CHAPTER

31

WEIGHTED GRAPHS AND APPLICATIONS

Objectives

- To represent weighted edges using adjacency matrices and priority queues (§31.2).
- To model weighted graphs using the `WeightedGraph` class that extends the `AbstractGraph` class (§31.3).
- To design and implement the algorithm for finding a minimum spanning tree (§31.4).
- To define the `MST` class that extends the `Tree` class (§31.4).
- To design and implement the algorithm for finding single-source shortest paths (§31.5).
- To define the `ShortestPathTree` class that extends the `Tree` class (§31.5).
- To solve the weighted nine tails problem using the shortest-path algorithm (§31.6).

31.1 Introduction

A graph is a weighted graph *if each edge is assigned a weight. Weighted graphs have many practical applications.*

Figure 30.1 assumes that the graph represents the number of flights among cities. You can apply the BFS to find the fewest number of flights between two cities. Assume that the edges represent the driving distances among the cities as shown in Figure 31.1. How do you find the minimal total distances for connecting all cities? How do you find the shortest path between two cities? This chapter will address these questions. The former is known as the *minimum spanning tree (MST) problem* and the latter as the *shortest path problem*.

problem

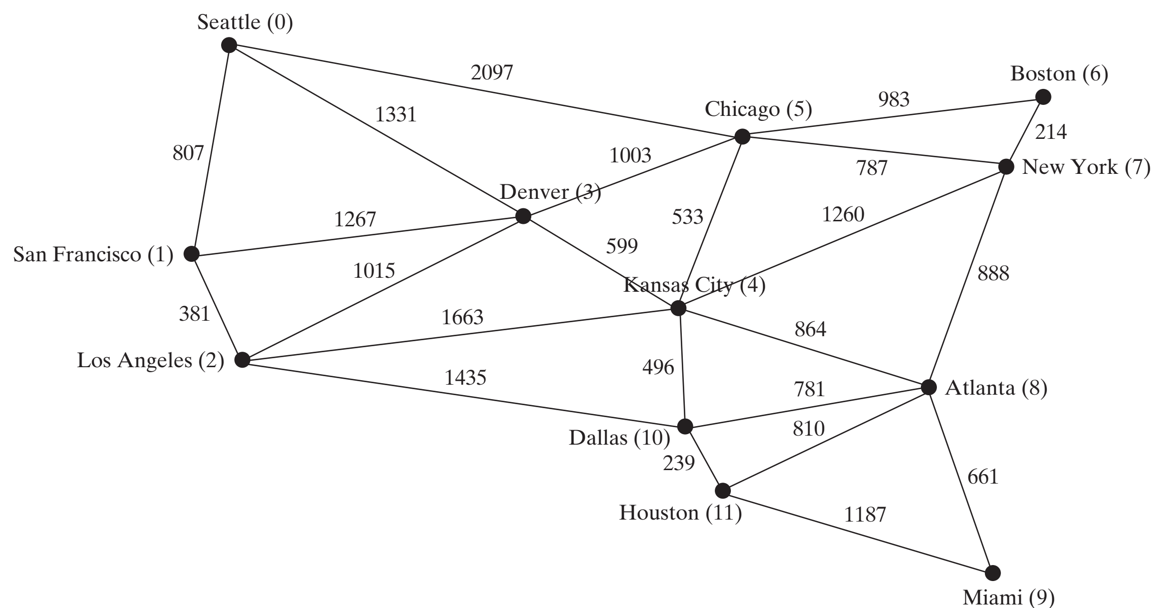


FIGURE 31.1 The graph models the distances among the cities.

The preceding chapter introduced the concept of graphs. You learned how to represent edges using edge arrays, edge lists, adjacency matrices, and adjacency lists, and how to model a graph using the `Graph` interface, the `AbstractGraph` class, and the `UnweightedGraph` class. The preceding chapter also introduced two important techniques for traversing graphs: depth-first search and breadth-first search, and applied traversal to solve practical problems. This chapter will introduce weighted graphs. You will learn the algorithm for finding a minimum spanning tree in Section 31.4 and the algorithm for finding shortest paths in Section 31.5.

weighted graph learning tool on Companion Website

Pedagogical Note

Before we introduce the algorithms and applications for weighted graphs, it is helpful to get acquainted with weighted graphs using the GUI interactive tool at www.cs.armstrong.edu/liang/animation/WeightedGraphLearningTool.html, as shown in Figure 31.2. The tool allows you to enter vertices, specify edges and their weights, view the graph, and find an MST and all shortest paths from a single source, as shown in Figure 31.2.

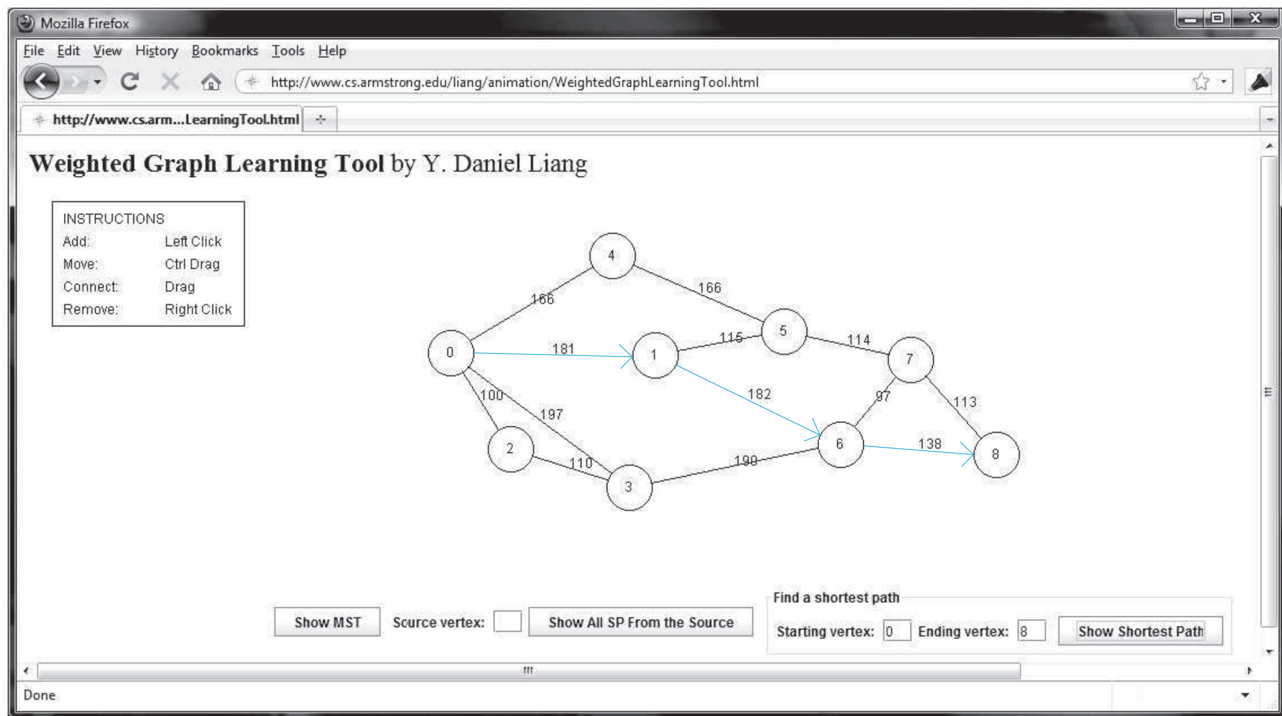


FIGURE 31.2 You can use the tool to create a weighted graph with mouse gestures and show the MST and shortest paths.

31.2 Representing Weighted Graphs

Often it is desirable to use a priority queue to store weighted edges.

There are two types of weighted graphs: vertex weighted and edge weighted. In a *vertex-weighted graph*, each vertex is assigned a weight. In an *edge-weighted graph*, each edge is assigned a weight. Of the two types, edge-weighted graphs have more applications. This chapter considers edge-weighted graphs.

vertex-weighted graph
edge-weighted graph

Weighted graphs can be represented in the same way as unweighted graphs, except that you have to represent the weights on the edges. As with unweighted graphs, the vertices in weighted graphs can be stored in an array. This section introduces three representations for the edges in weighted graphs.

31.2.1 Representing Weighted Edges: Edge Array

Weighted edges can be represented using a two-dimensional array. For example, you can store all the edges in the graph in Figure 31.3a using the array in Figure 31.3b.

Note

Weights can be of any type: **Integer**, **Double**, **BigDecimal**, and so on. You can use a two-dimensional array of the **Object** type to represent weighted edges as follows:

```
Object[][] edges = {
  {new Integer(0), new Integer(1), new SomeTypeForWeight(2)},
  {new Integer(0), new Integer(3), new SomeTypeForWeight(8)},
  ...
};
```

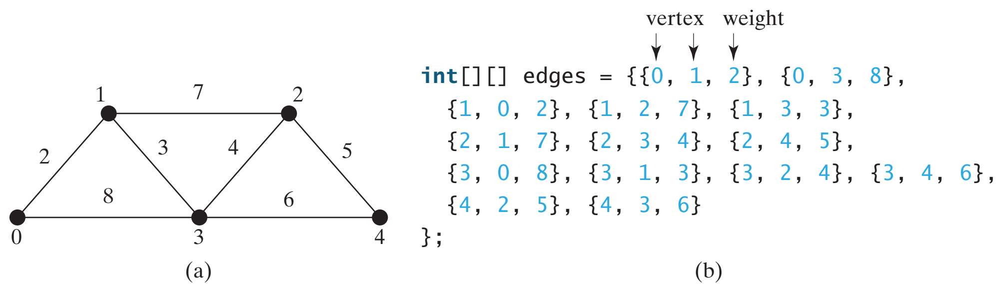


FIGURE 31.3 Each edge is assigned a weight in an edge-weighted graph.

31.2.2 Weighted Adjacency Matrices

Assume that the graph has n vertices. You can use a two-dimensional $n \times n$ matrix, say **weights**, to represent the weights on edges. **weights[i][j]** represents the weight on edge (**i**, **j**). If vertices **i** and **j** are not connected, **weights[i][j]** is **null**. For example, the weights in the graph in Figure 31.3a can be represented using an adjacency matrix as follows:

```
Integer[][] adjacencyMatrix = {
  {null, 2, null, 8, null},
  {2, null, 7, 3, null},
  {null, 7, null, 4, 5},
  {8, 3, 4, null, 6},
  {null, null, 5, 6, null}
};
```

	0	1	2	3	4
0	null	2	null	8	null
1	2	null	7	3	null
2	null	7	null	4	5
3	8	3	4	null	6
4	null	null	5	6	null

31.2.3 Priority Adjacency Lists

Another way to represent the edges is to define edges as objects. The **AbstractGraph.Edge** class was defined to represent an unweighted edge in Listing 30.3. For weighted edges, we define the **WeightedEdge** class as shown in Listing 31.1.

LISTING 31.1 WeightedEdge.java

edge weight

constructor

compare edges

```
 1 public class WeightedEdge extends AbstractGraph.Edge
 2     implements Comparable<WeightedEdge> {
 3   public double weight; // The weight on edge (u, v)
 4
 5   /** Create a weighted edge on (u, v) */
 6   public WeightedEdge(int u, int v, double weight) {
 7     super(u, v);
 8     this.weight = weight;
 9   }
10
11   @Override /** Compare two edges on weights */
12   public int compareTo(WeightedEdge edge) {
13     if (weight > edge.weight)
14       return 1;
15     else if (weight == edge.weight)
16       return 0;
17     else
18       return -1;
19   }
20 }
```

AbstractGraph.Edge is an inner class defined in the AbstractGraph class. It represents an edge from vertex u to v. WeightedEdge extends AbstractGraph.Edge with a new property weight.

To create a WeightedEdge object, use new WeightedEdge(i, j, w), where w is the weight on edge (i, j). It is often useful to store a vertex's adjacent edges in a priority queue so that you can remove the edges in increasing order of their weights. For this reason, the WeightedEdge class implements the Comparable interface.

For unweighted graphs, we use adjacency lists to represent edges. For weighted graphs, we still use adjacency lists, but the lists are priority queues. For example, the adjacency lists for the vertices in the graph in Figure 31.3a can be represented as follows:

```
java.util.PriorityQueue<WeightedEdge>[] queues =
  new java.util.PriorityQueue<WeightedEdge>[5];
```

queues[0]	WeightedEdge(0, 1, 2)	WeightedEdge(0, 3, 8)		
queues[1]	WeightedEdge(1, 0, 2)	WeightedEdge(1, 3, 3)	WeightedEdge(1, 2, 7)	
queues[2]	WeightedEdge(2, 3, 4)	WeightedEdge(2, 4, 5)	WeightedEdge(2, 1, 7)	
queues[3]	WeightedEdge(3, 1, 3)	WeightedEdge(3, 2, 4)	WeightedEdge(3, 4, 6)	WeightedEdge(3, 0, 8)
queues[4]	WeightedEdge(4, 2, 5)	WeightedEdge(4, 3, 6)		

queues[i] stores all edges adjacent to vertex i.

For flexibility, we will use an array list rather than a fixed-sized array to represent queues.

31.1 For the code WeightedEdge edge = new WeightedEdge(1, 2, 3.5), what is edge.u, edge.v, and edge.weight?

31.2 What is the printout of the following code?

```
List<WeightedEdge> list = new ArrayList<WeightedEdge>();
list.add(new WeightedEdge(1, 2, 3.5));
list.add(new WeightedEdge(2, 3, 4.5));
WeightedEdge e = java.util.Collections.max(list);
System.out.println(e.u);
System.out.println(e.v);
System.out.println(e.weight);
```

31.3 The WeightedGraph Class

The WeightedGraph class extends AbstractGraph.

The preceding chapter designed the Graph interface, the AbstractGraph class, and the UnweightedGraph class for modeling graphs. Following this pattern, we design WeightedGraph as a subclass of AbstractGraph, as shown in Figure 31.4.

WeightedGraph simply extends AbstractGraph with five constructors for creating concrete WeightedGraph instances. WeightedGraph inherits all methods from AbstractGraph, overrides the clear and addVertex methods, implements a new addEdge method for adding a weighted edge, and also introduces new methods for obtaining minimum spanning trees and for finding all single-source shortest paths. Minimum spanning trees and shortest paths will be introduced in Sections 31.4 and 31.5, respectively.

Listing 31.2 implements WeightedGraph. Priority adjacency lists (line 5) are used internally to store adjacent edges for a vertex. When a WeightedGraph is constructed, its priority adjacency lists are created (lines 15, 21, 27, and 34). The methods getMinimumSpanningTree() (lines 102–159) and getShortestPath() (lines 193–246) will be introduced in upcoming sections.

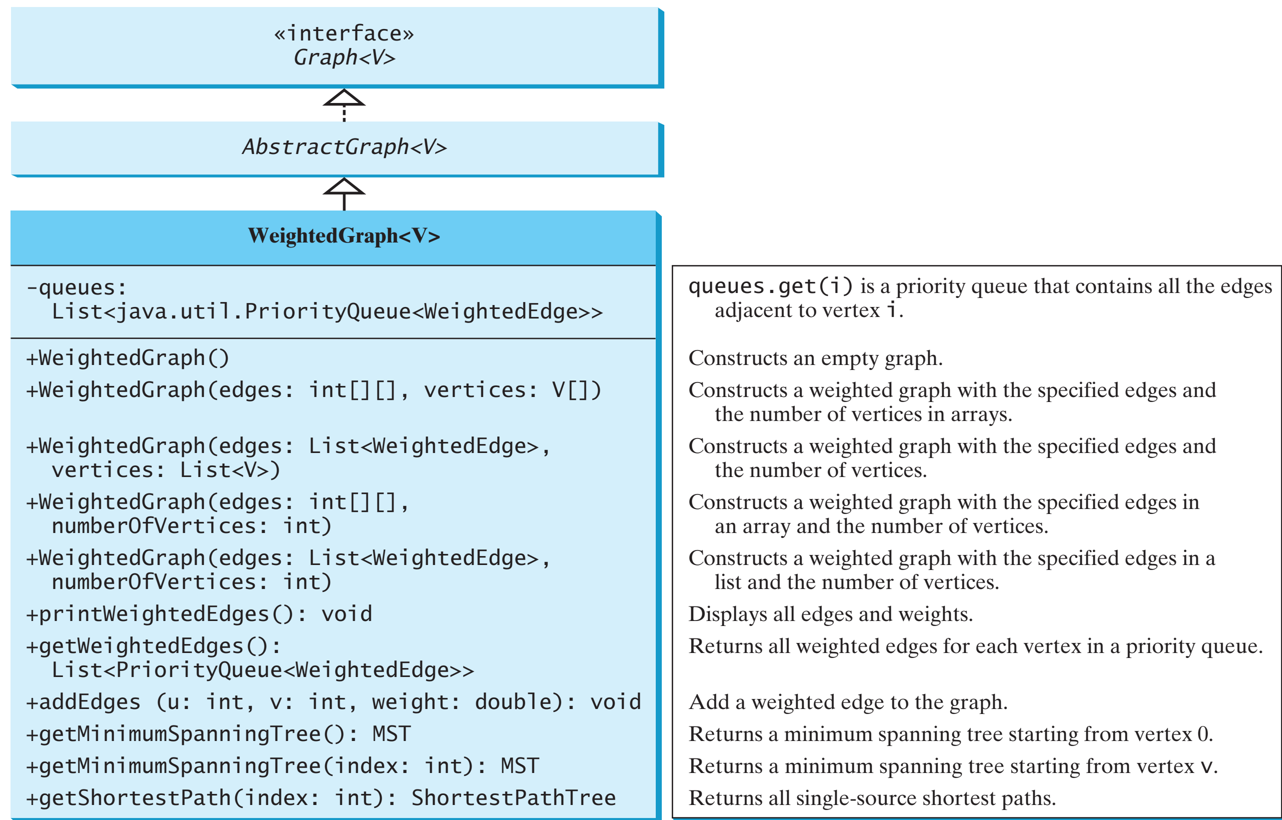


Figure 31.4 `WeightedGraph` extends `AbstractGraph`.

Listing 31.2 WeightedGraph.java

```
 1 import java.util.*;
 2
 3 public class WeightedGraph<V> extends AbstractGraph<V> {
 4   // Priority adjacency lists
 5   private List<PriorityQueue<WeightedEdge>> queues            // priority queue
 6     = new ArrayList<PriorityQueue<WeightedEdge>>();
 7
 8   /** Construct a WeightedGraph from edges and vertices in arrays */
 9   public WeightedGraph() {                                   // no-arg constructor
10   }
11
12   /** Construct a WeightedGraph from edges and vertices in arrays */
13   public WeightedGraph(int[][] edges, V[] vertices) {        // constructor
14     super(edges, vertices);                                  // superclass constructor
15     createQueues(edges, vertices.length);                    // create priority queues
16   }
17
18   /** Construct a WeightedGraph from edges and vertices in List */
19   public WeightedGraph(int[][] edges, int numberOfVertices) { // constructor
20     super(edges, numberOfVertices);
21     createQueues(edges, numberOfVertices);
22   }
23
24   /** Construct a WeightedGraph for vertices 0, 1, 2 and edge list */
25   public WeightedGraph(List<WeightedEdge> edges, List<V> vertices) { // constructor
26     super((List)edges, vertices);
```

```
27       createQueues(edges, vertices.size());
28     }
29
30     /** Construct a WeightedGraph from vertices 0, 1, and edge array */
31     public WeightedGraph(List<WeightedEdge> edges,                          constructor
32         int numberOfVertices) {
33       super((List)edges, numberOfVertices);
34       createQueues(edges, numberOfVertices);
35     }
36
37     /** Create priority adjacency lists from edge arrays */
38     private void createQueues(int[][] edges, int numberOfVertices) {   create priority queues
39       for (int i = 0; i < numberOfVertices; i++) {
40         queues.add(new PriorityQueue<WeightedEdge>()); // Create a queue
41       }
42
43       for (int i = 0; i < edges.length; i++) {
44         int u = edges[i][0];
45         int v = edges[i][1];
46         int weight = edges[i][2];
47         // Insert an edge into the queue
48         queues.get(u).offer(new WeightedEdge(u, v, weight));
49       }
50     }
51
52     /** Create priority adjacency lists from edge lists */
53     private void createQueues(List<WeightedEdge> edges,                  create queues
54         int numberOfVertices) {
55       for (int i = 0; i < numberOfVertices; i++) {
56         queues.add(new PriorityQueue<WeightedEdge>()); // Create a queue
57       }
58
59       for (WeightedEdge edge: edges) {
60         queues.get(edge.u).offer(edge); // Insert an edge into the queue
61       }
62     }
63
64     /** Display edges with weights */
65     public void printWeightedEdges() {                                   print edges
66       for (int i = 0; i < queues.size(); i++) {
67         System.out.print(getVertex(i) + " (" + i + "): ");
68         for (WeightedEdge edge : queues.get(i)) {
69           System.out.print("(" + edge.u +
70             ", " + edge.v + ", " + edge.weight + ") ");
71         }
72         System.out.println();
73       }
74     }
75
76     /** Get the edges from the weighted graph */
77     public List<PriorityQueue<WeightedEdge>> getWeightedEdges() {        get edges
78       return queues;
79     }
80
81     @Override  /** Clear the weighted graph */
82     public void clear() {                                                clear graph
83       vertices.clear();
84       neighbors.clear();
85       queues.clear();
86     }
```

```
87
88    @Override /** Add vertices to the weighted graph */
89    public void addVertex(V vertex) {
90      super.addVertex(vertex);
91      queues.add(new PriorityQueue<WeightedEdge>());
92    }
93
94    /** Add edges to the weighted graph */
95    public void addEdge(int u, int v, double weight) {
96      super.addEdge(u, v);
97      queues.get(u).add(new WeightedEdge(u, v, weight));
98      queues.get(v).add(new WeightedEdge(v, u, weight));
99    }
100
101   /** Get a minimum spanning tree rooted at vertex 0 */
102   public MST getMinimumSpanningTree() {
103     return getMinimumSpanningTree(0);
104   }
105
106   /** Get a minimum spanning tree rooted at a specified vertex */
107   public MST getMinimumSpanningTree(int startingVertex) {
108     List<Integer> T = new ArrayList<Integer>();
109     // T initially contains the startingVertex;
110     T.add(startingVertex);
111
112     int numberOfVertices = vertices.size();  // Number of vertices
113     int[] parent = new int[numberOfVertices]; // Parent of a vertex
114     // Initially set the parent of all vertices to -1
115     for (int i = 0; i < parent.length; i++)
116       parent[i] = -1;
117     double totalWeight = 0;  // Total weight of the tree thus far
118
119     // Clone the priority queue, so to keep the original queue intact
120     List<PriorityQueue<WeightedEdge>> queues = deepClone(this.queues);
121
122     // All vertices are found?
123     while (T.size() < numberOfVertices) {
124       // Search for the vertex with the smallest edge adjacent to
125       // a vertex in T
126       int v = -1;
127       double smallestWeight = Double.MAX_VALUE;
128       for (int u : T) {
129         while (!queues.get(u).isEmpty() &&
130             T.contains(queues.get(u).peek().v)) {
131           // Remove the edge from queues[u] if the adjacent
132           // vertex of u is already in T
133           queues.get(u).remove();
134         }
135
136         if (queues.get(u).isEmpty()) {
137           continue;  // Consider the next vertex in T
138         }
139
140         // Current smallest weight on an edge adjacent to u
141         WeightedEdge edge = queues.get(u).peek();
142         if (edge.weight < smallestWeight) {
143           v = edge.v;
144           smallestWeight = edge.weight;
145           // If v is added to the tree, u will be its parent
146           parent[v] = u;
```

add vertex (line 89)
add edge (line 95)
minimum spanning tree (line 102)
start from vertex 0 (line 103)
minimum spanning tree (line 107)
vertices in tree (line 108)
add to tree (line 110)
number of vertices (line 112)
parent array (line 113)
initialize parent (line 116)
total weight (line 117)
a copy of queues (line 120)
more vertices? (line 123)
every u in tree (line 128)
remove visited vertex (line 133)
`queues.get(u)` is empty (line 136)
smallest edge to u (line 141)
update `smallestWeight` (line 144)

```
147           }
148         } // End of for loop
149 
150         if (v != -1)
151           T.add(v); // Add a new vertex to the tree                add to tree
152         else
153           break; // The tree is not connected, a partial MST is found
154 
155         totalWeight += smallestWeight;                             update totalWeight
156       } // End of while loop
157 
158       return new MST(startingVertex, parent, T, totalWeight);
159     }
160 
161     /** Clone an array of queues */
162     private List<PriorityQueue<WeightedEdge>> deepClone(              clone queue
163       List<PriorityQueue<WeightedEdge>> queues) {
164       List<PriorityQueue<WeightedEdge>> copiedQueues =
165         new ArrayList<PriorityQueue<WeightedEdge>>();
166 
167       for (int i = 0; i < queues.size(); i++) {
168         copiedQueues.add(new PriorityQueue<WeightedEdge>());
169         for (WeightedEdge e : queues.get(i)) {
170           copiedQueues.get(i).add(e);                                clone every element
171         }
172       }
173 
174       return copiedQueues;
175     }
176 
177     /** MST is an inner class in WeightedGraph */
178     public class MST extends Tree {                                  MST inner class
179       private double totalWeight; // Total weight of the tree's edges   total weight in tree
180 
181       public MST(int root, int[] parent, List<Integer> searchOrder,
182           double totalWeight) {
183         super(root, parent, searchOrder);
184         this.totalWeight = totalWeight;
185       }
186 
187       public double getTotalWeight() {
188         return totalWeight;
189       }
190     }
191 
192     /** Find single-source shortest paths */
193     public ShortestPathTree getShortestPath(int sourceVertex) {      getShortestPath
194       // T stores the vertices of paths found so far
195       List<Integer> T = new ArrayList<Integer>();                    vertices found
196       // T initially contains the sourceVertex;
197       T.add(sourceVertex);                                           add source
198 
199       // vertices is defined in AbstractGraph
200       int numberOfVertices = vertices.size();                        number of vertices
201 
202       // parent[v] stores the previous vertex of v in the path
203       int[] parent = new int[numberOfVertices];                      parent array
204       parent[sourceVertex] = -1; // The parent of source is set to -1   parent of root
205 
206       // cost[v] stores the cost of the path from v to the source
```

```
207     double[] cost = new double[numberOfVertices];                        cost array
208     for (int i = 0; i < cost.length; i++) {
209       cost[i] = Double.MAX_VALUE; // Initial cost set to infinity
210     }
211     cost[sourceVertex] = 0; // Cost of source is 0                       source cost
212
213     // Get a copy of queues
214     List<PriorityQueue<WeightedEdge>> queues = deepClone(this.queues);   a copy of queues
215
216     // Expand T
217     while (T.size() < numberOfVertices) {                                more vertices left?
218       int v = -1; // Vertex to be determined                             determine one
219       double smallestCost = Double.MAX_VALUE; // Set to infinity
220       for (int u : T) {
221         while (!queues.get(u).isEmpty() &&
222             T.contains(queues.get(u).peek().v)) {
223           queues.get(u).remove(); // Remove the vertex in queue for u    remove visited vertex
224         }
225
226         if (queues.get(u).isEmpty()) {                                   queues.get(u) is empty
227           // All vertices adjacent to u are in T
228           continue;
229         }
230
231         WeightedEdge e = queues.get(u).peek();                           smallest edge to u
232         if (cost[u] + e.weight < smallestCost) {
233           v = e.v;
234           smallestCost = cost[u] + e.weight;                             update smallestCost
235           // If v is added to the tree, u will be its parent
236           parent[v] = u;                                                 v now found
237         }
238       } // End of for loop
239
240       T.add(v); // Add a new vertex to T                                 add to T
241       cost[v] = smallestCost;
242     } // End of while loop
243
244     // Create a ShortestPathTree
245     return new ShortestPathTree(sourceVertex, parent, T, cost);          create a path
246   }
247
248   /** ShortestPathTree is an inner class in WeightedGraph */
249   public class ShortestPathTree extends Tree {
250     private double[] cost; // cost[v] is the cost from v to source       cost
251
252     /** Construct a path */
253     public ShortestPathTree(int source, int[] parent,                    constructor
254         List<Integer> searchOrder, double[] cost) {
255       super(source, parent, searchOrder);
256       this.cost = cost;
257     }
258
259     /** Return the cost for a path from the root to vertex v */
260     public double getCost(int v) {                                       get cost
261       return cost[v];
262     }
263
264     /** Print paths from all vertices to the source */
265     public void printAllPaths() {                                        print all paths
266       System.out.println("All shortest paths from " +
```

```
267           vertices.get(getRoot()) + " are:");
268         for (int i = 0; i < cost.length; i++) {
269           printPath(i);  // Print a path from i to the source
270           System.out.println("(cost: " + cost[i] + ")"); // Path cost
271         }
272       }
273     }
274   }
```

When you construct a **WeightedGraph** using the no-arg constructor, the superclass's no-arg constructor is invoked. When you construct a **WeightedGraph** using the other four constructors, the superclass's constructor is invoked (lines 14, 20, 26, 33) to initialize the properties **vertices** and **neighbors** in **AbstractGraph**. Additionally, priority queues are created for instances of **WeightedGraph**. The **clear** and **addVertex** methods in **AbstractGraph** are overridden in lines 82–92 to handle the weighted edges. The **addEdge(u, v, weight)** method adds a new edge (**u**, **v**) with the specified weight to the graph (lines 95–99).

Listing 31.3 gives a test program that creates a graph for the one in Figure 31.1 and another graph for the one in Figure 31.3a.

LISTING 31.3 TestWeightedGraph.java

```
 1  public class TestWeightedGraph {
 2    public static void main(String[] args) {
 3      String[] vertices = {"Seattle", "San Francisco", "Los Angeles",      vertices
 4        "Denver", "Kansas City", "Chicago", "Boston", "New York",
 5        "Atlanta", "Miami", "Dallas", "Houston"};
 6
 7      int[][] edges = {                                                    edges
 8        {0, 1, 807}, {0, 3, 1331}, {0, 5, 2097},
 9        {1, 0, 807}, {1, 2, 381}, {1, 3, 1267},
10        {2, 1, 381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},
11        {3, 0, 1331}, {3, 1, 1267}, {3, 2, 1015}, {3, 4, 599},
12          {3, 5, 1003},
13        {4, 2, 1663}, {4, 3, 599}, {4, 5, 533}, {4, 7, 1260},
14          {4, 8, 864}, {4, 10, 496},
15        {5, 0, 2097}, {5, 3, 1003}, {5, 4, 533},
16          {5, 6, 983}, {5, 7, 787},
17        {6, 5, 983}, {6, 7, 214},
18        {7, 4, 1260}, {7, 5, 787}, {7, 6, 214}, {7, 8, 888},
19        {8, 4, 864}, {8, 7, 888}, {8, 9, 661},
20          {8, 10, 781}, {8, 11, 810},
21        {9, 8, 661}, {9, 11, 1187},
22        {10, 2, 1435}, {10, 4, 496}, {10, 8, 781}, {10, 11, 239},
23        {11, 8, 810}, {11, 9, 1187}, {11, 10, 239}
24      };
25
26      WeightedGraph<String> graph1 =
27        new WeightedGraph<String>(edges, vertices);                         create graph
28      System.out.println("The number of vertices in graph1: "
29        + graph1.getSize());
30      System.out.println("The vertex with index 1 is "
31        + graph1.getVertex(1));
32      System.out.println("The index for Miami is " +
33        graph1.getIndex("Miami"));
34      System.out.println("The edges for graph1:");
35      graph1.printWeightedEdges();                                          print edges
36
37      edges = new int[][]{                                                  edges
38        {0, 1, 2}, {0, 3, 8},
```

```
39         {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
40         {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
41         {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
42         {4, 2, 5}, {4, 3, 6}
43       };
44       WeightedGraph<Integer> graph2 =
45         new WeightedGraph<Integer>(edges, 5);
46       System.out.println("\nThe edges for graph2:");
47       graph2.printWeightedEdges();
48     }
49   }
```

create graph

print edges

```
The number of vertices in graph1: 12
The vertex with index 1 is San Francisco
The index for Miami is 9
The edges for graph1:
Vertex 0: (0, 1, 807) (0, 3, 1331) (0, 5, 2097)
Vertex 1: (1, 2, 381) (1, 0, 807) (1, 3, 1267)
Vertex 2: (2, 1, 381) (2, 3, 1015) (2, 4, 1663) (2, 10, 1435)
Vertex 3: (3, 4, 599) (3, 5, 1003) (3, 1, 1267)
  (3, 0, 1331) (3, 2, 1015)
Vertex 4: (4, 10, 496) (4, 8, 864) (4, 5, 533) (4, 2, 1663)
  (4, 7, 1260) (4, 3, 599)
Vertex 5: (5, 4, 533) (5, 7, 787) (5, 3, 1003)
  (5, 0, 2097) (5, 6, 983)
Vertex 6: (6, 7, 214) (6, 5, 983)
Vertex 7: (7, 6, 214) (7, 8, 888) (7, 5, 787) (7, 4, 1260)
Vertex 8: (8, 9, 661) (8, 10, 781) (8, 4, 864)
  (8, 7, 888) (8, 11, 810)
Vertex 9: (9, 8, 661) (9, 11, 1187)
Vertex 10: (10, 11, 239) (10, 4, 496) (10, 8, 781) (10, 2, 1435)
Vertex 11: (11, 10, 239) (11, 9, 1187) (11, 8, 810)

The edges for graph2:
Vertex 0: (0, 1, 2) (0, 3, 8)
Vertex 1: (1, 0, 2) (1, 2, 7) (1, 3, 3)
Vertex 2: (2, 3, 4) (2, 1, 7) (2, 4, 5)
Vertex 3: (3, 1, 3) (3, 4, 6) (3, 2, 4) (3, 0, 8)
Vertex 4: (4, 2, 5) (4, 3, 6)
```

The program creates **graph1** for the graph in Figure 31.1 in lines 3–27. The vertices for **graph1** are defined in lines 3–5. The edges for **graph1** are defined in lines 7–24. The edges are represented using a two-dimensional array. For each row **i** in the array, **edges[i][0]** and **edges[i][1]** indicate that there is an edge from vertex **edges[i][0]** to vertex **edges[i][1]** and the weight for the edge is **edges[i][2]**. For example, {**0**, **1**, **807**} (line 8) represents the edge from vertex **0** (**edges[0][0]**) to vertex **1** (**edges[0][1]**) with weight **807** (**edges[0][2]**). {**0**, **5**, **2097**} (line 8) represents the edge from vertex **0** (**edges[2][0]**) to vertex **5** (**edges[2][1]**) with weight **2097** (**edges[2][2]**). Line 35 invokes the **printWeightedEdges()** method on **graph1** to display all edges in **graph1**.

The program creates the edges for **graph2** for the graph in Figure 31.3a in lines 37–45. Line 47 invokes the **printWeightedEdges()** method on **graph2** to display all edges in **graph2**.

traversing priority queue

Note

The adjacent edges for each vertex are stored in a priority queue. When you remove an edge from the queue, the one with the smallest weight is always removed. However, if you traverse the edges in the queue, the edges are not necessarily in increasing order of weights.

31.3 What is the printout of the following code?

```
PriorityQueue<WeightedEdge> q =
  new PriorityQueue<WeightedEdge>();
q.offer(new WeightedEdge(1, 2, 3.5));
q.offer(new WeightedEdge(1, 6, 6.5));
q.offer(new WeightedEdge(1, 7, 1.5));
System.out.println(q.poll().weight);
System.out.println(q.poll().weight);
System.out.println(q.poll().weight);
```

31.4 What is wrong in the following code? Fix it and show the printout.

```
List<PriorityQueue<WeightedEdge>> queues =
  new ArrayList<PriorityQueue<WeightedEdge>>();
queues.get(0).offer(new WeightedEdge(0, 2, 3.5));
queues.get(0).offer(new WeightedEdge(0, 6, 6.5));
queues.get(0).offer(new WeightedEdge(0, 7, 1.5));
queues.get(1).offer(new WeightedEdge(1, 0, 3.5));
queues.get(1).offer(new WeightedEdge(1, 5, 8.5));
queues.get(1).offer(new WeightedEdge(1, 8, 19.5));
System.out.println(queues.get(0).peek()
  .compareTo(queues.get(1).peek()));
```

31.4 Minimum Spanning Trees

A minimum spanning tree of a graph is a spanning tree with the minimum total weights.

minimum spanning tree

A graph may have many spanning trees. Suppose that the edges are weighted. A *minimum spanning tree* has the minimum total weights. For example, the trees in Figures 31.5b, 31.5c, 31.5d are spanning trees for the graph in Figure 31.5a. The trees in Figures 31.3c and 31.3d are minimum spanning trees.

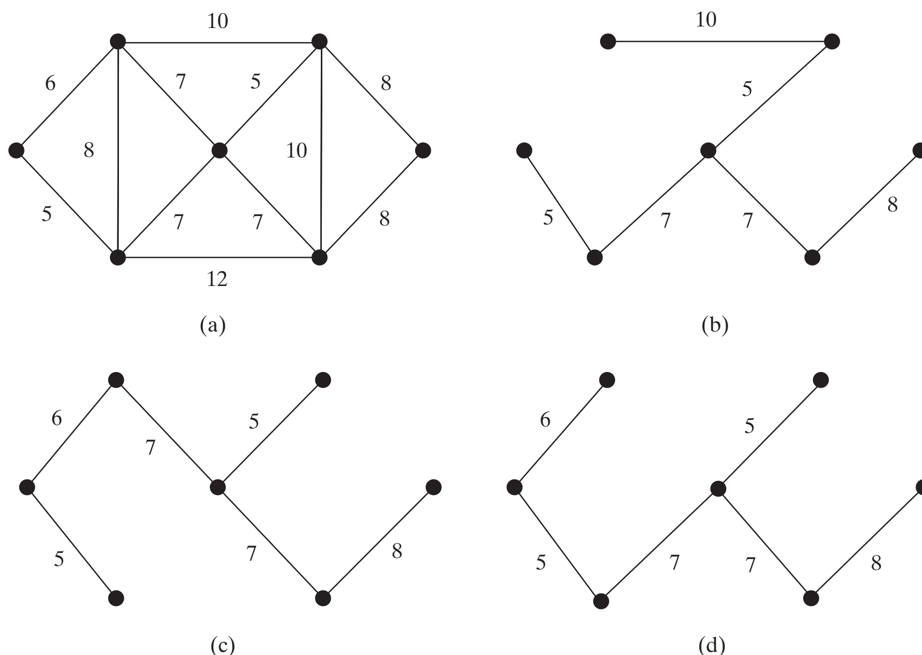


FIGURE 31.5 The trees in (c) and (d) are minimum spanning trees of the graph in (a).

The problem of finding a minimum spanning tree has many applications. Consider a company with branches in many cities. The company wants to lease telephone lines to connect all the branches together. The phone company charges different amounts of money to connect different pairs of cities. There are many ways to connect all branches together. The cheapest way is to find a spanning tree with the minimum total rates.

31.4.1 Minimum Spanning Tree Algorithms

Prim's algorithm

How do you find a minimum spanning tree? There are several well-known algorithms for doing so. This section introduces *Prim's algorithm*. Prim's algorithm starts with a spanning tree **T** that contains an arbitrary vertex. The algorithm expands the tree by repeatedly adding a vertex with the *lowest-cost* edge incident to a vertex already in the tree. Prim's algorithm is a greedy algorithm, and it is described in Listing 31.4.

LISTING 31.4 Prim's Minimum Spanning Tree Algorithm

```
 1  minimumSpanningTree() {
 2    Let V denote the set of vertices in the graph;
 3    Let T be a set for the vertices in the spanning tree;
 4    Initially, add the starting vertex to T;                 // add initial vertex
 5
 6    while (size of T < n) {                                  // more vertices?
 7      find u in T and v in V - T with the smallest weight    // find a vertex
 8        on the edge (u, v), as shown in Figure 31.6;
 9      add v to T;                                            // add to tree
10    }
11  }
```

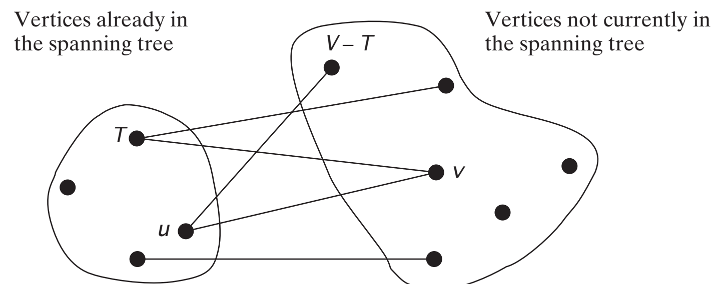


FIGURE 31.6 Find a vertex **u** in **T** that connects a vertex **v** in **V - T** with the smallest weight.

example

The algorithm starts by adding the starting vertex into **T**. It then continuously adds a vertex (say **v**) from **V - T** into **T**. **v** is the vertex that is adjacent to the vertex in **T** with the smallest weight on the edge. For example, there are five edges connecting vertices in **T** and **V - T** as shown in Figure 31.6, and (**u**, **v**) is the one with the smallest weight. Consider the graph in Figure 31.7. The algorithm adds the vertices to **T** in this order:

1. Add vertex **0** to **T**.
2. Add vertex **5** to **T**, since **Edge(5, 0, 5)** has the smallest weight among all edges incident to a vertex in **T**, as shown in Figure 31.7a.
3. Add vertex **1** to **T**, since **Edge(1, 0, 6)** has the smallest weight among all edges incident to a vertex in **T**, as shown in Figure 31.7b.

4. Add vertex `6` to `T`, since `Edge(6, 1, 7)` has the smallest weight among all edges incident to a vertex in `T`, as shown in Figure 31.7c.
5. Add vertex `2` to `T`, since `Edge(2, 6, 5)` has the smallest weight among all edges incident to a vertex in `T`, as shown in Figure 31.7d.
6. Add vertex `4` to `T`, since `Edge(4, 6, 7)` has the smallest weight among all edges incident to a vertex in `T`, as shown in Figure 31.7e.
7. Add vertex `3` to `T`, since `Edge(3, 2, 8`) has the smallest weight among all edges incident to a vertex in `T`, as shown in Figure 31.7f.

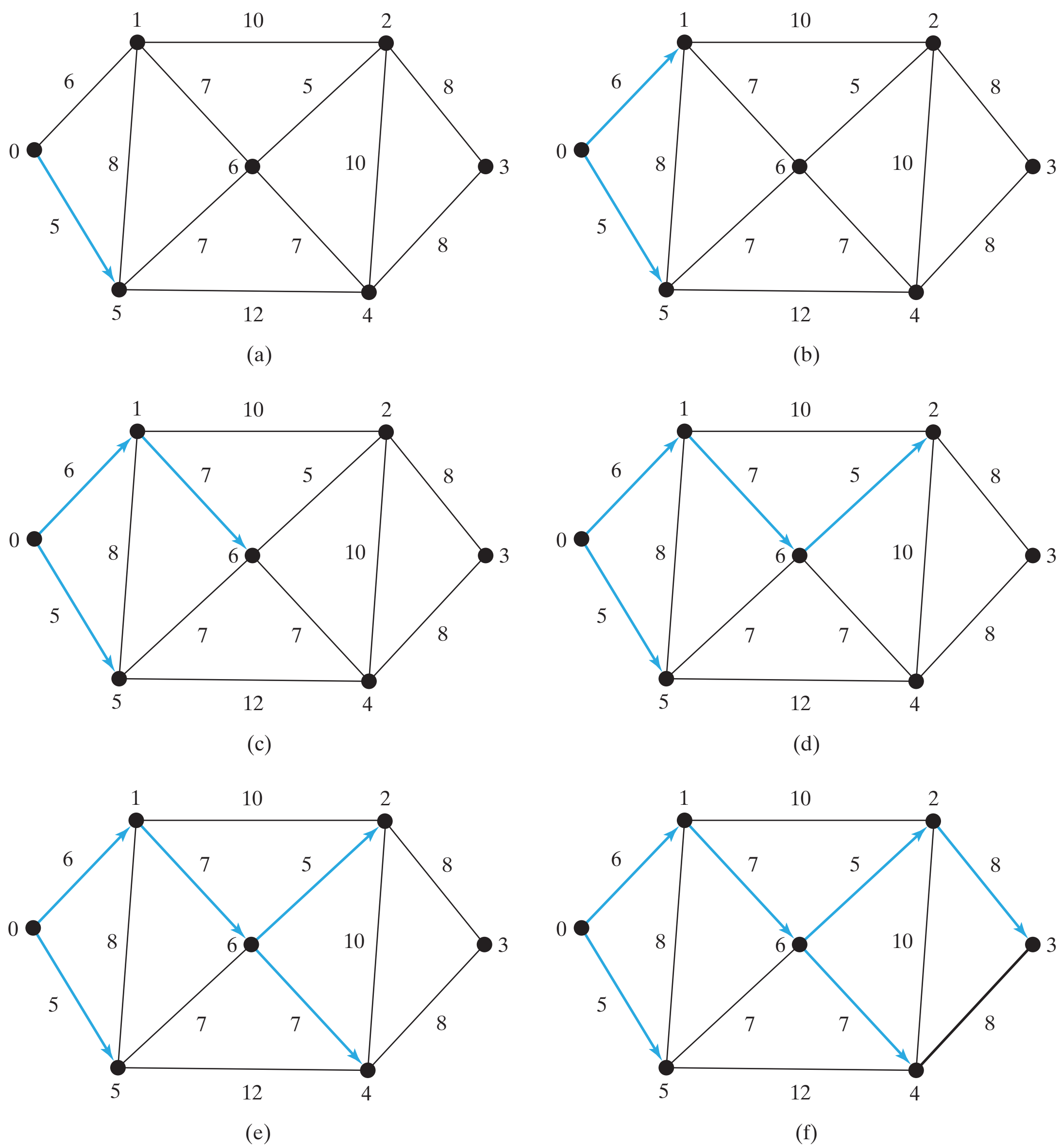


FIGURE 31.7 The adjacent vertices with the smallest weight are added successively to `T`.

unique tree?

Note

A minimum spanning tree is not unique. For example, both (c) and (d) in Figure 31.5 are minimum spanning trees for the graph in Figure 31.5a. However, if the weights are distinct, the graph has a unique minimum spanning tree.

connected and undirected

Note

Assume that the graph is connected and undirected. If a graph is not connected or directed, the algorithm will not work. You can modify the algorithm to find a spanning forest for any undirected graph. A spanning forest is a graph in which each connected component is a tree.

31.4.2 Implementation of the MST Algorithm

getMinimumSpanningTree()

The **getMinimumSpanningTree(int v)** method is defined in the **WeightedGraph** class. It returns an instance of the **MST** class, as shown in Figure 31.4. The **MST** class is defined as an inner class in the **WeightedGraph** class, which extends the **Tree** class, as shown in Figure 31.8. The **Tree** class was shown in Figure 30.11. The **MST** class was implemented in lines 178–190 in Listing 31.2.

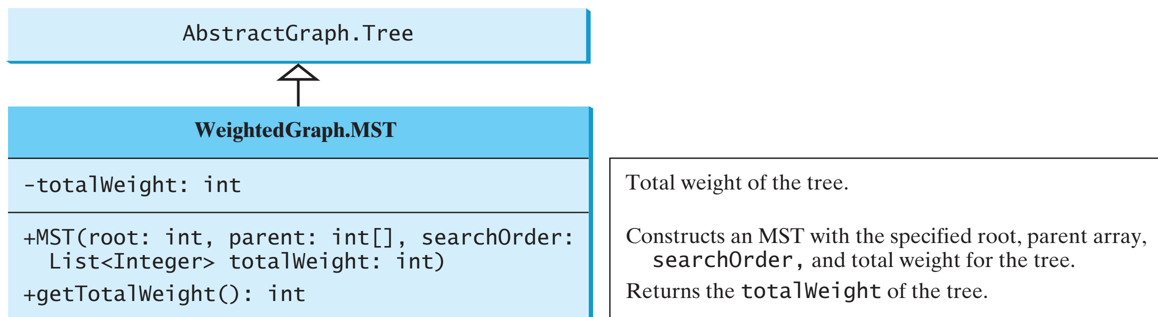


FIGURE 31.8 The **MST** class extends the **Tree** class.

The **getMinimumSpanningTree** method was implemented in lines 107–159 in Listing 31.2. The **getMinimumSpanningTree(int startingVertex)** method first adds **startingVertex** to **T** (line 110). **T** is a list that stores the vertices added into the spanning tree (line 108). **vertices** is defined as a protected data field in the **AbstractGraph** class, and it is an array list that stores all vertices in the graph. **vertices.size()** returns the number of the vertices in the graph (line 112).

A vertex is added to **T** if it is adjacent to one of the vertices in **T** with the smallest weight (line 151). Such a vertex is found using the following procedure:

1. For each vertex **u** in **T**, find its neighbor with the smallest weight to **u**. All the neighbors of **u** are stored in **queues.get(u)**. **queues.get(u).peek()** (line 130) returns the adjacent edge with the smallest weight. If a neighbor is already in **T**, remove it (line 133). To keep the original queues intact, a copy is created in line 120. After lines 129–138, **queues.get(u).peek()** (line 141) returns the vertex with the smallest weight to **u**.

2. Compare all these neighbors and find the one with the smallest weight (lines 141–147).

After a new vertex is added to **T** (line 151), **totalWeight** is updated (line 155). Once all vertices are added to **T**, an instance of **MST** is created (line 158). Note that the method will not work if the graph is not connected. However, you can modify it to obtain a partial MST.

The **MST** class extends the **Tree** class (line 178). To create an instance of **MST**, pass **root**, **parent**, **T**, and **totalWeight** (lines 181). The data fields **root**, **parent**, and **searchOrder** are defined in the **Tree** class, which is an inner class defined in **AbstractGraph**.

For each vertex, the program constructs a priority queue for its adjacent edges. It takes $O(\log|V|)$ time to insert an edge into a priority queue and the same time to remove an edge from the priority queue. Thus, the overall time complexity for the program is $O(|E| \log |V|)$, where $|E|$ denotes the number of edges and $|V|$ the number of vertices.

time complexity

Listing 31.5 gives a test program that displays minimum spanning trees for the graph in Figure 31.1 and the graph in Figure 31.3a, respectively.

LISTING 31.5 TestMinimumSpanningTree.java

```
1  public class TestMinimumSpanningTree {
2    public static void main(String[] args) {
3      String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
4        "Denver", "Kansas City", "Chicago", "Boston", "New York",
5        "Atlanta", "Miami", "Dallas", "Houston"};
6
7      int[][] edges = {
8        {0, 1, 807}, {0, 3, 1331}, {0, 5, 2097},
9        {1, 0, 807}, {1, 2, 381}, {1, 3, 1267},
10       {2, 1, 381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},
11       {3, 0, 1331}, {3, 1, 1267}, {3, 2, 1015}, {3, 4, 599},
12         {3, 5, 1003},
13       {4, 2, 1663}, {4, 3, 599}, {4, 5, 533}, {4, 7, 1260},
14         {4, 8, 864}, {4, 10, 496},
15       {5, 0, 2097}, {5, 3, 1003}, {5, 4, 533},
16         {5, 6, 983}, {5, 7, 787},
17       {6, 5, 983}, {6, 7, 214},
18       {7, 4, 1260}, {7, 5, 787}, {7, 6, 214}, {7, 8, 888},
19       {8, 4, 864}, {8, 7, 888}, {8, 9, 661},
20         {8, 10, 781}, {8, 11, 810},
21       {9, 8, 661}, {9, 11, 1187},
22       {10, 2, 1435}, {10, 4, 496}, {10, 8, 781}, {10, 11, 239},
23       {11, 8, 810}, {11, 9, 1187}, {11, 10, 239}
24     };
25
26     WeightedGraph<String> graph1 =
27       new WeightedGraph<String>(edges, vertices);
28     WeightedGraph<String>.MST tree1 = graph1.getMinimumSpanningTree();
29     System.out.println("Total weight is " + tree1.getTotalWeight());
30     tree1.printTree();
31
32     edges = new int[][]{
33       {0, 1, 2}, {0, 3, 8},
34       {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
35       {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
36       {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
37       {4, 2, 5}, {4, 3, 6}
38     };
39
40     WeightedGraph<Integer> graph2 =
41       new WeightedGraph<Integer>(edges, 5);
42     WeightedGraph<Integer>.MST tree2 =
```

create vertices (line 3)
create edges (line 7)
create graph1 (line 27)
MST for graph1 (line 28)
total weight (line 29)
print tree (line 30)
create edges (line 32)
create graph2 (line 41)

MST for graph2
total weight
print tree

```
43        graph2.getMinimumSpanningTree(1);
44      System.out.println("Total weight is " + tree2.getTotalWeight());
45      tree2.printTree();
46    }
47  }
```

```
Total weight is 6513.0
Root is: Seattle
Edges: (Seattle, San Francisco) (San Francisco, Los Angeles)
  (Los Angeles, Denver) (Denver, Kansas City) (Kansas City, Chicago)
  (New York, Boston) (Chicago, New York) (Dallas, Atlanta)
  (Atlanta, Miami) (Kansas City, Dallas) (Dallas, Houston)

Total weight is 14.0
Root is: 1
Edges: (1, 0) (3, 2) (1, 3) (2, 4)
```

The program creates a weighted graph for Figure 31.1 in line 27. It then invokes **getMinimumSpanningTree()** (line 28) to return an **MST** that represents a minimum spanning tree for the graph. Invoking **printTree()** (line 30) on the **MST** object displays the edges in the tree. Note that **MST** is a subclass of **Tree**. The **printTree()** method is defined in the **Tree** class.

graphical illustration

The graphical illustration of the minimum spanning tree is shown in Figure 31.9. The vertices are added to the tree in this order: Seattle, San Francisco, Los Angeles, Denver, Kansas City, Dallas, Houston, Chicago, New York, Boston, Atlanta, and Miami.

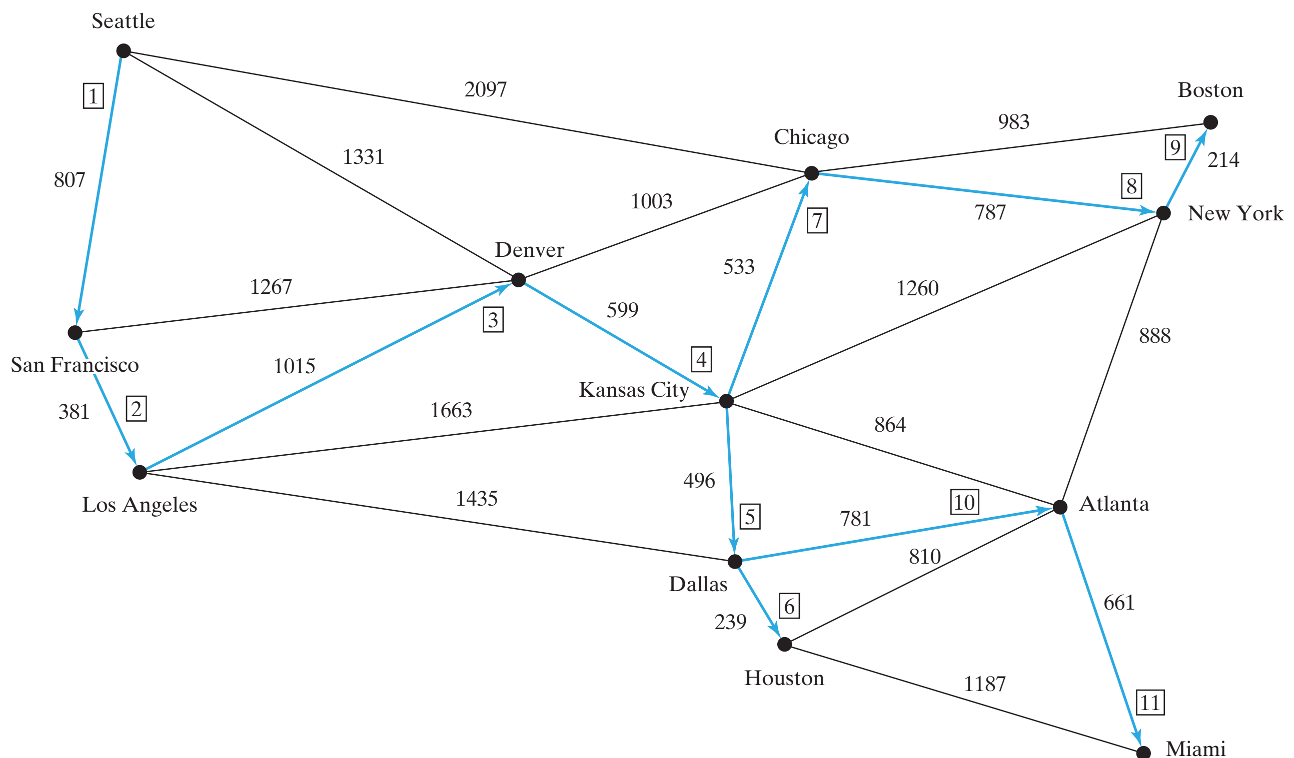
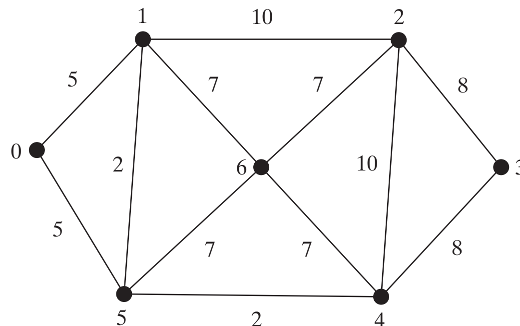


FIGURE 31.9 The edges in the minimum spanning tree for the cities are highlighted.

31.5 Find a minimum spanning tree for the following graph.



31.6 Is the minimum spanning tree unique if all edges have different weights?

31.7 If you use an adjacency matrix to represent weighted edges, what will be the time complexity for Prim's algorithm?

31.8 What happens to the `getMinimumSpanningTree()` method in `WeightedGraph` if the graph is not connected? Verify your answer by writing a test program that creates an unconnected graph and invokes the `getMinimumSpanningTree()` method. How do you fix the problem by obtaining a partial MST?

31.5 Finding Shortest Paths

The shortest path between two vertices is the path with the minimum total weights.

Given a graph with nonnegative weights on the edges, a well-known algorithm for finding a *shortest path* between two vertices was discovered by Edsger Dijkstra, a Dutch computer scientist. In order to find a shortest path from vertex `u` to vertex `v`, *Dijkstra's algorithm* finds the shortest path from `u` to all vertices. So *Dijkstra's algorithm* is known as a *single-source* shortest path algorithm. The algorithm uses `cost[v]` to store the cost of the *shortest path* from vertex `v` to the source vertex `s`. `cost[s]` is `0`. Initially assign infinity to `cost[v]` to indicate that no path is found from `v` to `s`. Let `V` denote all vertices in the graph and `T` denote the set of the vertices whose costs are known. Initially, the source vertex `s` is in `T`. The algorithm repeatedly finds a vertex `u` in `T` and a vertex `v` in `V - T` such that `cost[u] + w(u, v)` is the smallest, and moves `v` to `T`. Here, `w(u, v)` denotes the weight on edge (`u`, `v`).

shortest path
Dijkstra's algorithm
single-source shortest path

The algorithm is described in Listing 31.6.

Listing 31.6 Dijkstra's Single-Source Shortest-Path Algorithm

```
 1  shortestPath(s) {
 2    Let V denote the set of vertices in the graph;
 3    Let T be a set that contains the vertices whose
 4      paths to s are known;
 5    Initially T contains source vertex s with cost[s] = 0;
 6
 7    while (size of T < n) {
 8      find v in V - T with the smallest cost[u] + w(u, v) value
 9        among all u in T;
10      add v to T and set cost[v] = cost[u] + w(u, v);
11    }
12  }
```

add initial vertex
more vertex
find next vertex
add a vertex

This algorithm is very similar to Prim's for finding a minimum spanning tree. Both algorithms divide the vertices into two sets: `T` and `V - T`. In the case of Prim's algorithm, set `T` contains the vertices that are already added to the tree. In the case of Dijkstra's, set `T` contains the vertices whose shortest paths to the source have been found. Both algorithms repeatedly find a vertex from `V - T` and add it to `T`. In the case of Prim's algorithm, the vertex is adjacent to some vertex in the set with the minimum weight on the edge. In Dijkstra's algorithm, the vertex is adjacent to some vertex in the set with the minimum total cost to the source.

The algorithm starts by adding the source vertex `s` into `T` and sets `cost[s]` to `0` (line 5). It then continuously adds a vertex (say `v`) from `V - T` into `T`. `v` is the vertex that is adjacent to a vertex `u` in `T` with the smallest `cost[u] + w(u, v)`. For example, there are five edges connecting vertices in `T` and `V - T`, as shown in Figure 31.10; `(u, v)` is the one with the smallest `cost[u] + w(u, v)`. After `v` is added to `T`, set `cost[v]` to `cost[u] + w(u, v)` (line 10).

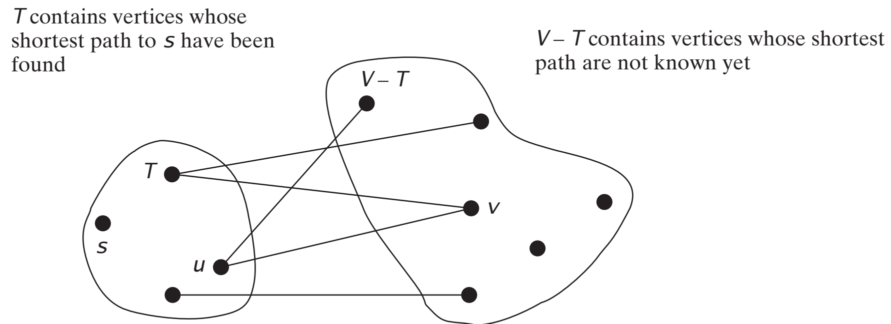


FIGURE 31.10 Find a vertex `u` in `T` that connects a vertex `v` in `V - T` with the smallest `cost[u] + w(u, v)`.

Let us illustrate Dijkstra's algorithm using the graph in Figure 31.11a. Suppose the source vertex is `1`. Therefore, `cost[1]` is `0` and the costs for all other vertices are initially ∞, as shown in Figure 31.11b. We use the `parent[i]` to denote the parent of `i` in the path. For convenience, set the parent of the source node to `-1`.

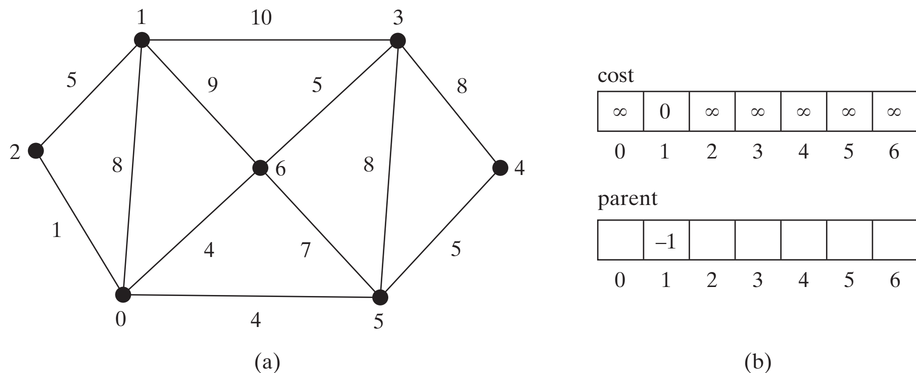


FIGURE 31.11 The algorithm will find all shortest paths from source vertex `1`.

Initially set `T` contains the source vertex. Vertices `2`, `0`, `6`, and `3` are adjacent to the vertices in `T`, and vertex `2` has the path of smallest cost to source vertex `1`, so add `2` to `T`. `cost[2]` now becomes `5`, as shown in Figure 31.12.

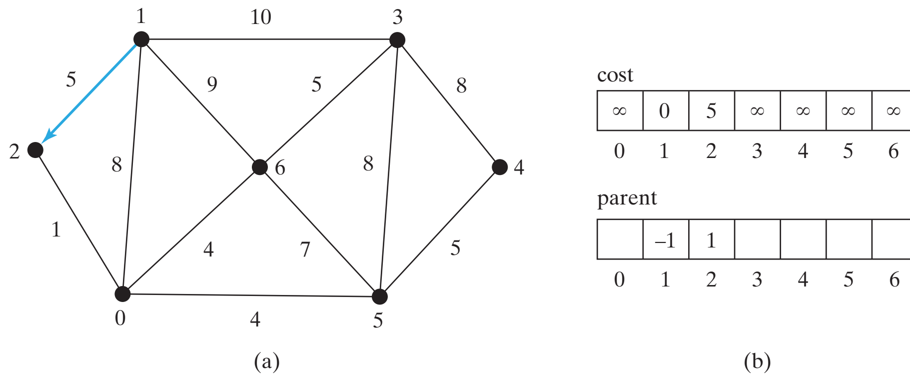


FIGURE 31.12 Now vertices **1** and **2** are in set **T**.

Now **T** contains {**1**, **2**}. Vertices **0**, **6**, and **3** are adjacent to the vertices in **T**, and vertex **0** has a path of smallest cost to source vertex **1**, so add **0** to **T**. **cost[0]** now becomes **6**, as shown in Figure 31.13.

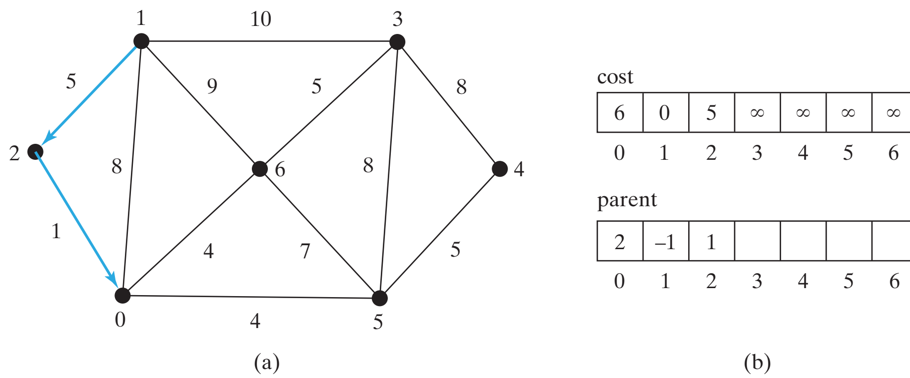


FIGURE 31.13 Now vertices {**1**, **2**, **0**} are in set **T**.

Now **T** contains {**1**, **2**, **0**}. Vertices **3**, **6**, and **5** are adjacent to the vertices in **T**, and vertex **6** has the path of smallest cost to source vertex **1**, so add **6** to **T**. **cost[6]** now becomes **9**, as shown in Figure 31.14.

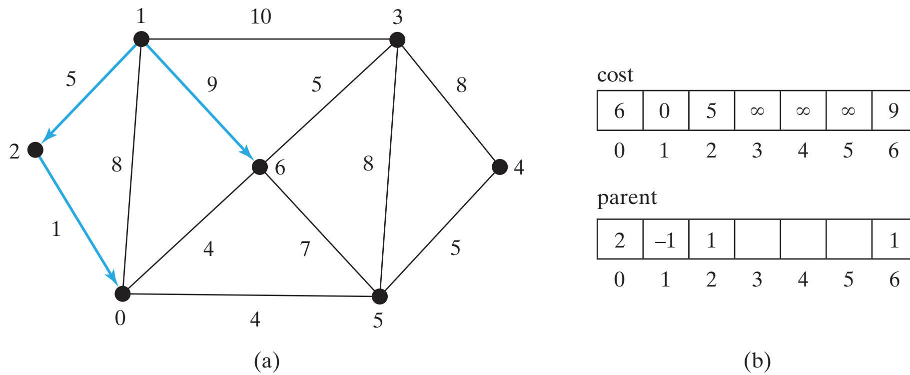


FIGURE 31.14 Now vertices {**1**, **2**, **0**, **6**} are in set **T**.

Now `T` contains {`1`, `2`, `0`, `6`}. Vertices `3` and `5` are adjacent to the vertices in `T`, and both vertices have a path of the same smallest cost to source vertex `1`. You can choose either `3` or `5`. Let us add `3` to `T`. `cost[3]` now becomes `10`, as shown in Figure 31.15.

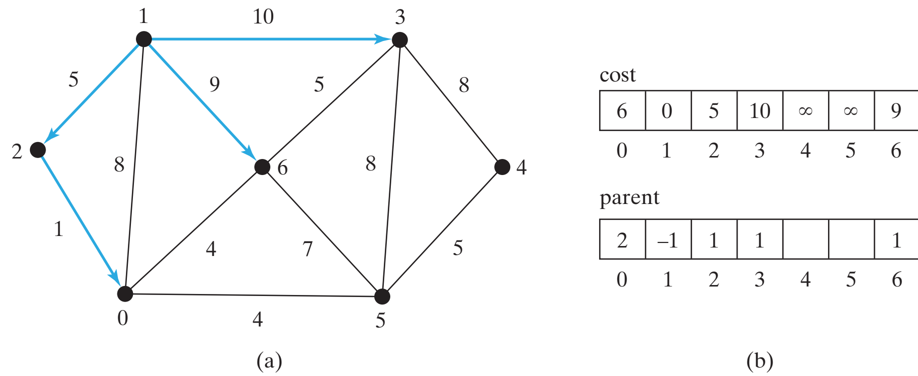


FIGURE 31.15 Now vertices {`1`, `2`, `0`, `6`, `3`} are in set `T`.

Now `T` contains {`1`, `2`, `0`, `6`, `3`}. Vertices `4` and `5` are adjacent to the vertices in `T`, and vertex `5` has the path of smallest cost to source vertex `1`, so add `5` to `T`. `cost[5]` now becomes `10`, as shown in Figure 31.16.

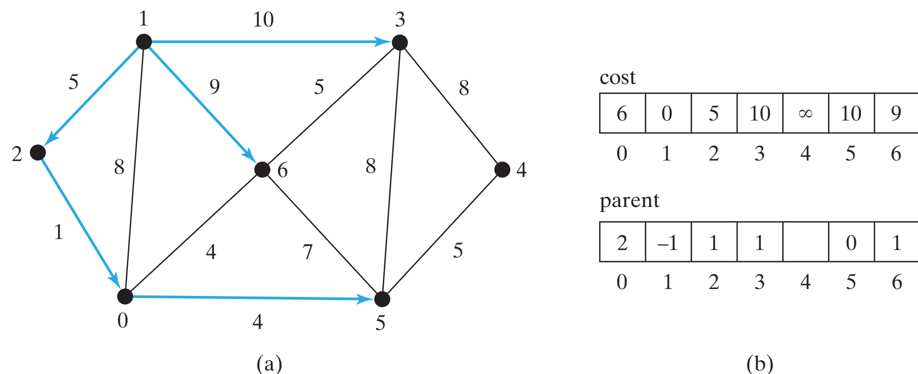


FIGURE 31.16 Now vertices {`1`, `2`, `0`, `6`, `3`, `5`} are in set `T`.

Now `T` contains {`1`, `2`, `0`, `6`, `3`, `5`}. The smallest cost for a path to connect `4` with `1` is `15`, as shown in Figure 31.17.

As you can see, the algorithm essentially finds all the shortest paths from a source vertex, which produces a tree rooted at the source vertex. We call this tree a *single-source all-shortest-path tree* (or simply a *shortest-path tree*). To model this tree, define a class named `ShortestPathTree` that extends the `Tree` class, as shown in Figure 31.18. `ShortestPathTree` is defined as an inner class in `WeightedGraph` in lines 249–273 in Listing 31.2.

shortest-path tree

The `getShortestPath(int sourceVertex)` method was implemented in lines 193–246 in Listing 31.2. The method first adds `sourceVertex` to `T` (line 197). `T` is a list that

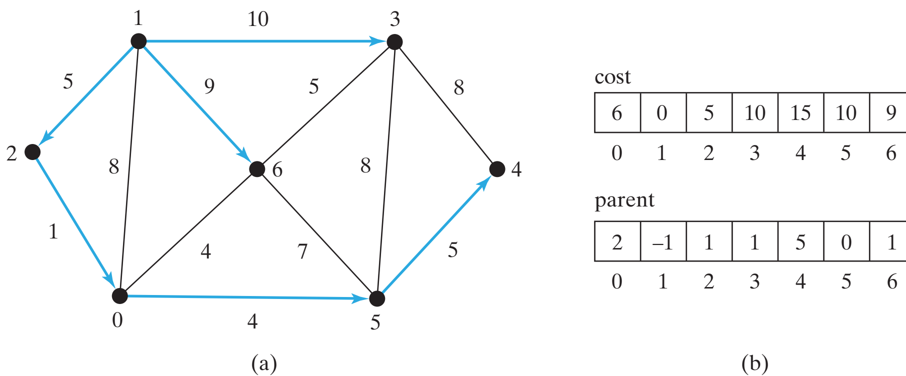


Figure 31.17 Now vertices {1, 2, 6, 0, 3, 5, 4} are in set T.

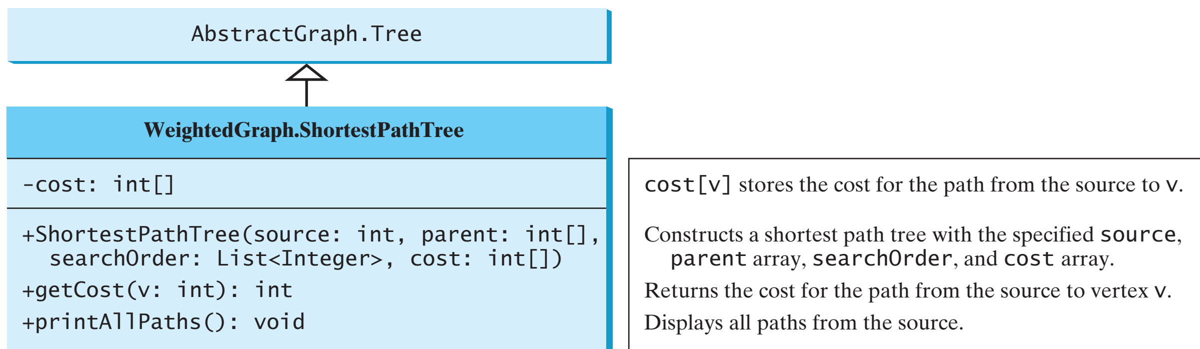


Figure 31.18 `WeightedGraph.ShortestPathTree` extends `AbstractGraph.Tree`.

stores the vertices whose paths have been found (line 195). `vertices` is defined as a protected data field in the `AbstractGraph` class, and it is an array that stores all vertices in the graph. `vertices.size()` returns the number of the vertices in the graph (line 200).

Each vertex is assigned a cost. The cost of the source vertex is `0` (line 211). The cost of all other vertices is initially assigned as infinity (line 209).

The method needs to remove the elements from the queues in order to find the one with the smallest total cost. To keep the original queues intact, queues are cloned in line 214.

A vertex is added to `T` if it is adjacent to one of the vertices in `T` with the smallest cost (line 240). Such a vertex is found using the following procedure:

1. For each vertex `u` in `T`, find its incident edge `e` with the smallest weight to `u`. All the incident edges to `u` are stored in `queues.get(u)`. `queues.get(u).peek()` (line 231) returns the incident edge with the smallest weight. If `e.v` is already in `T`, remove `e` from `queues.get(u)` (line 223). After lines 221–229, `queues.get(u).peek()` returns the edge `e`, such that `e` has the smallest weight to `u` and `e.v` is not in `T` (line 231).

2. Compare all these edges and find the one with the smallest value on `cost[u] + e.getWeight()` (line 232).

After a new vertex is added to `T` (line 240), the cost of this vertex is updated (line 241). Once all vertices are added to `T`, an instance of `ShortestPathTree` is created (line 245). Note that the method will not work if the graph is not connected. However, you can modify it to obtain the shortest paths to all connected vertices to the source vertex.

ShortestPathTree class

The **ShortestPathTree** class extends the **Tree** class (line 249). To create an instance of **ShortestPathTree**, pass **sourceVertex**, **parent**, **T**, and **cost** (lines 253). **sourceVertex** becomes the root in the tree. The data fields **root**, **parent**, and **searchOrder** are defined in the **Tree** class, which is an inner class defined in **AbstractGraph**.

Dijkstra's algorithm time complexity

Dijkstra's algorithm is implemented essentially in the same way as Prim's. Therefore, the time complexity for Dijkstra's algorithm is $O(|E| \log |V|)$, where $|E|$ denotes the number of edges and $|V|$ the number of vertices.

greedy and dynamic programming

Dijkstra's algorithm is a combination of a greedy algorithm and dynamic programming. It is a greedy algorithm in the sense that it always adds a new vertex that has the shortest distance to the source. It stores the shortest distance of each known vertex to the source and uses it later to avoid redundant computing, so Dijkstra's algorithm also uses dynamic programming.

shortest path animation on Companion Website

Pedagogical Note

Go to www.cs.armstrong.edu/liang/animation/ShortestPathAnimation.html to use a GUI interactive program to find the shortest path between any two cities, as shown in Figure 31.19.

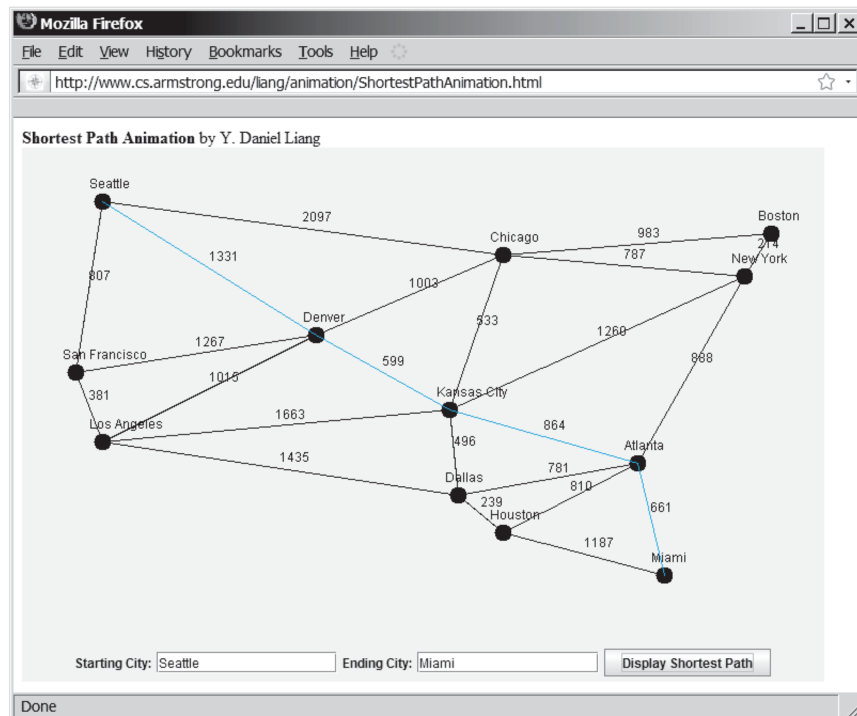


FIGURE 31.19 The animation tool displays a shortest path between two cities.

Listing 31.7 gives a test program that displays the shortest paths from Chicago to all other cities in Figure 31.1 and the shortest paths from vertex **3** to all vertices for the graph in Figure 31.3a, respectively.

LISTING 31.7 TestShortestPath.java

vertices

```
1 public class TestShortestPath {
2   public static void main(String[] args) {
3     String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
```

```
 4        "Denver", "Kansas City", "Chicago", "Boston", "New York",
 5        "Atlanta", "Miami", "Dallas", "Houston"};
 6
 7      int[][] edges = {
 8        {0, 1, 807}, {0, 3, 1331}, {0, 5, 2097},
 9        {1, 0, 807}, {1, 2, 381}, {1, 3, 1267},
10        {2, 1, 381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},
11        {3, 0, 1331}, {3, 1, 1267}, {3, 2, 1015}, {3, 4, 599},
12          {3, 5, 1003},
13        {4, 2, 1663}, {4, 3, 599}, {4, 5, 533}, {4, 7, 1260},
14          {4, 8, 864}, {4, 10, 496},
15        {5, 0, 2097}, {5, 3, 1003}, {5, 4, 533},
16          {5, 6, 983}, {5, 7, 787},
17        {6, 5, 983}, {6, 7, 214},
18        {7, 4, 1260}, {7, 5, 787}, {7, 6, 214}, {7, 8, 888},
19        {8, 4, 864}, {8, 7, 888}, {8, 9, 661},
20          {8, 10, 781}, {8, 11, 810},
21        {9, 8, 661}, {9, 11, 1187},
22        {10, 2, 1435}, {10, 4, 496}, {10, 8, 781}, {10, 11, 239},
23        {11, 8, 810}, {11, 9, 1187}, {11, 10, 239}
24      };
25
26      WeightedGraph<String> graph1 =
27        new WeightedGraph<String>(edges, vertices);
28      WeightedGraph<String>.ShortestPathTree tree1 =
29        graph1.getShortestPath(graph1.getIndex("Chicago"));
30      tree1.printAllPaths();
31
32      // Display shortest paths from Houston to Chicago
33      System.out.print("Shortest path from Houston to Chicago: ");
34      java.util.List<String> path = tree1.getPath(11);
35      for (String s: path) {
36        System.out.print(s + " ");
37      }
38
39      edges = new int[][]{
40        {0, 1, 2}, {0, 3, 8},
41        {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
42        {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
43        {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
44        {4, 2, 5}, {4, 3, 6}
45      };
46      WeightedGraph<Integer> graph2 =
47        new WeightedGraph<Integer>(edges, 5);
48      WeightedGraph<Integer>.ShortestPathTree tree2 =
49        graph2.getShortestPath(3);
50      tree2.printAllPaths();
51    }
52  }
```

edges

create graph1

shortest path

create edges

create graph2

print paths

```
All shortest paths from Chicago are:
A path from Chicago to Seattle: Chicago Seattle (cost: 2097)
A path from Chicago to San Francisco:
  Chicago Denver San Francisco (cost: 2270)
A path from Chicago to Los Angeles:
  Chicago Denver Los Angeles (cost: 2018)
A path from Chicago to Denver: Chicago Denver (cost: 1003)
A path from Chicago to Kansas City: Chicago Kansas City (cost: 533)
```

```
A path from Chicago to Chicago: Chicago (cost: 0)
A path from Chicago to Boston: Chicago Boston (cost: 983)
A path from Chicago to New York: Chicago New York (cost: 787)
A path from Chicago to Atlanta:
  Chicago Kansas City Atlanta (cost: 1397)
A path from Chicago to Miami:
  Chicago Kansas City Atlanta Miami (cost: 2058)
A path from Chicago to Dallas:
  Chicago Kansas City Dallas (cost: 1029)
A path from Chicago to Houston:
  Chicago Kansas City Dallas Houston (cost: 1268)

Shortest path from Chicago to Houston:
  Chicago Kansas City Dallas Houston

All shortest paths from 3 are:
A path from 3 to 0: 3 1 0 (cost: 5)
A path from 3 to 1: 3 1 (cost: 3)
A path from 3 to 2: 3 2 (cost: 4)
A path from 3 to 3: 3 (cost: 0)
A path from 3 to 4: 3 4 (cost: 6)
```

The program creates a weighted graph for Figure 31.1 in line 27. It then invokes the **getShortestPath(graph1.getIndex("Chicago"))** method to return a **Path** object that contains all shortest paths from Chicago. Invoking **printAllPaths()** on the **ShortestPathTree** object displays all the paths (line 30).

The graphical illustration of all shortest paths from Chicago is shown in Figure 31.20. The shortest paths from Chicago to the cities are found in this order: Kansas City, New York,

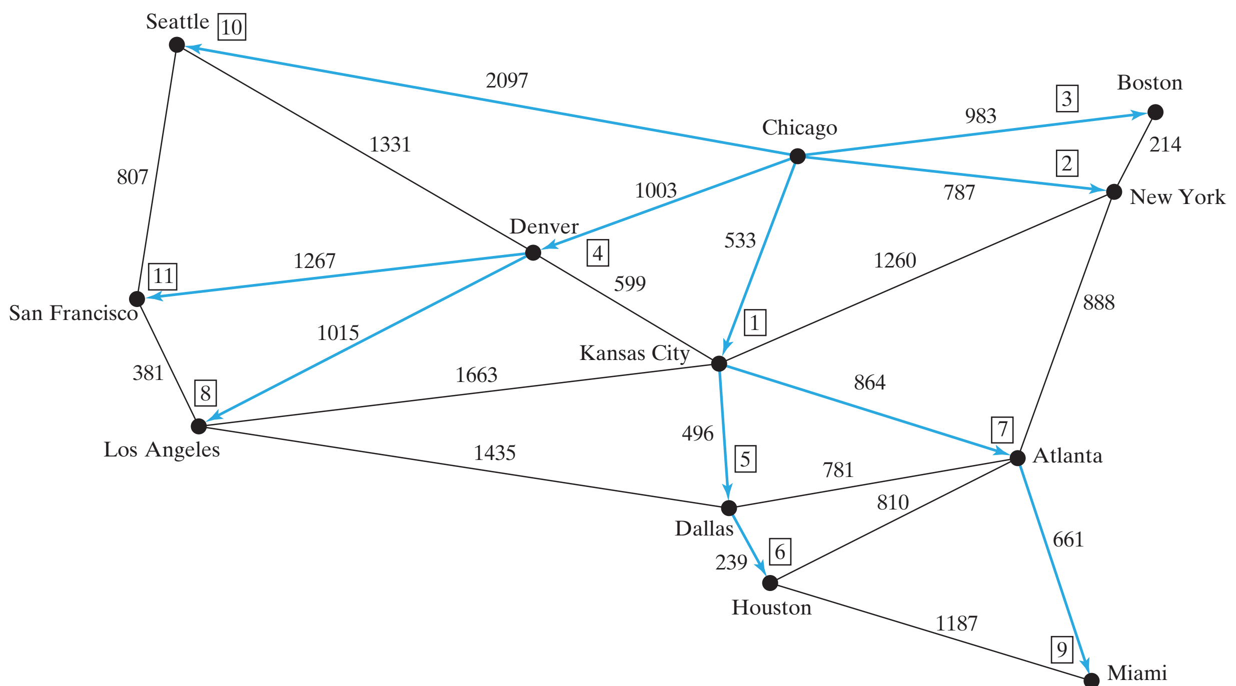


FIGURE 31.20 The shortest paths from Chicago to all other cities are highlighted.

Boston, Denver, Dallas, Houston, Atlanta, Los Angeles, Miami, Seattle, and San Francisco.

MyProgrammingLab™

31.9 Trace Dijkstra's algorithm for finding shortest paths from Boston to all other cities in Figure 31.1.

31.10 Is the shortest path between two vertices unique if all edges have different weights?

31.11 If you use an adjacency matrix to represent weighted edges, what would be the time complexity for Dijkstra's algorithm?

31.12 What happens to the `getShortestPath()` method in `WeightedGraph` if the graph is not connected? Verify your answer by writing a test program that creates an unconnected graph and invoke the `getShortestPath()` method.

31.6 Case Study: The Weighted Nine Tails Problem

The weighted nine tails problem can be reduced to the weighted shortest path problem.

Section 30.10 presented the nine tails problem and solved it using the BFS algorithm. This section presents a variation of the problem and solves it using the shortest-path algorithm.

The nine tails problem is to find the minimum number of the moves that lead to all coins facing down. Each move flips a head coin and its neighbors. The weighted nine tails problem assigns the number of flips as a weight on each move. For example, you can move from the coins in Figure 31.21a to those in Figure 31.21b by flipping the first coin in the first row and its two neighbors. Thus, the weight for this move is `3`.

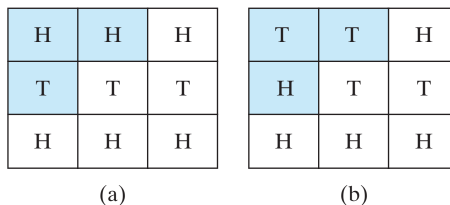


FIGURE 31.21 The weight for each move is the number of flips for the move.

The weighted nine tails problem can be reduced to finding the shortest path from a starting node to the target node in an edge-weighted graph. The graph has 512 nodes. Create an edge from node `v` to `u` if there is a move from node `u` to node `v`. Assign the number of flips to be the weight of the edge.

Recall that in Section 30.10 we defined a class `NineTailModel` for modeling the nine tails problem. We now define a new class named `WeightedNineTailModel` that extends `NineTailModel`, as shown in Figure 31.22.

The `NineTailModel` class creates a `Graph` and obtains a `Tree` rooted at the target node `511`. `WeightedNineTailModel` is the same as `NineTailModel` except that it creates a `WeightedGraph` and obtains a `ShortestPathTree` rooted at the target node `511`. `WeightedNineTailModel` extends `NineTailModel`. The method `getEdges()` finds all edges in the graph. The `getNumberOfFlips(int u, int v)` method returns the number of flips from node `u` to node `v`. The `getNumberOfFlips(int u)` method returns the number of flips from node `u` to the target node.

Listing 31.8 implements `WeightedNineTailModel`.

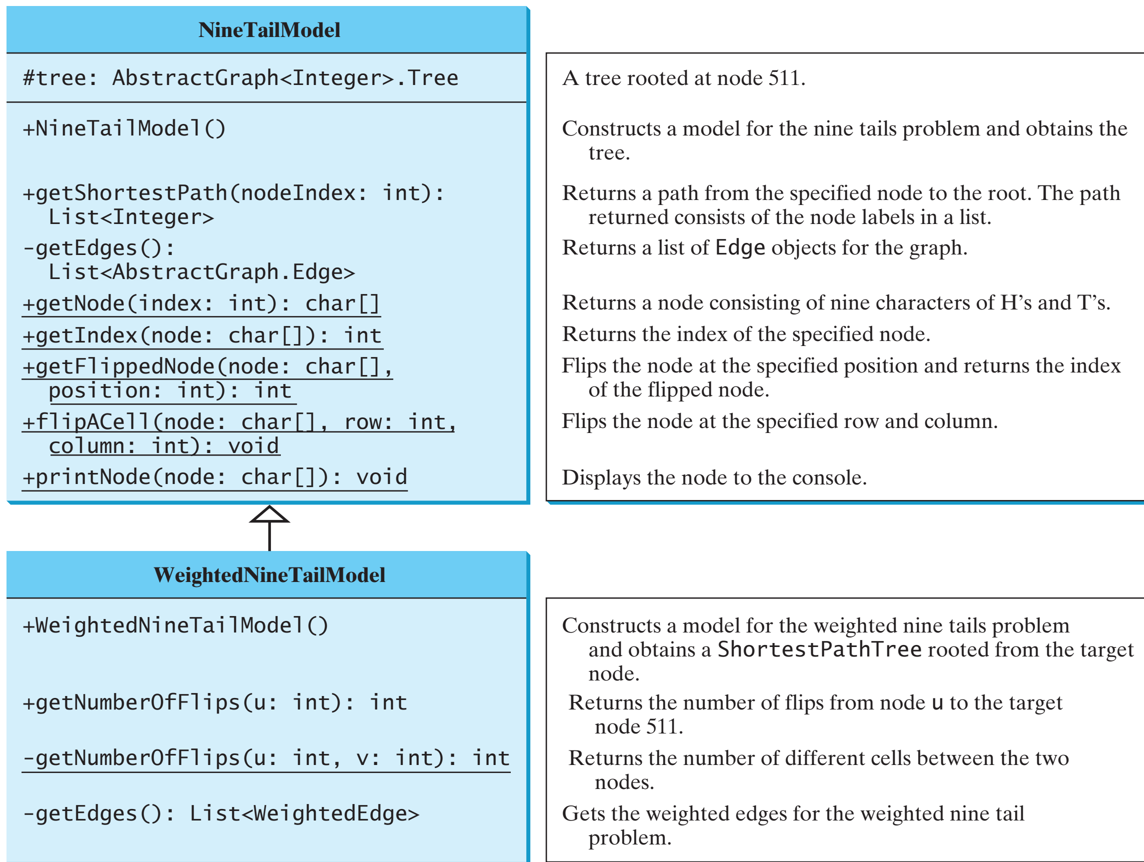


FIGURE 31.22 The `WeightedNineTailModel` class extends `NineTailModel`.

LISTING 31.8 WeightedNineTailModel.java

```
 1  import java.util.*;
 2
 3  public class WeightedNineTailModel extends NineTailModel {
 4    /** Construct a model */
 5    public WeightedNineTailModel() {
 6      // Create edges
 7      List<WeightedEdge> edges = getEdges();
 8
 9      // Create a graph
10      WeightedGraph<Integer> graph = new WeightedGraph<Integer>(
11        edges, NUMBER_OF_NODES);
12
13      // Obtain a shortest path tree rooted at the target node
14      tree = graph.getShortestPath(511);
15    }
16
17    /** Create all edges for the graph */
18    private List<WeightedEdge> getEdges() {
19      // Store edges
20      List<WeightedEdge> edges = new ArrayList<WeightedEdge>();
21
22      for (int u = 0; u < NUMBER_OF_NODES; u++) {
```

extends NineTailModel (line 3)
constructor (line 5)
get edges (line 7)
create a graph (line 10)
get a tree (line 14)
get weighted edges (line 18)

```
23         for (int k = 0; k < 9; k++) {
24           char[] node = getNode(u);  // Get the node for vertex u
25           if (node[k] == 'H') {
26             int v = getFlippedNode(node, k);
27             int numberOfFlips = getNumberOfFlips(u, v);
28
29             // Add edge (v, u) for a legal move from node u to node v
30             edges.add(new WeightedEdge(v, u, numberOfFlips));
31           }
32         }
33       }
34
35       return edges;
36     }
37
38     private static int getNumberOfFlips(int u, int v) {
39       char[] node1 = getNode(u);
40       char[] node2 = getNode(v);
41
42       int count = 0;  // Count the number of different cells
43       for (int i = 0; i < node1.length; i++)
44         if (node1[i] != node2[i]) count++;
45
46       return count;
47     }
48
49     public int getNumberOfFlips(int u) {
50       return (int)((WeightedGraph<Integer>.ShortestPathTree)tree)
51         .getCost(u);
52     }
53   }
```

get adjacent node

weight

add an edge

number of flips

total number of flips

WeightedNineTailModel extends **NineTailModel** to build a **WeightedGraph** to model the weighted nine tails problem (lines 10–11). For each node **u**, the **getEdges()** method finds a flipped node **v** and assigns the number of flips as the weight for edge (**v**, **u**) (line 30). The **getNumberOfFlips(int u, int v)** method returns the number of flips from node **u** to node **v** (lines 38–47). The number of flips is the number of the different cells between the two nodes (line 44).

The **WeightedNineTailModel** obtains a **ShortestPathTree** rooted at the target node **511** (line 14). Note that **tree** is a protected data field defined in **NineTailModel** and **ShortestPathTree** is a subclass of **Tree**. The methods defined in **NineTailModel** use the **tree** property.

The **getNumberOfFlips(int u)** method (lines 49–52) returns the number of flips from node **u** to the target node, which is the cost of the path from node **u** to the target node. This cost can be obtained by invoking the **getCost(u)** method defined in the **ShortestPathTree** class (line 51).

Listing 31.9 gives a program that prompts the user to enter an initial node and displays the minimum number of flips to reach the target node.

LISTING 31.9 WeightedNineTail.java

```
1  import java.util.Scanner;
2
3  public class WeightedNineTail {
4    public static void main(String[] args) {
5      // Prompt the user to enter the nine coins' Hs and Ts
```

```
 6     System.out.print("Enter an initial nine coins' Hs and Ts: ");
 7     Scanner input = new Scanner(System.in);
 8     String s = input.nextLine();
 9     char[] initialNode = s.toCharArray();                          // initial node
10
11     WeightedNineTailModel model = new WeightedNineTailModel();     // create model
12     java.util.List<Integer> path =
13       model.getShortestPath(NineTailModel.getIndex(initialNode));  // get shortest path
14
15     System.out.println("The steps to flip the coins are ");
16     for (int i = 0; i < path.size(); i++)
17       NineTailModel.printNode(                                     // print node
18         NineTailModel.getNode(path.get(i).intValue()));
19
20     System.out.println("The number of flips is " +
21       model.getNumberOfFlips(NineTailModel.getIndex(initialNode))); // number of flips
22   }
23 }
```

```
Enter an initial nine coins Hs and Ts: HHHTTTHHH [Enter]

The steps to flip the coins are
HHH
TTT
HHH

HHH
THT
TTT

TTT
TTT
TTT

The number of flips is 8
```

The program prompts the user to enter an initial node with nine letters with a combination of `H`s and `T`s as a string in line 8, obtains an array of characters from the string (line 9), creates a model (line 11), obtains the shortest path from the initial node to the target node (lines 12–13), displays the nodes in the path (lines 16–18), and invokes `getNumberOfFlips` to get the number of flips needed to reach the target node (line 21).

Check Point

31.13 Why is the `tree` data field in `NineTailModel` in Section 30.13 defined protected?

31.14 How are the nodes created for the graph in `WeightedNineTailModel`?

31.15 How are the edges created for the graph in `WeightedNineTailModel`?

KEY TERMS

Dijkstra's algorithm 1111
edge-weighted graph 1095
minimum spanning tree 1105
Prim's algorithm 1106
shortest path 1111
single-source shortest path 1111
vertex-weighted graph 1095

Chapter Summary

1. You can use adjacency matrices or priority queues to represent weighted edges in graphs.
2. A spanning tree of a graph is a subgraph that is a tree and connects all vertices in the graph. You learned how to implement *Prim's algorithm* for finding a *minimum spanning tree*.
3. You learned how to implement *Dijkstra's algorithm* for finding *shortest paths*.

Test Questions

Do the test questions for this chapter online at www.cs.armstrong.edu/liang/intro9e/test.html.

Programming Exercises

MyProgrammingLab™

***31.1** (*Kruskal's algorithm*) The text introduced Prim's algorithm for finding a minimum spanning tree. Kruskal's algorithm is another well-known algorithm for finding a minimum spanning tree. The algorithm repeatedly finds a minimum-weight edge and adds it to the tree if it does not cause a cycle. The process ends when all vertices are in the tree. Design and implement an algorithm for finding an MST using Kruskal's algorithm.

***31.2** (*Implement Prim's algorithm using an adjacency matrix*) The text implements Prim's algorithm using priority queues on adjacent edges. Implement the algorithm using an adjacency matrix for weighted graphs.

***31.3** (*Implement Dijkstra's algorithm using an adjacency matrix*) The text implements Dijkstra's algorithm using priority queues on adjacent edges. Implement the algorithm using an adjacency matrix for weighted graphs.

***31.4** (*Modify weight in the nine tails problem*) In the text, we assign the number of the flips as the weight for each move. Assuming that the weight is three times of the number of flips, revise the program.

***31.5** (*Prove or disprove*) The conjecture is that both `NineTailModel` and `WeightedNineTailModel` result in the same shortest path. Write a program to prove or disprove it. (*Hint*: Let `tree1` and `tree2` denote the trees rooted at node `511` obtained from `NineTailModel` and `WeightedNineTailModel`, respectively. If the depth of a node `u` is the same in `tree1` and in `tree2`, the length of the path from `u` to the target is the same.)

****31.6** (*Weighted 4 × 4 16 tails model*) The weighted nine tails problem in the text uses a 3 × 3 matrix. Assume that you have 16 coins placed in a 4 × 4 matrix. Create a new model class named `WeightedTailModel16`. Create an instance of the model and save the object into a file named **WeightedTailModel16.dat**.

****31.7** (*Weighted 4 × 4 16 tails view*) Listing 31.9, WeightedNineTail.java, presents a view for the nine tails problem. Revise this program for the weighted 4 × 4 16 tails problem. Your program should read the model object created from the preceding exercise.

****31.8** (*Traveling salesperson problem*) The traveling salesperson problem (TSP) is to find the shortest round-trip route that visits each city exactly once and then returns to the starting city. The problem is equivalent to finding the shortest Hamiltonian

cycle in Programming Exercise 30.17. Add the following method in the **WeightedGraph** class:

```
// Return the shortest cycle
// Return null if no such cycle exists
public List<Integer> getShortestHamiltonianCycle()
```

***31.9** (*Find a minimum spanning tree*) Write a program that reads a connected graph from a file and displays its minimum spanning tree. The first line in the file contains a number that indicates the number of vertices (**n**). The vertices are labeled as **0**, **1**, ..., **n-1**. Each subsequent line describes the edges in the form of **u1, v1, w1 | u2, v2, w2 | ...**. Each triplet in this form describes an edge and its weight. Figure 31.23 shows an example of the file for the corresponding graph. Note that we assume the graph is undirected. If the graph has an edge (**u**, **v**), it also has an edge (**v**, **u**). Only one edge is represented in the file. When you construct a graph, both edges need to be considered.

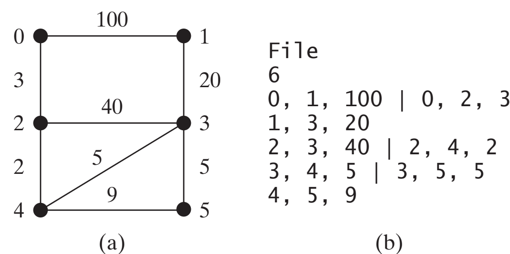


FIGURE 31.23 The vertices and edges of a weighted graph can be stored in a file.

Your program should prompt the user to enter the name of the file, read data from the file, create an instance **g** of **WeightedGraph**, invoke **g.printWeightedEdges()** to display all edges, invoke **getMinimumSpanningTree()** to obtain an instance **tree** of **WeightedGraph.MST**, invoke **tree.getTotalWeight()** to display the weight of the minimum spanning tree, and invoke **tree.printTree()** to display the tree. Here is a sample run of the program:

```
Enter a file name: c:\exercise\WeightedGraphSample.txt [Enter]
The number of vertices is 6
Vertex 0: (0, 2, 3) (0, 1, 100)
Vertex 1: (1, 3, 20) (1, 0, 100)
Vertex 2: (2, 4, 2) (2, 3, 40) (2, 0, 3)
Vertex 3: (3, 4, 5) (3, 5, 5) (3, 1, 20) (3, 2, 40)
Vertex 4: (4, 2, 2) (4, 3, 5) (4, 5, 9)
Vertex 5: (5, 3, 5) (5, 4, 9)
Total weight is 35
Root is: 0
Edges: (3, 1) (0, 2) (4, 3) (2, 4) (3, 5)
```

(*Hint*: Use **new WeightedGraph(list, numberOfVertices)** to create a graph, where **list** contains a list of **WeightedEdge** objects. Use **new WeightedEdge(u, v, w)** to create an edge. Read the first line to get the number

of vertices. Read each subsequent line into a string **s** and use **s.split("[\\|]")** to extract the triplets. For each triplet, use **triplet.split("[,]")** to extract vertices and weight.)

***31.10** (*Create a file for a graph*) Modify Listing 31.3, TestWeightedGraph.java, to create a file for representing **graph1**. The file format is described in Programming Exercise 31.9. Create the file from the array defined in lines 7–24 in Listing 31.3. The number of vertices for the graph is **12**, which will be stored in the first line of the file. An edge (**u**, **v**) is stored if **u < v**. The contents of the file should be as follows:

```
12
0, 1, 807 | 0, 3, 1331 | 0, 5, 2097
1, 2, 381 | 1, 3, 1267
2, 3, 1015 | 2, 4, 1663 | 2, 10, 1435
3, 4, 599 | 3, 5, 1003
4, 5, 533 | 4, 7, 1260 | 4, 8, 864 | 4, 10, 496
5, 6, 983 | 5, 7, 787
6, 7, 214
7, 8, 888
8, 9, 661 | 8, 10, 781 | 8, 11, 810
9, 11, 1187
10, 11, 239
```

***31.11** (*Find shortest paths*) Write a program that reads a connected graph from a file. The graph is stored in a file using the same format specified in Programming Exercise 31.9. Your program should prompt the user to enter the name of the file then two vertices, and should display the shortest path between the two vertices. For example, for the graph in Figure 31.23, the shortest path between **0** and **1** can be displayed as **0 2 4 3 1**.

Here is a sample run of the program:

```
Enter a file name: WeightedGraphSample2.txt ↵Enter
Enter two vertices (integer indexes): 0 1 ↵Enter
The number of vertices is 6
Vertex 0: (0, 2, 3) (0, 1, 100)
Vertex 1: (1, 3, 20) (1, 0, 100)
Vertex 2: (2, 4, 2) (2, 3, 40) (2, 0, 3)
Vertex 3: (3, 4, 5) (3, 5, 5) (3, 1, 20) (3, 2, 40)
Vertex 4: (4, 2, 2) (4, 3, 5) (4, 5, 9)
Vertex 5: (5, 3, 5) (5, 4, 9)
A path from 0 to 1: 0 2 4 3 1
```

***31.12** (*Display weighted graphs*) Revise **GraphView** in Listing 30.6 to display a weighted graph. Write a program that displays the graph in Figure 31.1 as shown in Figure 31.24a.

***31.13** (*Display shortest paths*) Revise **GraphView** in Listing 30.6 to display a weighted graph and the shortest path between the two specified cities, as shown in Figure 31.19. You need to add a data field **path** in **GraphView**. If a **path** is not null, the edges in the path are displayed in red. If a city not in the map is entered, the program displays a dialog box to alert the user.

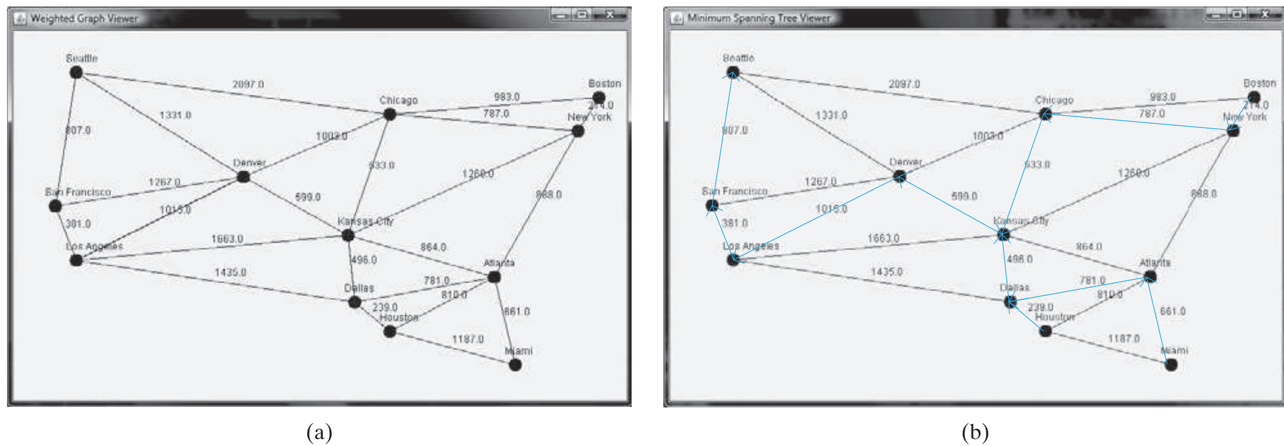


FIGURE 31.24 (a) Exercise 31.12 displays a weighted graph. (b) Exercise 31.14 displays an MST.

***31.14** (*Display a minimum spanning tree*) Revise `GraphView` in Listing 30.6 to display a weighted graph and a minimum spanning tree for the graph in Figure 31.1, as shown in Figure 31.24b. The edges in the MST are shown in red.

*****31.15** (*Dynamic graphs*) Write a program that lets the users create a weighted graph dynamically. The user can create a vertex by entering its name and location, as shown in Figure 31.25. The user can also create an edge to connect two vertices. To simplify the program, assume that vertex names are the same as vertex indices. You have to add the vertex indices `0`, `1`, . . . , and `n`, in this order. The user can specify two vertices and let the program display their shortest path in red.

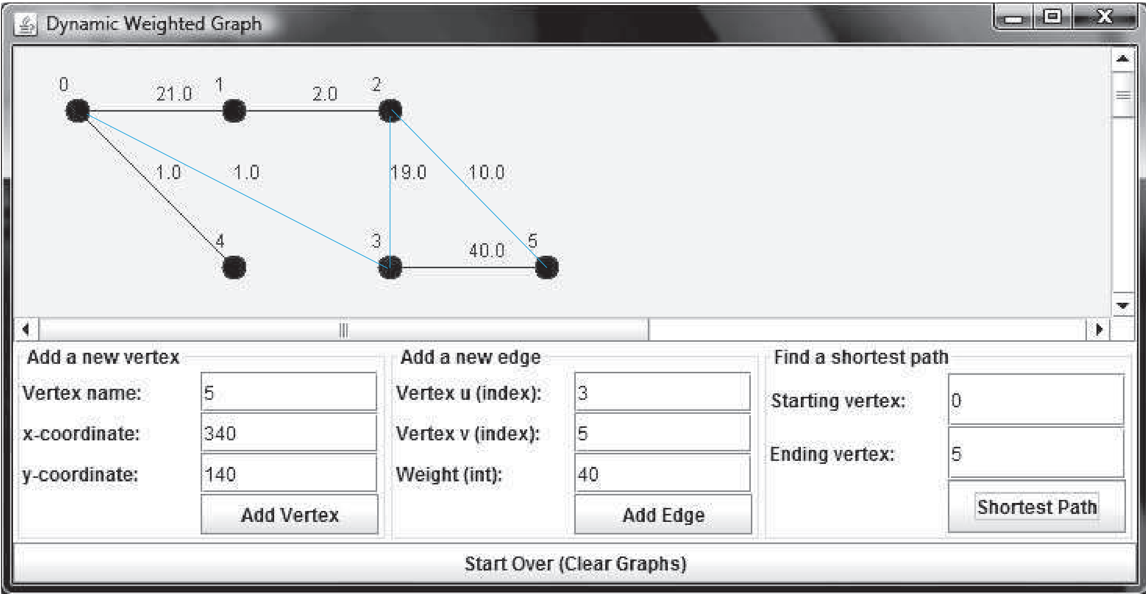


FIGURE 31.25 The program can add vertices and edges and display the shortest path between two specified vertices.

*****31.16** (*Display a dynamic MST*) Write a program that lets the user create a weighted graph dynamically. The user can create a vertex by entering its name and location, as shown in Figure 31.26. The user can also create an edge to connect two vertices. To simplify the program, assume that vertex names are the same as those of vertex indices. You have to add the vertex indices **0**, **1**, . . ., and **n**, in this order. The edges in the MST are displayed in red. As new edges are added, the MST is redisplayed.

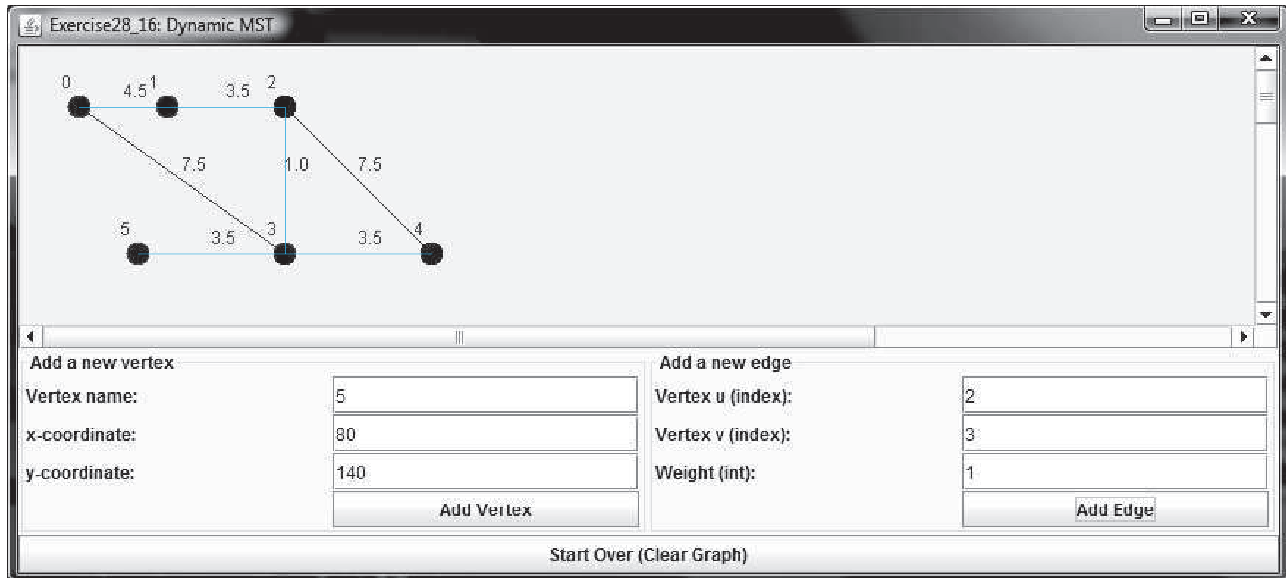


FIGURE 31.26 The program can add vertices and edges and display MST dynamically.

*****31.17** (*Weighted graph visualization tool*) Develop an applet as shown in Figure 31.2, with the following requirements: (1) The radius of each vertex is 20 pixels. (2) The user clicks the left mouse button to place a vertex centered at the mouse point, provided that the mouse point is not inside or too close to an existing vertex. (3) The user clicks the right mouse button inside an existing vertex to remove the vertex. (4) The user presses a mouse button inside a vertex and drags to another vertex and then releases the button to create an edge, and the distance between the two vertices is also displayed. (5) The user drags a vertex while pressing the *CTRL* key to move a vertex. (6) The vertices are numbers starting from **0**. When a vertex is removed, the vertices are renumbered. (7) You can click the *Show MST* or *Show All SP From the Source* button to display an MST or SP tree from a starting vertex. (8) You can click the *Show Shortest Path* button to display the shortest path between the two specified vertices.