  return P
```

Exercises

25.3-1

The FIND-AUGMENTING-PATH procedure checks in two places (lines 19 and 31) whether a vertex it discovers in R is unmatched. Show how to rewrite the pseudocode so that it checks for an unmatched vertex in R in only one place. What is the downside of doing so?

25.3-2

Show that for any bipartite graph, the GREEDY-BIPARTITE-MATCHING procedure on page 726 returns a matching at least half the size of a maximum matching.

25.3-3

Show that if an edge (l, r) belongs to the directed equality subgraph $G_{M,h}$ but is not a member of $G_{M,h'}$, where h' is given by equation (25.5), then $l \in L - F_L$ and $r \in F_R$ at the time that h' is computed.

25.3-4

At line 29 in the FIND-AUGMENTING-PATH procedure, it has already been established that $v \in R$. This line checks to see whether v is already discovered by testing whether $v \in F_R$. Why doesn't the procedure need to check whether v is already discovered for the case when $v \in L$, in lines 26–28?

25.3-5

Professor Hrabosky asserts that the directed equality subgraph $G_{M,h}$ must be constructed and maintained by the Hungarian algorithm, so that line 6 of HUNGARIAN and line 15 of FIND-AUGMENTING-PATH are required. Argue that the professor is incorrect by showing how to determine whether an edge belongs to $E_{M,h}$ without explicitly constructing $G_{M,h}$.

25.3-6

How can you modify the Hungarian algorithm to find a matching of vertices in L to vertices in R that minimizes, rather than maximizes, the sum of the edge weights in the matching?

25.3-7

How can an assignment problem with $|L| \neq |R|$ be modified so that the Hungarian algorithm solves it?

Problems

25-1 Perfect matchings in a regular bipartite graph

a. Problem 20-3 asked about Euler tours in directed graphs. Prove that a connected, *undirected* graph $G = (V, E)$ has an Euler tour—a cycle traversing each edge exactly once, though it may visit a vertex multiple times—if and only if the degree of every vertex in V is even.

b. Assuming that G is connected, undirected, and every vertex in V has even degree, give an $O(E)$-time algorithm to find an Euler tour of G, as in Problem 20-3(b).

c. Exercise 25.1-6 states that if $G = (V, E)$ is a d-regular bipartite graph, then it contains d disjoint perfect matchings. Suppose that d is an exact power of 2. Give an algorithm to find all d disjoint perfect matchings in a d-regular bipartite graph in $\Theta(E \lg d)$ time.

25-2 Reducing the running time of the Hungarian algorithm to $O(n^3)$

In this problem, you will show how to reduce the running time of the Hungarian algorithm from $O(n^4)$ to $O(n^3)$ by showing how to reduce the running time of the FIND-AUGMENTING-PATH procedure from $O(n^3)$ to $O(n^2)$. Exercise 25.3-5 demonstrates that line 6 of HUNGARIAN and line 15 of FIND-AUGMENTING-PATH are unnecessary. Now you will show how to reduce the running time of each execution of line 10 in FIND-AUGMENTING-PATH to $O(n)$.

For each vertex $r \in R - F_R$, define a new attribute $r.\sigma$, where

$$r.\sigma = \min \{l.h + r.h - w(l, r) : l \in F_L\} .$$

That is, $r.\sigma$ indicates how close r is to being adjacent to some vertex $l \in F_L$ in the directed equality subgraph $G_{m,h}$. Initially, before placing any vertices into F_L, set $r.\sigma$ to ∞ for all $r \in R$.

a. Show how to compute δ in line 10 in $O(n)$ time, based on the σ attribute.

b. Show how to update all the σ attributes in $O(n)$ time after δ has been computed.

c. Show that updating all the σ attributes when F_L changes takes $O(n^2)$ time per call of FIND-AUGMENTING-PATH.

d. Conclude that the HUNGARIAN procedure can be implemented to run in $O(n^3)$ time.

25-3 Other matching problems

The Hungarian algorithm finds a maximum-weight perfect matching in a complete bipartite graph. It is possible to use the Hungarian algorithm to solve problems in other graphs by modifying the input graph, running the Hungarian algorithm, and then possibly modifying the output. Show how to solve the following matching problems in this manner.

a. Give an algorithm to find a maximum-weight matching in a weighted bipartite graph that is not necessarily complete and with all edge weights positive.

b. Redo part (a), but with edge weights allowed to also be 0 or negative.

c. A ***cycle cover*** in a directed graph, not necessarily bipartite, is a set of edge-disjoint directed cycles such that each vertex lies on at most one cycle. Given nonnegative edge weights $w(u, v)$, let C be the set of edges in a cycle cover, and define $w(C) = \sum_{(u,v)\in C} w(u, v)$ to be the weight of the cycle cover. Give an algorithm to find a maximum-weight cycle cover.

25-4 Fractional matchings

It is possible to define a ***fractional matching***. Given a graph $G = (V, E)$, we define a fractional matching x as a function $x : E \to [0, 1]$ (real numbers between 0 and 1, inclusive) such that for every vertex $u \in V$, we have $\sum_{(u,v)\in E} x(u, v) \le 1$. The value of a fractional matching is $\sum_{(u,v)\in E} x(u, v)$. The definition of a fractional matching is identical to that of a matching, except that a matching has the additional constraint that $x(u, v) \in \{0, 1\}$ for all edges $(u, v) \in E$. Given a graph, we let M^* denote a maximum matching and x^* denote a fractional matching with maximum value.

a. Argue that, for any bipartite graph, we must have $\sum_{(u,v)\in E} x^*(u, v) \ge |M^*|$.

b. Prove that, for any bipartite graph, we must have $\sum_{(u,v)\in E} x^*(e) \le |M^*|$. (*Hint:* Give an algorithm that converts a fractional matching with an integer value to a matching.) Conclude that the maximum value of a fractional matching in a bipartite graph is the same as the size of the maximum cardinality matching.

c. We can define a fractional matching in a weighted graph in the same manner: the value of the matching is now $\sum_{(u,v)\in E} w(u, v)x(u, v)$. Extend the results of the previous parts to show that in a weighted bipartite graph, the maximum value of a weighted fractional matching is equal to the value of a maximum weighted matching.

d. In a general graph, the analogous results do not necessarily hold. Give an example of a small graph that is not bipartite for which the fractional matching with maximum value is not a maximum matching.

25-5 Computing vertex labels

You are given a complete bipartite graph $G = (V, E)$ with edge weights $w(l, r)$ for all $(l, r) \in E$. You are also given a maximum-weight perfect matching M^* for G. You wish to compute a feasible vertex labeling h such that M^* is a perfect matching in the equality subgraph G_h. That is, you want to compute a labeling h of vertices such that

$$l.h + r.h \geq w(l, r) \quad \text{for all } l \in L \text{ and } r \in R\ , \tag{25.6}$$

$$l.h + r.h = w(l, r) \quad \text{for all } (l, r) \in M^*\ . \tag{25.7}$$

(Requirement (25.6) holds for all edges, and the stronger requirement (25.7) holds for all edges in M^*.) Give an algorithm to compute the feasible vertex labeling h, and prove that it is correct. (*Hint:* Use the similarity between conditions (25.6) and (25.7) and some of the properties of shortest paths proved in Chapter 22, in particular the triangle inequality (Lemma 22.10) and the convergence property (Lemma 22.14.))

Chapter notes

Matching algorithms have a long history and have been central to many breakthroughs in algorithm design and analysis. The book by Lovász and Plummer [306] is an excellent reference on matching problems, and the chapter on matching in the book by Ahuja, Magnanti and Orlin [10] also has extensive references.

The Hopcroft-Karp algorithm is by Hopcroft and Karp [224]. Madry [308] gave an $\widetilde{O}(E^{10/7})$-time algorithm, which is asymptotically faster than Hopcroft-Karp for sparse graphs.

Corollary 25.4 is due to Berge [53], and it also holds in graphs that are not bipartite. Matching in general graphs requires more complicated algorithms. The first polynomial-time algorithm, running in $O(V^4)$ time, is due to Edmonds [130] (in a paper that also introduced the notion of a polynomial-time algorithm). Like the bipartite case, this algorithm also uses augmenting paths, although the algorithm for finding augmenting paths in general graphs is more involved than the one for bipartite graphs. Subsequently, several $O(\sqrt{V}E)$-time algorithms appeared, including ones by Gabow and Tarjan [168] as part of an algorithm for weighted matching and a simpler one by Gabow [164].

The Hungarian algorithm is described in the book by Bondy and Murty [67] and is based on work by Kuhn [273] and Munkres [337]. Kuhn adopted the name "Hungarian algorithm" because the algorithm derived from work by the Hungarian mathematicians D. Kőnig and J. Egervéry. The algorithm is an early example of a primal-dual algorithm. A faster algorithm that runs in $O(\sqrt{V}E \log(VW))$ time, where the edge weights are integers from 0 to W, was given by Gabow and Tarjan [167], and an algorithm with the same time bound for maximum-weight matching in general graphs was given by Duan, Pettie, and Su [127].

The stable-marriage problem was first defined and analyzed by Gale and Shapley [169]. The stable-marriage problem has numerous variants. The books by Gusfield and Irving [203], Knuth [266], and Manlove [313] serve as excellent sources for cataloging and solving them.