```
            // Print the vertices in reverse finish order.
            System.out.println("The Topological Sort is");
            for (int i = numVertices - 1; i >= 0; i--) {
                System.out.println(finishOrder[i]);
            }
        }
    }
```

Testing Test this program using several different graphs. Use sparse graphs and dense graphs. Make sure that each graph you try has no loops or cycles. If it does, the algorithm may display an invalid output.

EXERCISES FOR SECTION 10.5

SELF-CHECK

1. Draw the depth-first search tree of the graph in Figure 10.24 and then list the vertices in reverse finish order.
2. List some alternative topological sorts for the graph in Figure 10.24.

10.6 Algorithms Using Weighted Graphs

Finding the Shortest Path from a Vertex to All Other Vertices

The breadth-first search discussed in Section 10.4 found the shortest path from the start vertex to all other vertices, assuming that the length of each edge was the same. We now consider the problem of finding the shortest path where the length of each edge may be different—that is, in a weighted directed graph such as that shown in Figure 10.26. The computer scientist Edsger W. Dijkstra developed an algorithm, now called Dijkstra's algorithm ("A Note on Two Problems in Connection with Graphs," *Numerische Mathematik*, Vol. 1 [1959], pp. 269–271), to solve this problem. This algorithm makes the assumption that all of the edge values are positive.

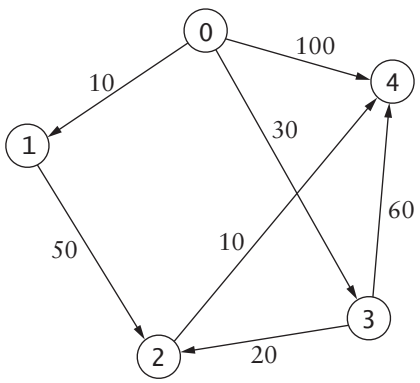

FIGURE 10.26 Weighted Directed Graph

For Dijkstra's algorithm, we need two sets, *S* and *V–S,* and two arrays, d and p. *S* will contain the vertices for which we have computed the shortest distance, and *V–S* will contain the vertices that we still need to process. The entry d[*v*] will contain the shortest distance from *s* to *v,* and p[*v*] will contain the predecessor of *v* in the path from *s* to *v.*

We initialize *S* by placing the start vertex, *s,* into it. We initialize *V–S* by placing the remaining vertices into it. For each *v* in *V–S*, we initialize d by setting d[*v*] equal to the weight of the edge *w*(*s, v*) for each vertex, *v,* adjacent to *s* and to ∞ for each vertex that is not adjacent to *s*. We initialize p[*v*] to *s* for each *v* in *V–S*.

For example, given the graph shown in Figure 10.26, the set *S* would initially be {0}, and *V–S* would be {1, 2, 3, 4}. The arrays d and p would be defined as follows.

v	d[v]	p[v]
1	10	0
2	∞	0
3	30	0
4	100	0

The first row shows that the distance from vertex 0 to vertex 1 is 10 and that vertex 0 is the predecessor of vertex 1. The second row shows that vertex 2 is not adjacent to vertex 0.

We now find the vertex *u* in *V–S* that has the smallest value of d[*u*]. Using our example, this is 1. We now consider the vertices *v* that are adjacent to *u*. If the distance from *s* to *u* (d[*u*]) plus the distance from *u* to *v* (i.e., *w*(*u, v*)) is smaller than the known distance from *s* to *v,* d[*v*], then we update d[*v*] to be d[*u*] + *w*(*u, v*), and we set p[*v*] to *u*. In our example, the value of d[1] is 10, and *w*(1, 2) is 50. Since 10 + 50 = 60 is less than ∞, we set d[2] to 60 and p[2] to 1. We remove 1 from *V–S* and place it into *S*. We repeat this until *V–S* is empty.

After the first pass through this loop, *S* is {0, 1}, *V–S* is {2, 3, 4}, and d and p are as follows:

v	d[v]	p[v]
1	10	0
2	60	1
3	30	0
4	100	0

We again select *u* from *V–S* with the smallest d[*u*]. This is now 3. The adjacent vertices to 3 are 2 and 4. The distance from 0 to 3, d[3], is 30. The distance from 3 to 2 is 20. Because 30 + 20 = 50 is less than the current value of d[2], 60, we update d[2] to 50 and change p[2] to 3. Also, because 30 + 60 = 90 is less than 100, we update d[4] to 90 and set p[4] to 3.

Now *S* is {0, 1, 3}, and *V–S* is {2, 4}. The arrays d and p are as follows:

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	90	3

Next, we select vertex 2 from *V*–*S*. The only vertex adjacent to 2 is 4. Since d[2] + $w(2, 4)$ = 50 + 10 = 60 is less than d[4], 90, we update d[4] to 60 and p[4] to 2. Now *S* is {0, 1, 2, 3}, *V*–*S* is {4}, and d and p are as follows:

v	d[*v*]	p[*v*]
1	10	0
2	50	3
3	30	0
4	60	2

Finally, we remove 4 from *V*–*S* and find that it has no adjacent vertices. We are now done. The array d shows the shortest distances from the start vertex to all other vertices, and the array p can be used to determine the corresponding paths. For example, the path from vertex 0 to vertex 4 has a length of 60, and it is the reverse of 4, 2, 3, 0; therefore, the shortest path is 0 → 3 → 2 → 4.

Dijkstra's Algorithm

1. Initialize *S* with the start vertex, *s*, and *V*–*S* with the remaining vertices.
2. **for** all *v* in *V*–*S*
3. Set p[*v*] to *s*.
4. **if** there is an edge (*s*, *v*)
5. Set d[*v*] to $w(s, v)$.
   else
6. Set d[*v*] to ∞.
7. **while** *V*–*S* is not empty
8. **for** all *u* in *V*–*S*, find the smallest d[*u*].
9. Remove *u* from *V*–*S* and add *u* to *S*.
10. **for** all *v* adjacent to *u* in *V*–*S*
11. **if** d[*u*] + $w(u, v)$ is less than d[*v*]
12. Set d[*v*] to d[*u*] + $w(u, v)$.
13. Set p[*v*] to *u*.

Analysis of Dijkstra's Algorithm

Step 1 requires $|V|$ steps.

The loop at Step 2 will be executed $|V - 1|$ times.

The loop at Step 7 will also be executed $|V - 1|$ times.

Within the loop at Step 7, we have to consider Steps 8 and 9. For these steps, we will have to search each value in *V*–*S*. This decreases each time through the loop at Step 7, so we will have $|V| - 1 + |V| - 2 + \cdots 1$. This is $\mathbf{O}(|V|^2)$. Therefore, Dijkstra's algorithm as stated is $\mathbf{O}(|V|^2)$. We will look at possible improvements to this for sparse graphs when we discuss a similar algorithm in the next subsection.

Implementation

Listing 10.7 provides a straightforward implementation of Dijkstra's algorithm using `HashSet vMinusS` to represent set *V*–*S*. We chose to implement the algorithm as a **`static`** method with the inputs (the graph and starting point) and outputs (predecessor and distance