CHAPTER TWENTY-ONE

Shortest Paths

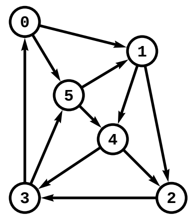
EVERY PATH IN a weighted digraph has an associated *path weight,* the value of which is the sum of the weights of that path's edges. This essential measure allows us to formulate such problems as "find the lowest-weight path between two given vertices." These *shortest-paths problems* are the topic of this chapter. Not only are shortest-paths problems intuitive for many direct applications, but they also take us into a powerful and general realm where we seek efficient algorithms to solve general problems that can encompass a broad variety of specific applications.

Several of the algorithms that we consider in this chapter relate directly to various algorithms that we examined in Chapters 17 through 20. Our basic graph-search paradigm applies immediately, and several of the specific mechanisms that we used in Chapters 17 and 19 to address connectivity in graphs and digraphs provide the basis for us to solve shortest-paths problems.

For economy, we refer to weighted digraphs as *networks*. Figure 21.1 shows a sample network, with standard representations. We have already developed an ADT interface with adjacency-matrix and adjacency-lists class implementations for networks in Section 20.1—we just pass `true` as a second parameter when we call the constructor so that the class keeps one representation of each edge, precisely as we did when deriving digraph representations in Chapter 19 from the undirected graph representations in Chapter 17 (see Programs 20.1 through 20.4).

As discussed at length in Chapter 20, we use references to abstract edges for weighted digraphs to broaden the applicability of our

```
0-1 .41
1-2 .51
2-3 .50
4-3 .36
3-5 .38
3-0 .45
0-5 .29
5-4 .21
1-4 .32
4-2 .32
5-1 .29
```



	0	1	2	3	4	5
0	0	.41				.29
1		0	.51		.32	
2			0	.50		
3	.45			0		.38
4			.32	.36	0	
5		.29			.21	0

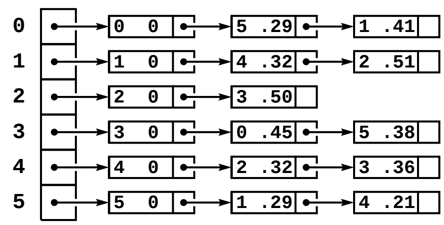


Figure 21.1
Sample network and representations

This network (weighted digraph) is shown in four representations: list of edges, drawing, adjacency matrix, and adjacency lists (left to right). *As we did for MST algorithms, we show the weights in matrix entries and in list nodes, but use edge references in our programs. While we often use edge weights that are proportional to their lengths in the drawing (as we did for MST algorithms), we do not insist on this rule because most shortest-paths algorithms handle arbitrary nonnegative weights (negative weights do present special challenges). The adjacency matrix is not symmetric, and the adjacency lists contain one node for each edge (as in unweighted digraphs). Nonexistent edges are represented by null references in the matrix (blank in the figure) and are not present at all in the lists. Self-loops of length* `0` *are present because they simplify our implementations of shortest-paths algorithms. They are omitted from the list of edges at left for economy and to indicate the typical scenario where we add them by convention when we create an adjacency-matrix or adjacency-lists representation.*

implementations. This approach has certain implications that are different for digraphs than the ones that we considered for undirected graphs in Section 20.1 and are worth noting. First, since there is only one representation of each edge, we do not need to use the `from` method in the edge class (see Program 20.1) when using an iterator: In a digraph, `e.from(v)` is `true` for every edge reference `e` returned by an iterator for `v`. Second, as we saw in Chapter 19, it is often useful when processing a digraph to be able to work with its reverse graph, but we need a different approach than that taken by Program 19.1, because that implementation creates edges to create the reverse, and we assume that a graph ADT whose clients provide references to edges should not create edges on its own (see Exercise 21.3).

In applications or systems for which we need all types of graphs, it is a textbook exercise in software engineering to define a network ADT from which ADTs for the unweighted undirected graphs of Chapters 17 and 18, the unweighted digraphs of Chapter 19, or the weighted undirected graphs of Chapter 20 can be derived (see Exercise 21.10).

When we work with networks, it is generally convenient to keep self-loops in all the representations. This convention allows algorithms the flexibility to use a sentinel maximum-value weight to indicate that a vertex cannot be reached from itself. In our examples, we use self-loops of weight `0`, although positive-weight self-loops certainly make sense in many applications. Many applications also call for parallel edges, perhaps with differing weights. As we mentioned in Section 20.1, various options for ignoring or combining such edges are appropriate in various different applications. In this chapter, for simplicity, none of our examples use parallel edges, and we do not allow parallel edges in the adjacency-matrix representation; we also do not check for parallel edges or remove them in adjacency lists.

```
0-1 .99
1-2 .51
2-3 .50
4-3 .36
3-5 .38
0-3 .45
0-5 .83
5-4 .21
1-4 .10
4-2 .41
5-1 .10
```

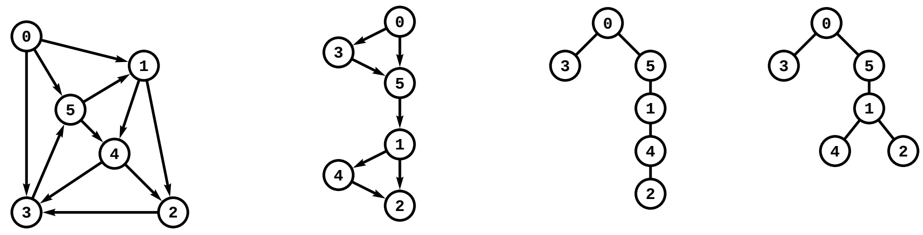


Figure 21.2
Shortest-path trees

A shortest-path tree (SPT) defines shortest paths from the root to other vertices (see Definition 21.2). *In general, different paths may have the same length, so there may be multiple SPTs defining the shortest paths from a given vertex. In the example network shown at left, all shortest paths from* 0 *are subgraphs of the DAG shown to the right of the network. A tree rooted at* 0 *spans this DAG if and only if it is an SPT for* 0. *The two trees at right are such trees.*

All the connectivity properties of digraphs that we considered in Chapter 19 are relevant in networks. In that chapter, we wished to know whether it is *possible* to get from one vertex to another; in this chapter, we take weights into consideration—we wish to find the *best* way to get from one vertex to another.

Definition 21.1 *A* **shortest path** *between two vertices s and t in a network is a directed simple path from s to t with the property that no other such path has a lower weight.*

This definition is succinct, but its brevity masks points worth examining. First, if t is not reachable from s, there is no path at all, and therefore there is no shortest path. For convenience, the algorithms that we consider often treat this case as equivalent to one in which there exists an infinite-weight path from s to t. Second, as we did for MST algorithms, we use networks where edge weights are proportional to edge lengths in examples, but the definition has no such requirement and our algorithms (other than the one in Section 21.5) do not make this assumption. Indeed, shortest-paths algorithms are at their best when they discover counterintuitive shortcuts, such as a path between two vertices that passes through several other vertices but has total weight smaller than that of a direct edge connecting those vertices. Third, there may be multiple paths of the same weight from one vertex to another; we typically are content to find one of them. Figure 21.2 shows an example with general weights that illustrates these points.

The restriction in the definition to simple paths is unnecessary in networks that contain edges that have nonnegative weight, because any cycle in a path in such a network can be removed to give a path that is no longer (and is shorter unless the cycle comprises zero-weight edges). But when we consider networks with edges that could have negative weight, the need for the restriction to simple paths is readily apparent: Otherwise, the concept of a shortest path is meaningless if there is a cycle in the network that has negative weight. For example, suppose that the edge 3-5 in the network in Figure 21.1 were to have weight -.38, and edge 5-1 were to have weight -.31. Then, the weight of the cycle 1-4-3-5-1 would be .32 + .36 - .38 - .31 = -.01, and we could spin around that cycle to generate arbitrarily short paths. Note carefully that, as is true in this example, it is *not* necessary for all the edges on a negative-weight cycle to be of negative weight; what counts is the *sum* of the edge weights. For brevity, we use the term *negative cycle* to refer to directed cycles whose total weight is negative.

In the definition, suppose that some vertex on a path from s to t is also on a negative cycle. In this case, the existence of a (nonsimple) shortest path from s to t would be a contradiction, because we could use the cycle to construct a path that had a weight lower than any given value. To avoid this contradiction, we restrict to simple paths in the definition so that the concept of a shortest path is well defined in any network. However, we do not consider negative cycles in networks until Section 21.7, because, as we see there, they present a truly fundamental barrier to the solution of shortest-paths problems.

To find shortest paths in a weighted undirected graph, we build a network with the same vertices and with two edges (one in each direction) corresponding to each edge in the graph. There is a one-to-one correspondence between simple paths in the network and simple paths in the graph, and the costs of the paths are the same; so shortest-paths problems are equivalent. Indeed, we build precisely such a network when we build the standard adjacency-lists or adjacency-matrix representation of a weighted undirected graph (see, for example, Figure 20.3). This construction is not helpful if weights can be negative, because it gives negative cycles in the network, and we do not know how to solve shortest-paths problems in networks that have negative cycles (see Section 21.7). Otherwise, the algorithms for networks that we consider in this chapter also work for weighted undirected graphs.

In certain applications, it is convenient to have weights on vertices instead of, or in addition to, weights on edges; and we might also consider more complicated problems where both the number of edges on the path and the overall weight of the path play a role. We can handle such problems by recasting them in terms of edge-weighted

0		.41	0-1	.82	0-5-4-2	.86	0-5-4-3	.50	0-5-4	.29	0-5
1.13	1-4-3-0		1	.51	1-2	.68	1-4-3	.32	1-4	1.06	1-4-3-5
.95	2-3-0	1.17	2-3-5-1		2	.50	2-3	1.09	2-3-5-4	.88	2-3-5
.45	3-0	.67	3-5-1	.91	3-5-4-2	0	3	.59	3-5-4	.38	3-5
.81	4-3-0	1.03	4-3-5-1	.32	4-2	.36	4-3	0	4	.74	4-3-5
1.02	5-4-3-0	.29	5-1	.53	5-4-2	.57	5-4-3	.21	5-4	0	5

Figure 21.3
All shortest paths

This table gives all the shortest paths in the network of Figure 21.1 and their lengths. This network is strongly connected, so there exist paths connecting each pair of vertices.

The goal of a source-sink shortest-path algorithm is to compute one of the entries in this table; the goal of a single-source shortest-paths algorithm is to compute one of the rows in this table; and the goal of an all-pairs shortest-paths algorithm is to compute the whole table. Generally, we use more compact representations, which contain essentially the same information and allow clients to trace any path in time proportional to its number of edges (see Figure 21.8).

networks (see, for example, Exercise 21.4) or by slightly extending the basic algorithms (see, for example, Exercise 21.52).

Because the distinction is clear from the context, we do not introduce special terminology to distinguish shortest paths in weighted graphs from shortest paths in graphs that have no weights (where a path's weight is simply its number of edges—see Section 17.7). The usual nomenclature refers to (edge-weighted) networks, as used in this chapter, since the special cases presented by undirected or unweighted graphs are handled easily by algorithms that process networks.

We are interested in the same basic problems that we defined for undirected and unweighted graphs in Section 18.7. We restate them here, noting that Definition 21.1 implicitly generalizes them to take weights into account in networks.

Source–sink shortest path Given a start vertex s and a finish vertex t, find a shortest path in the graph from s to t. We refer to the start vertex as the *source* and to the finish vertex as the *sink*, except in contexts where this usage conflicts with the definition of sources (vertices with no incoming edges) and sinks (vertices with no outgoing edges) in digraphs.

Single-source shortest paths Given a start vertex s, find shortest paths from s to each other vertex in the graph.

All-pairs shortest paths Find shortest paths connecting each pair of vertices in the graph. For brevity, we sometimes use the term *all shortest paths* to refer to this set of V^2 paths.

If there are multiple shortest paths connecting any given pair of vertices, we are content to find any one of them. Since paths have varying number of edges, our implementations provide methods that allow clients to trace paths in time proportional to the paths' lengths. Any shortest path also implicitly gives us the shortest-path length, but our implementations explicitly provide lengths. In summary, to be precise, when we say "find a shortest path" in the problem statements just given, we mean "compute the shortest-path length and a way to trace a specific path in time proportional to that path's length."

Figure 21.3 illustrates shortest paths for the example network in Figure 21.1. In networks with V vertices, we need to specify V paths to solve the single-source problem and to specify V^2 paths to solve the all-pairs problem. In our implementations, we use a representation more compact than these lists of paths; we first noted it in Section 18.7, and we consider it in detail in Section 21.1.

In Java implementations, we build our algorithmic solutions to these problems into ADT implementations that allow us to build efficient client programs that can solve a variety of practical graph-processing problems. For example, as we see in Section 21.3, we implement solutions to the all-pairs shortest-paths classes as constructors within classes that support constant-time shortest-path queries. We also build classes to solve single-source problems so that clients who need to compute shortest paths from a specific vertex (or a small set of them) can avoid the expense of computing shortest paths for other vertices. Careful consideration of such issues and proper use of the algorithms that we examine can mean the difference between an efficient solution to a practical problem and a solution that is so costly that no client could afford to use it.

Shortest-paths problems arise in various guises in numerous applications. Many of the applications appeal immediately to geometric intuition, but many others involve arbitrary cost structures. As we did with minimum spanning trees (MSTs) in Chapter 20, we sometimes take advantage of geometric intuition to help develop an understanding of algorithms that solve the problems but stay cognizant that our algorithms operate properly in more general settings. In Section 21.5, we do consider specialized algorithms for Euclidean networks. More important, in Sections 21.6 and 21.7, we see that the basic algorithms are effective for numerous applications where networks represent an abstract model of the computation.

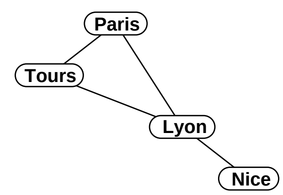
Road maps Tables that give distances between all pairs of major cities are a prominent feature of many road maps. We presume that the map maker took the trouble to be sure that the distances are the shortest ones, but our assumption is not necessarily always valid (see, for example, Exercise 21.11). Generally, such tables are for undirected graphs that we should treat as networks with edges in both directions corresponding to each road, though we might contemplate handling one-way streets for city maps and some similar applications.

As we see in Section 21.3, it is not difficult to provide other useful information, such as a table that tells how to execute the shortest paths (see Figure 21.4). In modern applications, embedded systems provide this kind of capability in cars and transportation systems. Maps are Euclidean graphs; in Section 21.4, we examine shortest-paths algorithms that take into account the vertex position when they seek shortest paths.

Airline routes Route maps and schedules for airlines or other transportation systems can be represented as networks for which various shortest-paths problems are of direct importance. For example, we might wish to minimize the time that it takes to fly between two cities or to minimize the cost of the trip. Costs in such networks might involve functions of time, of money, or of other complicated resources. For example, flights between two cities typically take more time in one direction than the other because of prevailing winds. Air travelers also know that the fare is not necessarily a simple function of the distance between the cities—situations where it is cheaper to use a circuitous route (or endure a stopover) than to take a direct flight are all too common. Such complications can be handled by the basic shortest-paths algorithms that we consider in this chapter; these algorithms are designed to handle any positive costs.

The fundamental shortest-paths computations suggested by these applications only scratch the surface of the applicability of shortest-paths algorithms. In Section 21.6, we consider problems from applications areas that appear unrelated to these natural ones, in the context of a discussion of *reduction*, a formal mechanism for proving relationships among problems. We solve problems for these applications by transforming them into abstract shortest-paths problems that do not have the intuitive geometric feel of the problems just described. Indeed, some applications lead us to consider shortest-paths problems in networks with negative weights. Such problems can be far more difficult to solve than are problems where negative weights cannot occur. Shortest-paths problems for such applications not only bridge a gap between elementary algorithms and unsolved algorithmic challenges but also lead us to powerful and general problem-solving mechanisms.

As with MST algorithms in Chapter 20, we often mix the weight, cost, and distance metaphors. Again, we normally exploit the natural appeal of geometric intuition even when working in more general



	Lyon	Nice	Paris	Tours
Lyon		200	400	420
Nice	200		600	620
Paris	400	600		120
Tours	420	620	120	

	Lyon	Nice	Paris	Tours
Lyon		Nice	Paris	Tours
Nice	Lyon		Lyon	Lyon
Paris	Lyon	Lyon		Tours
Tours	Lyon	Lyon	Paris	

Figure 21.4
Distances and paths

Road maps typically contain distance tables like the one in the center for this tiny subset of French cities connected by highways as shown in the graph at the top. Though rarely found in maps, a table like the one at the bottom would also be useful, as it tells what signs to follow to execute the shortest path. For example, to decide how to get from Paris to Nice, we can check the table, which says to begin by following signs to Lyon.

settings with arbitrary edge weights; thus we refer to the "length" of paths and edges when we should say "weight" and to one path as "shorter" than another when we should say that it "has lower weight." We also might say that `v` is "closer" to `s` than `w` when we should say that "the lowest-weight directed path from `s` to `v` has weight lower than that of the lowest-weight directed path `s` to `w`," and so forth. This usage is inherent in the standard use of the term "shortest paths" and is natural even when weights are not related to distances (see Figure 21.2); however, when we expand our algorithms to handle negative weights in Section 21.6, we must abandon such usage.

This chapter is organized as follows: After introducing the basic underlying principles in Section 21.1, we introduce basic algorithms for the single-source and all-pairs shortest-paths problems in Sections 21.2 and 21.3. Then, we consider acyclic networks (or, in a clash of shorthand terms, weighted DAGs) in Section 21.4 and ways of exploiting geometric properties for the source–sink problem in Euclidean graphs in Section 21.5. We then cast off in the other direction to look at more general problems in Sections 21.6 and 21.7, where we explore shortest-paths algorithms, perhaps involving networks with negative weights, as a high-level problem-solving tool.

Exercises

▷ **21.1** Label the following points in the plane 0 through 5, respectively:

```
(1, 3) (2, 1) (6, 5) (3, 4) (3, 7) (5, 3).
```

Taking edge lengths to be weights, consider the network defined by the edges

```
1-0 3-5 5-2 3-4 5-1 0-3 0-4 4-2 2-3.
```

Draw the network and give the adjacency-lists structure that is built by Program 20.5.

21.2 Show, in the style of Figure 21.3, all shortest paths in the network defined in Exercise 21.1.

∘ **21.3** Develop a network class implementation that represents the reverse of the weighted digraph defined by the edges inserted. Include a "reverse copy" constructor that takes a graph as parameter and inserts all that graph's edges to build its reverse.

∘ **21.4** Show that shortest-paths computations in networks with nonnegative weights on *both* vertices and edges (where the weight of a path is defined to be the sum of the weights of the vertices and the edges on the path) can be handled by building a network ADT that has weights on only the edges.

21.5 Find a large network online—perhaps a geographic database with entries for roads that connect cities or an airline or railroad schedule that contains distances or costs.

21.6 Write a random-network generator for sparse networks based on Program 17.12. To assign edge weights, define a random-edge–weight ADT and write two implementations: one that generates uniformly distributed weights, another that generates weights according to a Gaussian distribution. Write client programs to generate sparse random networks for both weight distributions with a well-chosen set of values of V and E so that you can use them to run empirical tests on graphs drawn from various distributions of edge weights.

∘ **21.7** Write a random-network generator for dense networks based on Program 17.13 and edge-weight generators as described in Exercise 21.6. Write client programs to generate random networks for both weight distributions with a well-chosen set of values of V and E so that you can use them to run empirical tests on graphs drawn from these models.

21.8 Implement a representation-independent network client that builds a network by taking edges with weights (pairs of integers between 0 and $V - 1$ with weights between 0 and 1) from standard input.

• **21.9** Write a program that generates V random points in the plane, then builds a network with edges (in both directions) connecting all pairs of points within a given distance d of one another (see Exercise 17.74), setting each edge's weight to the distance between the two points that that edge connects. Determine how to set d so that the expected number of edges is E.

∘ **21.10** Write a base class and derived classes that implement ADTs for graphs that may be undirected or directed graphs, weighted or unweighted, and dense or sparse.

▷ **21.11** The following table from a published road map purports to give the length of the shortest routes connecting the cities. It contains an error. Correct the table. Also, add a table that shows how to execute the shortest routes, in the style of Figure 21.4.

	Providence	Westerly	New London	Norwich
Providence	–	53	54	48
Westerly	53	–	18	101
New London	54	18	–	12
Norwich	48	101	12	–

21.1 Underlying Principles

Our shortest-paths algorithms are based on a simple operation known as *relaxation*. We start a shortest-paths algorithm knowing only the network's edges and weights. As we proceed, we gather information about the shortest paths that connect various pairs of vertices. Our algorithms all update this information incrementally, making new inferences about shortest paths from the knowledge gained so far. At each step, we test whether we can find a path that is shorter than some known path. The term "relaxation" is commonly used to describe this step, which *relaxes* constraints on the shortest path. We can think of a rubber band stretched tight on a path connecting two vertices: A successful relaxation operation allows us to relax the tension on the rubber band along a shorter path.

Our algorithms are based on applying repeatedly one of two types of relaxation operations:

- *Edge relaxation:* Test whether traveling along a given edge gives a new shortest path to its destination vertex.
- *Path relaxation:* Test whether traveling through a given vertex gives a new shortest path connecting two other given vertices.

Edge relaxation is a special case of path relaxation; we consider the operations separately, however, because we use them separately (the former in single-source algorithms; the latter in all-pairs algorithms). In both cases, the prime requirement that we impose on the data structures that we use to represent the current state of our knowledge about a network's shortest paths is that we can update them easily to reflect changes implied by a relaxation operation.

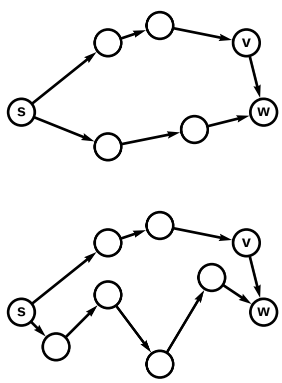


Figure 21.5
Edge relaxation

These diagrams illustrate the relaxation operation that underlies our single-source shortest-paths algorithms. We keep track of the shortest known path from the source `s` *to each vertex and ask whether an edge* `v-w` *gives us a shorter path to* `w`*. In the top example, it does not; so we would ignore it. In the bottom example, it does; so we would update our data structures to indicate that the best known way to get to* `w` *from* `s` *is to go to* `v`*, then take* `v-w`*.*

First, we consider edge relaxation, which is illustrated in Figure 21.5. All the single-source shortest-paths algorithms that we consider are based on this step: Does a given edge lead us to consider a shorter path to its destination from the source?

The data structures that we need to support this operation are straightforward. First, we have the basic requirement that we need to compute the shortest-paths lengths from the source to each of the other vertices. Our convention will be to store in a vertex-indexed array `wt` the lengths of the shortest known paths from the source to each vertex. Second, to record the paths themselves as we move from vertex to vertex, our convention will be the same as the one that we used for other graph-search algorithms that we examined in Chapters 18 through 20: We use a vertex-indexed array `spt` to record the last edge on a shortest path from the source to the indexed vertex. These edges constitute a tree.

With these data structures, implementing edge relaxation is a straightforward task. In our single-source shortest-paths code, we use the following code to relax along an edge `e` from `v` to `w`:

```
if (wt[w] > wt[v] + e.wt())
  { wt[w] = wt[v] + e.wt(); spt[w] = e; }
```

This code fragment is both simple and descriptive; we include it in this form in our implementations, rather than defining relaxation as a higher-level abstract operation.

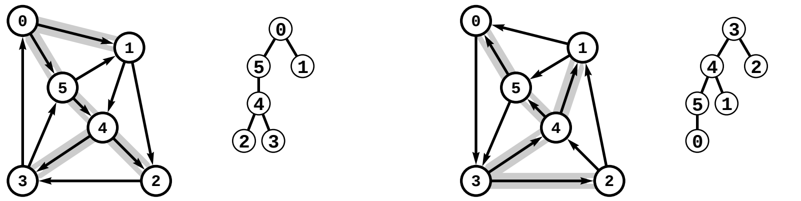
Definition 21.2 *Given a network and a designated vertex s, a* **shortest-paths tree (SPT)** *for s is a subnetwork containing s and all the vertices reachable from s that forms a directed tree rooted at s such that every tree path is a shortest path in the network.*

There may be multiple paths of the same length connecting a given pair of nodes, so SPTs are not necessarily unique. In general, as illustrated in Figure 21.2, if we take shortest paths from a vertex s to every vertex reachable from s in a network and from the subnetwork induced by the edges in the paths, we may get a DAG. Different shortest paths connecting pairs of nodes may each appear as a subpath in some longer path containing both nodes. Because of such effects, we generally are content to compute any SPT for a given digraph and start vertex.

Our algorithms generally initialize the entries in the `wt` array with a sentinel value. That value needs to be sufficiently small that the addition in the relaxation test does not cause overflow and sufficiently large that no simple path has a larger weight. For example, if edge weights are between `0` and `1`, we can use the value `V`. Note that we have to take extra care to check our assumptions when using sentinels in networks that could have negative weights. For example, if both vertices have the sentinel value, the relaxation code just given takes no action if `e.wt` is nonnegative (which is probably what we intend in most implementations), but it will change `wt[w]` and `spt[w]` if the weight is negative.

Our code always uses the *destination* vertex as the index to save the SPT edges (`spt[w].w() == w`). For economy and consistency with Chapters 17 through 19, we use the notation `st[w]` to refer to the vertex `spt[w].v()` (in the text and particularly in the figures) to emphasize that the `spt` array is actually a parent-link representation of the shortest-paths tree, as illustrated in Figure 21.6. We can compute

0-1	.41
1-2	.51
2-3	.50
4-3	.36
3-5	.38
3-0	.45
0-5	.29
5-4	.21
1-4	.32
4-2	.32
5-1	.29



	0	1	2	3	4	5
st	0	0	4	4	5	0
wt	0	.41	.32	.36	.21	.29

	0	1	2	3	4	5
st	5	4	3	3	3	4
wt	.29	.32	.50	0	.36	.21

Figure 21.6
Shortest-paths trees

The shortest paths from 0 *to the other nodes in this network are* 0-1*,* 0-5-4-2*,* 0-5-4-3*,* 0-5-4*, and* 0-5*, respectively. These paths define a spanning tree, which is depicted in three representations (gray edges in the network drawing, oriented tree, and parent links with weights) in the center. Links in the parent-link representation (the one that we typically compute) run in the opposite direction than links in the digraph, so we sometimes work with the reverse digraph. The spanning tree defined by shortest paths from* 3 *to each of the other nodes in the reverse is depicted on the right. The parent-link representation of this tree gives the shortest paths from each of the other nodes to* 2 *in the original graph. For example, we can find the shortest path* 0-5-4-3 *from* 0 *to* 3 *by following the links* st[0] = 5*,* st[5] = 4*, and* st[4] = 3*.*

the shortest path from s to t by traveling up the tree from t to s; when we do so, we are traversing edges in the direction opposite from their direction in the network and are visiting the vertices on the path in reverse order (t, st[t], st[st[t]], and so forth).

One way to get the edges on the path in order from source to sink from an SPT is to use a stack. For example, the following code prints a path from the source to a given vertex w:

```
EdgeStack P = new EdgeStack(G.V()); Edge e = st[v];
while (e != null) { P.push(e); e = st[e.()]); }
if (P.empty()) Out.print(P.top().v());
while (!P.empty())
  {  Out.print("-" + P.top().w()); P.pop(); }
```

In a class implementation, we could use code similar to this to provide clients with a array that contains the edges of the path.

If we simply want to print or otherwise process the edges on the path, going all the way through the path in reverse order to get to the first edge in this way may be undesirable. One approach to get around this difficulty is to work with the reverse network, as illustrated in Figure 21.6. We use reverse order and edge relaxation in single-source problems because the SPT gives a compact representation of the shortest paths from the source to all the other vertices, in an array with just V entries.

Next, we consider path relaxation, which is the basis of some of our all-pairs algorithms: Does going through a given vertex lead us to a shorter path that connects two other given vertices? For example, suppose that, for three vertices s, x, and t, we wish to know whether it is better to go from s to x and then from x to t or to go from s to t without going through x. For straight-line connections in a Euclidean space, the triangle inequality tells us that the route through x cannot be shorter than the direct route from s to t, but for paths in a network, it could be (see Figure 21.7). To determine which, we need to know the lengths of paths from s to x, x to t, and of those from s to t (that do not include x). Then, we simply test whether or not the sum of the first two is less than the third; if it is, we update our records accordingly.

Path relaxation is appropriate for all-pairs solutions where we maintain the lengths of the shortest paths that we have encountered between all pairs of vertices. Specifically, in all-pairs shortest-paths code of this kind, we maintain an array of arrays `d` such that `d[s][t]` is the shortest-path length from `s` to `t`, and we also maintain an array of arrays `p` such that `p[s][t]` is the next vertex on a shortest path from `s` to `t`. We refer to the former as the *distances* matrix and the latter as the *paths* matrix. Figure 21.8 shows the two matrices for our example network. The distances matrix is a prime objective of the computation, and we use the paths matrix because it is clearly more compact than, but carries the same information as, the full list of paths that is illustrated in Figure 21.3.

In terms of these data structures, path relaxation amounts to the following code:

```
if (d[s][t] > d[s][x] + d[x][t])
  { d[s][t] = d[s][x] + d[x][t]; p[s][t] = p[s][x]; }
```

Like edge relaxation, this code reads as a restatement of the informal description that we have given, so we use it directly in our implementations. More formally, path relaxation reflects the following:

Property 21.1 *If a vertex x is on a shortest path from s to t, then that path consists of a shortest path from s to x followed by a shortest path from x to t.*

Proof: By contradiction. We could use any shorter path from s to x or from x to t to build a shorter path from s to t. ■

We encountered the path-relaxation operation when we discussed transitive-closure algorithms, in Section 19.3. If the edge and path weights are either `1` or infinite (that is, a path's weight is `1` only if all that path's edges have weight `1`), then path relaxation is the operation that we used in Warshall's algorithm (if we have a path from `s` to `x` and a path from `x` to `t`, then we have a path from `s` to `t`). If we define a path's weight to be the number of edges on that path, then Warshall's algorithm generalizes to Floyd's algorithm for finding all

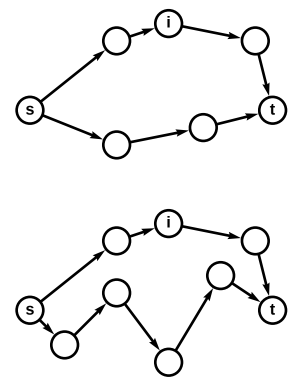
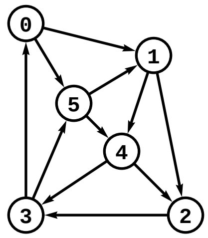


Figure 21.7
Path relaxation

These diagrams illustrate the relaxation operation that underlies our all-pairs shortest-paths algorithms. We keep track of the best known path between all pairs of vertices and ask whether a vertex `i` is evidence that the shortest known path from `s` to `t` could be improved. In the top example, it is not; in the bottom example, it is. Whenever we encounter a vertex `i` such that the length of the shortest known path from `s` to `i` plus the length of the shortest known path from `i` to `t` is smaller than the length of the shortest known path from `s` to `t`, then we update our data structures to indicate that we now know a shorter path from `s` to `t` (head towards `i` first).

```
0-1 .41
1-2 .51
2-3 .50
4-3 .36
3-5 .38
3-0 .45
0-5 .29
5-4 .21
1-4 .32
4-2 .32
5-1 .29
```



	0	1	2	3	4	5
0	0	.41	.82	.86	.50	.29
1	1.13	0	.51	.68	.32	1.06
2	.95	1.17	0	.50	1.09	.88
3	.45	.67	.91	0	.59	.38
4	.81	1.03	.32	.36	0	.74
5	1.02	.29	.53	.57	.21	0

	0	1	2	3	4	5
0	0	1	5	5	5	5
1	4	1	2	4	4	4
2	3	3	2	3	3	3
3	0	5	5	3	5	5
4	3	3	2	3	4	3
5	4	1	4	4	4	5

Figure 21.8
All shortest paths

The two matrices on the right are compact representations of all the shortest paths in the sample network on the left, containing the same information in the exhaustive list in Figure 21.3. The distances matrix on the left contains the shortest-path length: The entry in row `s` *and column* `t` *is the length of the shortest path from* `s` *to* `t`*. The paths matrix on the right contains the information needed to execute the path: The entry in row* `s` *and column* `t` *is the next vertex on the path from* `s` *to* `t`*.*

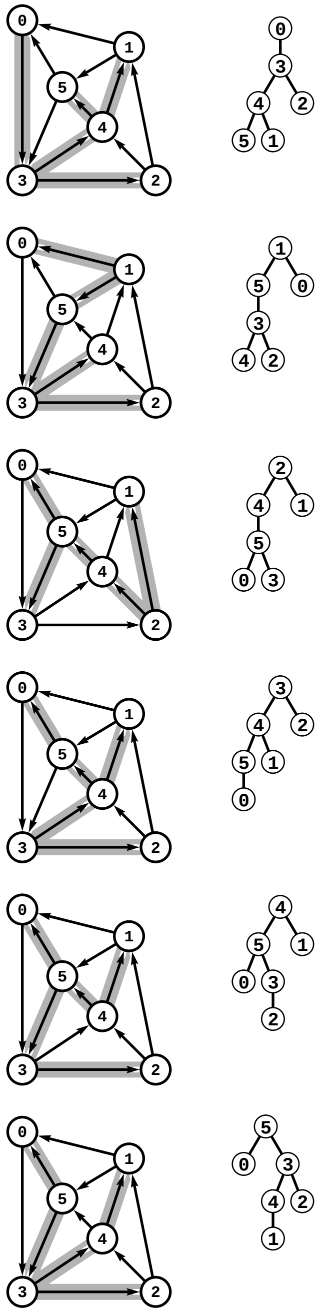
shortest paths in unweighted digraphs; it further generalizes to apply to networks, as we see in Section 21.3.

From a mathematician's perspective, it is important to note that these algorithms all can be cast in a general algebraic setting that unifies and helps us to understand them. From a programmer's perspective, it is important to note that we can implement each of these algorithms using an abstract `+` operator (to compute path weights from edge weights) and an abstract `<` operator (to compute the minimum value in a set of path weights), both solely in the context of the relaxation operation (see Exercises 19.55 and 19.56).

Property 21.1 implies that a shortest path from s to t contains shortest paths from s to every other vertex along the path to t. Most shortest-paths algorithms also compute shortest paths from s to every vertex that is closer to s than to t (whether or not the vertex is on the path from s to t), although that is not a requirement (see Exercise 21.18). Solving the source–sink shortest-paths problem with such an algorithm when t is the vertex that is farthest from s is equivalent to solving the single-source shortest-paths problem for s. Conversely, we could use a solution to the single-source shortest-paths problem from s as a method for finding the vertex that is farthest from s.

The paths matrix that we use in our implementations for the all-pairs problem is also a representation of the shortest-paths trees for each of the vertices. We defined `p[s][t]` to be the vertex that follows `s` on a shortest path from `s` to `t`. It is thus the same as the vertex that precedes `s` on the shortest path from `t` to `s` in the reverse network. In other words, *column* `t` in the paths matrix of a network is a vertex-indexed array that represents the SPT for vertex `t` in its reverse. Conversely, we can build the paths matrix for a network by filling each column with the vertex-indexed array representation of the

```
1-0 .41
2-1 .51
3-2 .50
3-4 .36
5-3 .38
0-3 .45
5-0 .29
4-5 .21
4-1 .32
2-4 .32
1-5 .29
```



	0	1	2	3	4	5
st	0	4	3	0	3	4
wt	0	1.13	.95	.45	.81	1.02

	0	1	2	3	4	5
st	1	1	3	5	3	1
wt	.41	0	1.17	.67	1.03	.29

	0	1	2	3	4	5
st	5	2	2	5	2	4
wt	.82	.51	0	.91	.32	.53

	0	1	2	3	4	5
st	5	4	3	3	3	4
wt	.86	.68	.50	0	.36	.57

	0	1	2	3	4	5
st	5	4	3	5	4	4
wt	.50	.32	1.09	.59	0	.21

	0	1	2	3	4	5
st	5	4	3	5	3	5
wt	.29	1.06	.88	.38	.74	0

Figure 21.9
All shortest paths in a network

These diagrams depict the SPTs for each vertex in the reverse of the network in Figure 21.8 (0 to 5, top to bottom)*, as network subtrees* (left)*, oriented trees* (center)*, and parent-link representation including a vertex-indexed array for path length* (right)*. Putting the arrays together to form path and distance matrices (where each array becomes a column) gives the solution to the all-pairs shortest-paths problem illustrated in Figure 21.8.*

SPT for the appropriate vertex in the reverse. This correspondence is illustrated in Figure 21.9.

In summary, relaxation gives us the basic abstract operations that we need to build our shortest paths algorithms. The primary complication is the choice of whether to provide the first or final edge on the shortest path. For example, single-source algorithms are more naturally expressed by providing the final edge on the path so that we need only a single vertex-indexed array to reconstruct the path, since all paths lead back to the source. This choice does not present a fundamental difficulty because we can either use the reverse graph as warranted or provide methods that hide this difference from clients. For example, we could specify a method in the interface that returns the edges on the shortest path in an array (see Exercises 21.15 and 21.16).

Accordingly, for simplicity, all of our implementations in this chapter include a method `dist` that returns a shortest-path length and *either* a method `path` that returns the first edge on a shortest path *or* a method `pathR` that returns the final edge on a shortest path. For example, our single-source implementations that use edge relaxation typically implement these methods as follows:

```
Edge pathR(int w) { return spt[w]; }
double dist(int v) { return wt[v]; }
```

Similarly, our all-paths implementations that use path relaxation typically implement these methods as follows:

```
Edge path(int s, int t) { return p[s][t]; }
double dist(int s, int t) { return d[s][t]; }
```

In some situations, it might be worthwhile to build interfaces that standardize on one or the other or both of these options; we choose the most natural one for the algorithm at hand.

Exercises

▷ **21.12** Draw the SPT from 0 for the network defined in Exercise 21.1 and for its reverse. Give the parent-link representation of both trees.

21.13 Consider the edges in the network defined in Exercise 21.1 to be *undirected* edges such that each edge corresponds to equal-weight edges in both directions in the network. Answer Exercise 21.12 for this corresponding network.

▷ **21.14** Change the direction of edge 0-2 in Figure 21.2. Draw two different SPTs that are rooted at 2 for this modified network.

21.15 Write a method that uses the `pathR` method from a single-source implementation to put references to the edges on the path from the source `v` to a given vertex `w` in a Java `Vector`.

21.16 Write a method that uses the `path` method from an all-paths implementation to put references to the edges on the path from a given vertex `v` to another given vertex `w` in a Java `Vector`.

21.17 Write a program that uses your method from Exercise 21.16 to print out all of the paths, in the style of Figure 21.3.

21.18 Give an example that shows how we could know that a path from `s` to `t` is shortest without knowing the length of a shorter path from `s` to `x` for some `x`.

21.2 Dijkstra's Algorithm

In Section 20.3, we discussed Prim's algorithm for finding the minimum spanning tree (MST) of a weighted undirected graph: We build it one edge at a time, always taking next the shortest edge that connects a vertex on the MST to a vertex not yet on the MST. We can use a nearly identical scheme to compute an SPT. We begin by putting the source on the SPT; then, we build the SPT one edge at a time, always taking next the edge that gives a shortest path from the source to a vertex not on the SPT. In other words, we add vertices to the SPT in order of their distance (through the SPT) to the start vertex. This method is known as *Dijkstra's algorithm*.

As usual, we need to make a distinction between the algorithm at the level of abstraction in this informal description and various concrete implementations (such as Program 21.1) that differ primarily in graph representation and priority-queue implementations, even though such a distinction is not always made in the literature. We shall consider other implementations and discuss their relationships with Program 21.1 after establishing that Dijkstra's algorithm correctly performs the single-source shortest-paths computation.

Property 21.2 *Dijkstra's algorithm solves the single-source shortest-paths problem in networks that have nonnegative weights.*

Proof: Given a source vertex s, we have to establish that the tree path from the root s to each vertex x in the tree computed by Dijkstra's algorithm corresponds to a shortest path in the graph from s to x. This fact follows by induction. Assuming that the subtree so far computed has the property, we need only to prove that adding a new vertex x adds a shortest path to that vertex. But all other paths to x must begin with a tree path followed by an edge to a vertex not on the tree. By construction, all such paths are longer than the one from s to x that is under consideration.

The same argument shows that Dijkstra's algorithm solves the source–sink shortest-paths problem, if we start at the source and stop when the sink comes off the priority queue. ■

The proof breaks down if the edge weights could be negative, because it assumes that a path's length does not decrease when we add more edges to the path. In a network with negative edge weights, this assumption is not valid because *any* edge that we encounter might lead to some tree vertex and might have a sufficiently large negative weight to give a path to that vertex shorter than the tree path. We consider this defect in Section 21.7 (see Figure 21.28).

Figure 21.10 shows the evolution of an SPT for a sample graph when computed with Dijkstra's algorithm; Figure 21.11 shows an oriented drawing of a larger SPT tree. Although Dijkstra's algorithm differs from Prim's MST algorithm in only the choice of priority, SPT trees are different in character from MSTs. They are rooted at the start vertex and all edges are directed away from the root, whereas MSTs are unrooted and undirected. We sometimes represent MSTs as directed, rooted trees when we use Prim's algorithm, but such structures are still different in character from SPTs (compare the oriented drawing in Figure 20.9 with the drawing in Figure 21.11). Indeed, the nature of the SPT somewhat depends on the choice of start vertex as well, as depicted in Figure 21.12.

Dijkstra's original implementation, which is suitable for dense graphs, is precisely like Prim's MST algorithm. Specifically, we simply change the assignment of the priority `P` in Program 20.6 from

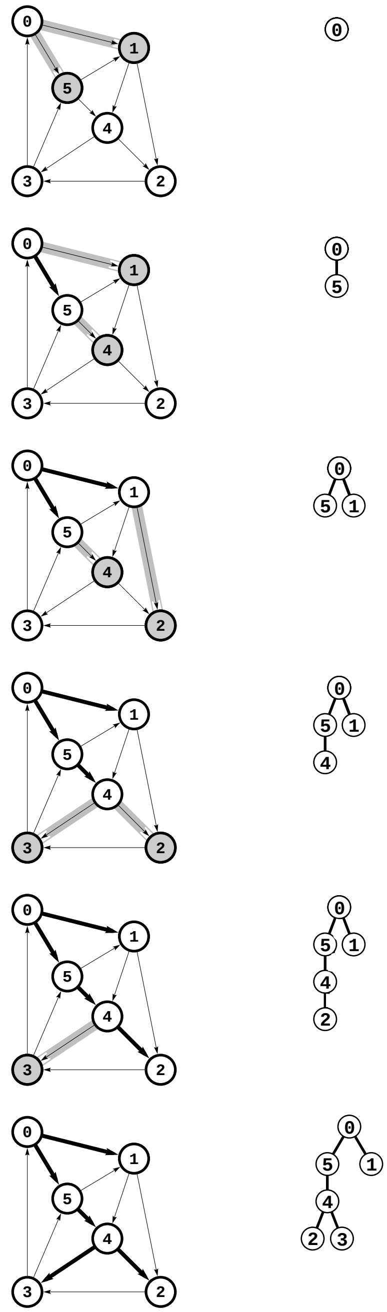
```
P = e.wt()
```

(the edge weight) to

```
P = wt[v] + e.wt()
```

(the distance from the source to the edge's destination). This change gives the classical implementation of Dijkstra's algorithm: We grow an SPT one edge at a time, each time updating the distance to the tree of all vertices adjacent to its destination while at the same time

```
0-1 .41
1-2 .51
2-3 .50
4-3 .36
3-5 .38
3-0 .45
0-5 .29
5-4 .21
1-4 .32
4-2 .32
5-1 .29
```



```
0-5 .29
0-1 .41

0-1 .41
5-4 .50

5-4 .50
1-2 .92

4-2 .82
4-3 .86

4-3 .86
```

Figure 21.10
Dijkstra's algorithm

This sequence depicts the construction of a shortest-paths spanning tree rooted at vertex `0` *by Dijkstra's algorithm for a sample network. Thick black edges in the network diagrams are tree edges, and thick gray edges are fringe edges. Oriented drawings of the tree as it grows are shown in the center, and a list of fringe edges is given on the right.*

The first step is to add `0` *to the tree and the edges leaving it,* `0-1` *and* `0-5`*, to the fringe* (top). *Second, we move the shortest of those edges,* `0-5`*, from the fringe to the tree and check the edges leaving it: The edge* `5-4` *is added to the fringe and the edge* `5-1` *is discarded because it is not part of a shorter path from* `0` *to* `1` *than the known path* `0-1` (second from top). *The priority of* `5-4` *on the fringe is the length of the path from* `0` *that it represents,* `0-5-4`*. Third, we move* `0-1` *from the fringe to the tree, add* `1-2` *to the fringe, and discard* `1-4` (third from top). *Fourth, we move* `5-4` *from the fringe to the tree, add* `4-3` *to the fringe, and replace* `1-2` *with* `4-2` *because* `0-5-4-2` *is a shorter path than* `0-1-2` (fourth from top). *We keep at most one edge to any vertex on the fringe, choosing the one on the shortest path from* `0`*. We complete the computation by moving* `4-2` *and then* `4-3` *from the fringe to the tree* (bottom).

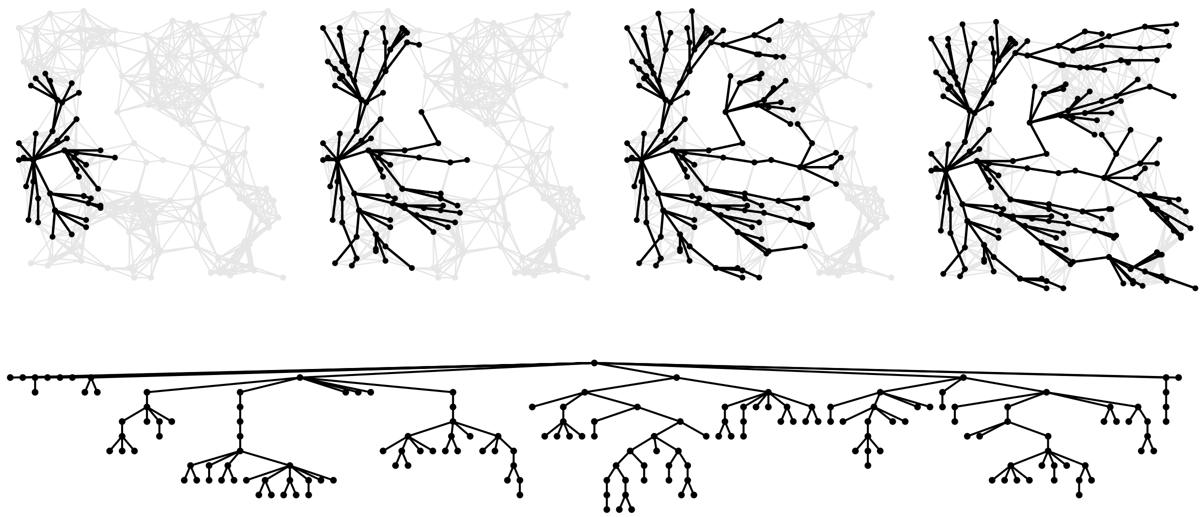


Figure 21.11
Shortest-paths spanning tree

This figure illustrates the progress of Dijkstra's algorithm in solving the single-source shortest-paths problem in a random Euclidean near-neighbor digraph (with directed edges in both directions corresponding to each line drawn), in the same style as Figures 18.13, 18.24, and 20.9. The search tree is similar in character to BFS because vertices tend to be connected to one another by short paths, but it is slightly deeper and less broad because distances lead to slightly longer paths than path lengths.

checking all the nontree vertices to find an edge to move to the tree whose destination vertex is a nontree vertex of minimal distance from the source.

Property 21.3 *With Dijkstra's algorithm, we can find any SPT in a dense network in linear time.*

Proof: As for Prim's MST algorithm, it is immediately clear, from inspection of the code of Program 20.6, that the nested loops mean that the running time is proportional to V^2, which is linear for dense graphs. ∎

For sparse graphs, we can do better by using a linked-list representation and a priority queue. Implementing this approach simply amounts to viewing Dijkstra's algorithm as a generalized graph-searching method that differs from depth-first search (DFS), from breadth-first search (BFS), and from Prim's MST algorithm in only the rule used to add edges to the tree. As in Chapter 20, we keep edges that connect tree vertices to nontree vertices on a generalized queue called the *fringe*, use a priority queue to implement the generalized queue, and provide for updating priorities so as to encompass DFS, BFS, and Prim's algorithm in a single implementation (see Section 20.3). This priority-first search (PFS) scheme also encompasses

Dijkstra's algorithm. That is, changing the assignment of P in Program 20.7 to

```
P = wt[v] + e.wt()
```

(the distance from the source to the edge's destination) gives an implementation of Dijkstra's algorithm that is suitable for sparse graphs.

Program 21.1 is an alternative PFS implementation for sparse graphs that is slightly simpler than Program 20.7 and that directly matches the informal description of Dijkstra's algorithm given at the beginning of this section. It differs from Program 20.7 in that it initializes the priority queue with all the vertices in the network and maintains the queue with the aid of a sentinel value for those vertices that are neither on the tree nor on the fringe (unseen vertices with sentinel values); in contrast, Program 20.7 keeps on the priority queue only those vertices that are reachable by a single edge from the tree. Keeping all the vertices on the queue simplifies the code but can incur a small performance penalty for some graphs (see Exercise 21.31).

The general results that we considered concerning the performance of priority-first search (PFS) in Chapter 20 give us specific information about the performance of these implementations of Dijkstra's algorithm for sparse graphs (Program 21.1 and Program 20.7, suitably modified). For reference, we restate those results in the present context. Since the proofs do not depend on the priority function, they apply without modification. They are worst-case results that apply to both programs, although Program 20.7 may be more efficient for many classes of graphs because it maintains a smaller fringe.

Property 21.4 *For all networks and all priority functions, we can compute a spanning tree with PFS in time proportional to the time required for V* **insert**, *V* **delete the minimum**, *and E* **decrease key** *operations in a priority queue of size at most V.*

Proof: This fact is immediate from the priority-queue–based implementations in Program 20.7 or Program 21.1. It represents a conservative upper bound because the size of the priority queue is often much smaller than V, particularly for Program 20.7. ■

Property 21.5 *With a PFS implementation of Dijkstra's algorithm that uses a heap for the priority-queue implementation, we can compute any SPT in time proportional to $E \lg V$.*

Program 21.1 Dijkstra's algorithm (priority-first search)

This class implements a single-source shortest-paths ADT with linear-time preprocessing, private data that takes space proportional to V, and constant-time member methods that give the length of the shortest path and the final vertex on the path from the source to any given vertex. The constructor is an implementation of Dijkstra's algorithm that uses a priority queue of vertices (in order of their distance from the source) to compute an SPT. The priority-queue interface is the same one used in Program 20.7 and implemented in Program 20.10.

The constructor is also a generalized graph search that implements other PFS algorithms with other assignments to the priority `P` (*see text*). The statement to reassign the weight of tree vertices to `0` is needed for a general PFS implementation but not for Dijkstra's algorithm, since the priorities of the vertices added to the SPT are nondecreasing.

```
class GraphSPT
{ private double[] wt;
  private Edge[] spt;
  GraphSPT(Graph G, int s)
  { int V = G.V();
    wt = new double[V]; spt = new Edge[V];
    for (int v = 0; v < V; v++) wt[v] = maxWT;
    doublePQi pQ = new doublePQi(V, wt);
    for (int v = 0; v < V; v++) pQ.insert(v);
    wt[s] = 0.0; pQ.lower(s);
    while (!pQ.empty())
    { int v = pQ.getmin(); // wt[v] = 0.0;
      if (v != s && spt[v] == null) return;
      AdjList A = G.getAdjList(v);
      for (Edge e = A.beg(); !A.end(); e = A.nxt())
        { int w = e.other(v);
          double P = wt[v] + e.wt();
          if (P < wt[w])
            { wt[w] = P; pQ.lower(w); spt[w] = e; }
        }
    }
  }
  Edge pathR(int v) { return spt[v]; }
  double dist(int v) { return wt[v]; }
}
```

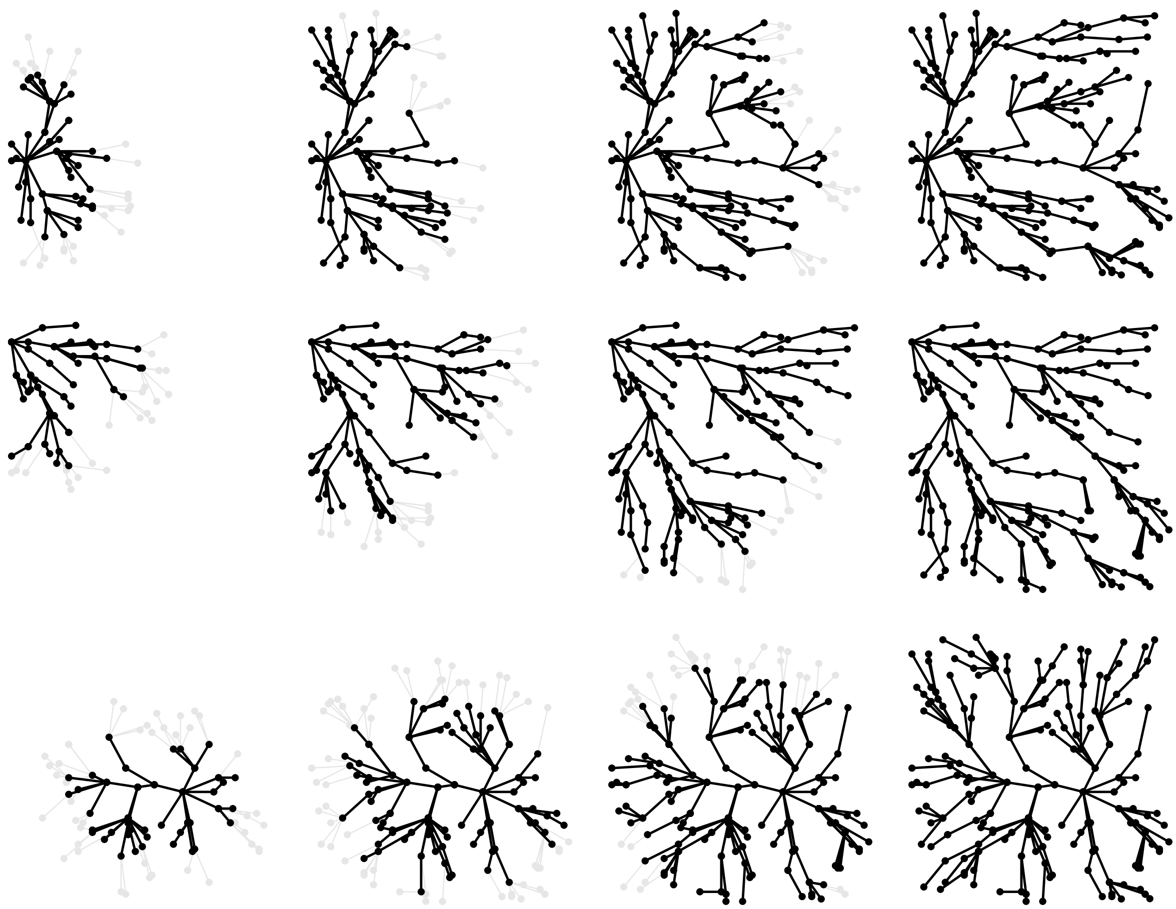


Figure 21.12
SPT examples

These three examples show growing SPTs for three different source locations: left edge (top)*, upper left corner* (center)*, and center* (bottom).

Proof: This result is a direct consequence of Property 21.4. ■

Property 21.6 *Given a graph with V vertices and E edges, let d denote the density E/V. If $d < 2$, then the running time of Dijkstra's algorithm is proportional to $V \lg V$. Otherwise, we can improve the worst-case running time by a factor of $\lg(E/V)$, to $O(E \lg_d V)$ (which is linear if E is at least $V^{1+\epsilon}$) by using a $\lceil E/V \rceil$-ary heap for the priority queue.*

Proof: This result directly mirrors Property 20.12 and the multiway-heap priority-queue implementation discussed directly thereafter. ■

Table 21.1 Priority-first search algorithms

These four classical graph-processing algorithms all can be implemented with PFS, a generalized priority-queue–based graph search that builds graph spanning trees one edge at a time. Details of search dynamics depend upon graph representation, priority-queue implementation, and PFS implementation; but the search trees generally characterize the various algorithms, as illustrated in the figures referenced in the fourth column.

algorithm	priority	result	Figure
DFS	reverse preorder	recursion tree	18.13
BFS	preorder	SPT (edges)	18.24
Prim	edge weight	MST	20.8
Dijkstra	path weight	SPT	21.9

Table 21.1 summarizes pertinent information about the four major PFS algorithms that we have considered. They differ in only the priority function used, but this difference leads to spanning trees that are entirely different from one another in character (as required). For the example in the figures referred to in the table (and for many other graphs), the DFS tree is tall and thin, the BFS tree is short and fat, the SPT is like the BFS tree but neither quite as short nor quite as fat, and the MST is neither short and fat nor tall and thin.

We have also considered four different implementations of PFS. The first is the classical dense-graph implementation that encompasses Dijkstra's algorithm and Prim's MST algorithm (Program 20.6); the other three are sparse-graph implementations that differ in priority-queue contents:

- Fringe edges (Program 18.10)
- Fringe vertices (Program 20.7)
- All vertices (Program 21.1)

Of these, the first is primarily of pedagogical value, the second is the most refined of the three, and the third is perhaps the simplest. This framework already describes 16 different implementations of classical graph-search algorithms—when we factor in different priority-queue implementations, the possibilities multiply further. This proliferation

Table 21.2 Cost of implementations of Dijkstra's algorithm

This table summarizes the cost (worst-case running time) of various implementations of Dijkstra's algorithm. With appropriate priority-queue implementations, the algorithm runs in linear time (time proportional to V^2 for dense networks, E for sparse networks), except for networks that are extremely sparse.

algorithm	worst-case cost	comment
classical	V^2	optimal for dense graphs
PFS, full heap	$E \lg V$	simplest implementation
PFS, fringe heap	$E \lg V$	conservative upper bound
PFS, d-heap	$E \lg_d V$	linear unless extremely sparse

of networks, algorithms, and implementations underscores the utility of the general statements about performance in Properties 21.4 through 21.6, which are also summarized in Table 21.2.

As is true of MST algorithms, actual running times of shortest-paths algorithms are likely to be lower than these worst-case time bounds suggest, primarily because most edges do not necessitate *decrease key* operations. In practice, except for the sparsest of graphs, we regard the running time as being linear.

The name *Dijkstra's algorithm* is commonly used to refer both to the abstract method of building an SPT by adding vertices in order of their distance from the source and to its implementation as the V^2 algorithm for the adjacency-matrix representation, because Dijkstra presented both in his 1959 paper (and also showed that the same approach could compute the MST). Performance improvements for sparse graphs are dependent on later improvements in ADT technology and priority-queue implementations that are not specific to the shortest-paths problem. Improved performance of Dijkstra's algorithm is one of the most important applications of that technology (*see reference section*). As with MSTs, we use terminology such as the "PFS implementation of Dijkstra's algorithm using d-heaps" to identify specific combinations.