

CHAPTER

29

WEIGHTED GRAPHS AND APPLICATIONS

Objectives

- To represent weighted edges using adjacency matrices and adjacency lists (§29.2).
- To model weighted graphs using the **WeightedGraph** class that extends the **UnweightedGraph** class (§29.3).
- To design and implement the algorithm for finding a minimum spanning tree (§29.4).
- To define the **MST** class that extends the **SearchTree** class (§29.4).
- To design and implement the algorithm for finding single-source shortest paths (§29.5).
- To define the **ShortestPathTree** class that extends the **SearchTree** class (§29.5).
- To solve the weighted nine tails problem using the shortest-path algorithm (§29.6).



29.1 Introduction



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

Figure 28.1 assumes the graph represents the number of flights among cities. You can apply the Breadth-First Search (BFS) to find the fewest number of flights between two cities. Assume the edges represent the driving distances among the cities as shown in Figure 29.1. 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

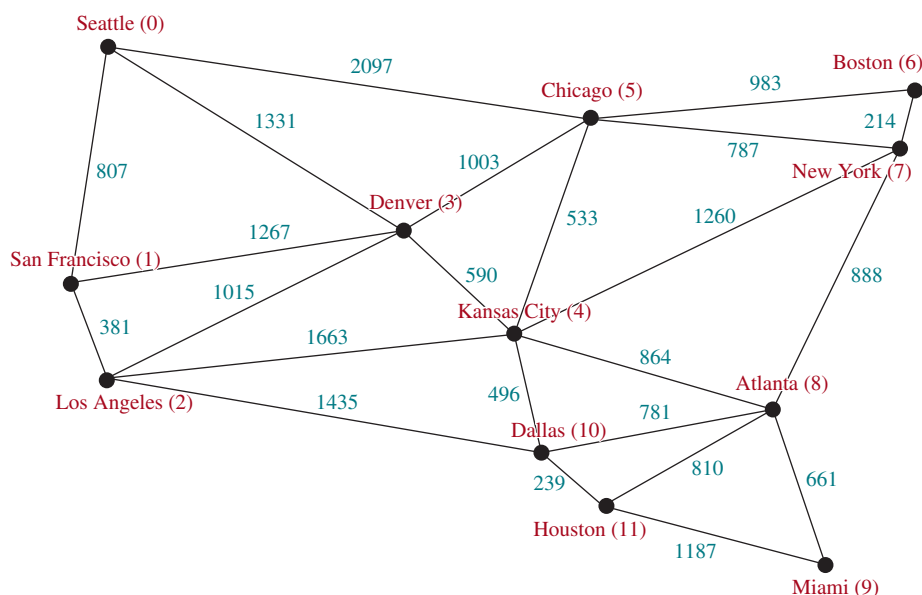


FIGURE 29.1 The graph models the distances among the cities.

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 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 29.4, and the algorithm for finding shortest paths in Section 29.5.



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 liveexample.pearsoncmg.com/dsanimation/WeightedGraphLearningToolBook.html, as shown in Figure 29.2. The tool allows you to enter vertices, create weighted edges, view the graph, and find an MST, all shortest paths from a single source other operations.

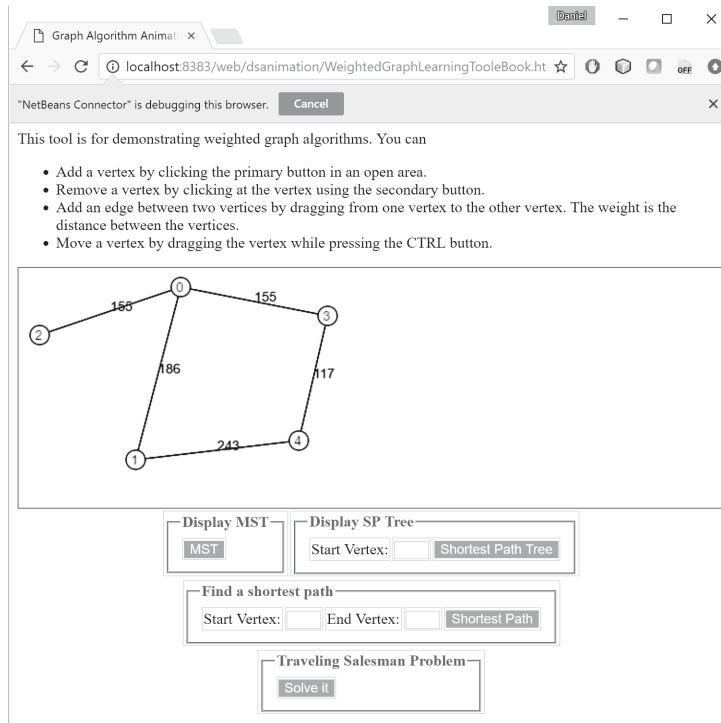


FIGURE 29.2 You can use the tool to create a weighted graph with mouse gestures and show the MST and shortest paths, and perform other operations. *Source:* © Mozilla Firefox.

29.2 Representing Weighted Graphs

Weighted edges can be stored in adjacency lists.

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.

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.

29.2.1 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 29.3a using the array in Figure 29.3b.



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 = {
    {Integer.valueOf(0), Integer.valueOf(1), new SomeTypeForWeight(2)},
    {Integer.valueOf(0), Integer.valueOf(3), new SomeTypeForWeight(8)},
    ...
};
```



vertex-weighted graph
edge-weighted graph

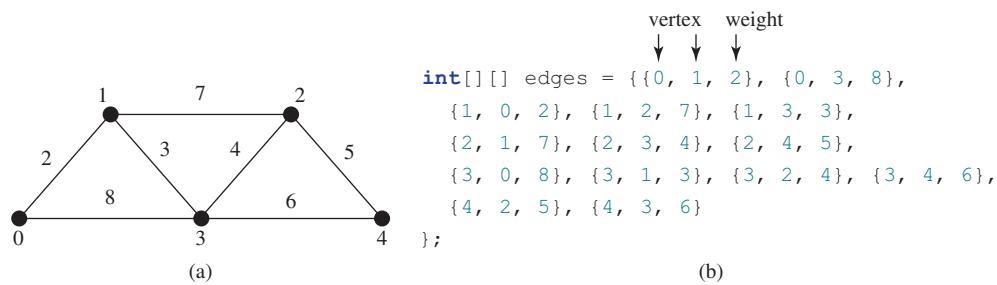


FIGURE 29.3 Each edge is assigned a weight in an edge-weighted graph.

29.2.2 Weighted Adjacency Matrices

Assume 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 29.3a 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

29.2.3 Adjacency Lists

Another way to represent the edges is to define edges as objects. The `Edge` class was defined to represent an unweighted edge in Listing 28.3. For weighted edges, we define the `WeightedEdge` class as shown in Listing 29.1.

LISTING 29.1 WeightedEdge.java

edge weight

constructor

compare edges

```
1 public class WeightedEdge extends 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 }
```

An **Edge** object represents an edge from vertex **u** to **v**. **WeightedEdge** extends **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**). Often you need to compare the weights of the edges. 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, the adjacency lists for the vertices in the graph in Figure 29.3a can be represented as follows:

```
java.util.List<WeightedEdge>[] list = new java.util.List[5];
```

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

list[i] stores all edges adjacent to vertex **i**.

For flexibility, we will use an array list rather than a fixed-sized array to represent **list** as follows:

```
List<List<WeightedEdge>> list = new java.util.ArrayList<>();
```

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



29.2.2 What is the output of the following code?

```
List<WeightedEdge> list = new ArrayList<>();
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);
```

29.3 The **WeightedGraph** Class

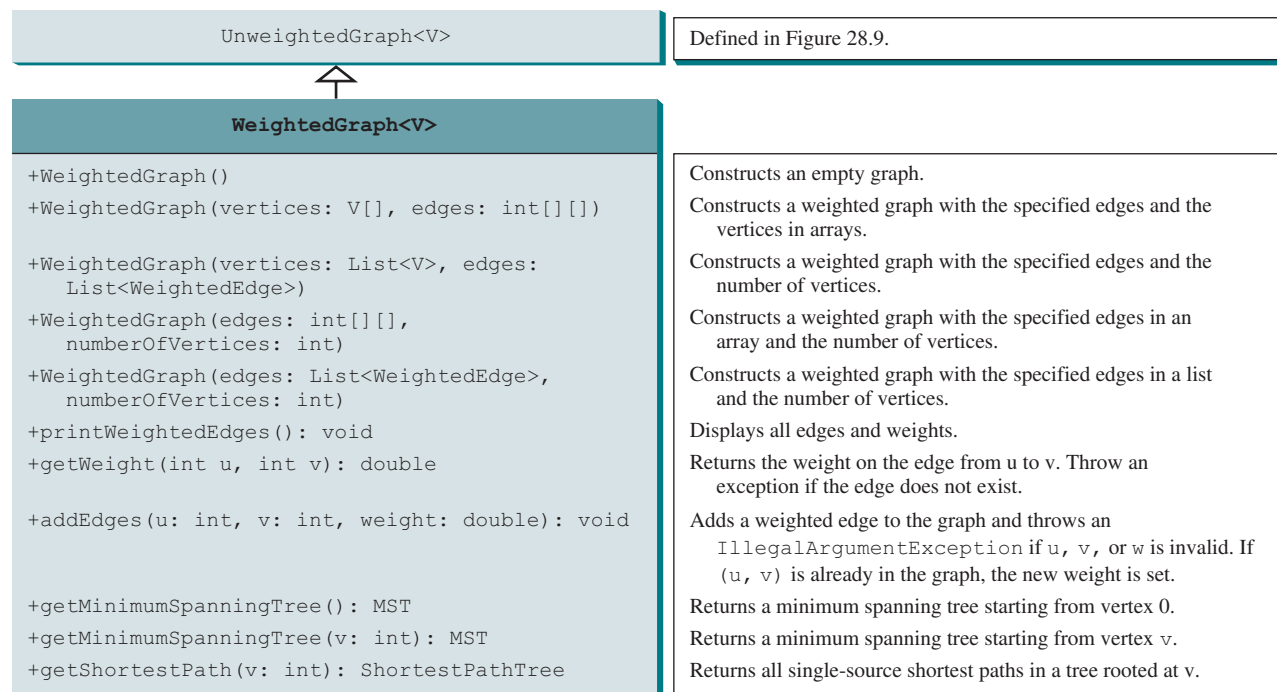
The **WeightedGraph** class extends **UnweightedGraph**.

The preceding chapter designed the **Graph** interface and the **UnweightedGraph** class for modeling graphs. We now design **WeightedGraph** as a subclass of **UnweightedGraph**, as shown in Figure 29.4.

WeightedGraph simply extends **UnweightedGraph** with five constructors for creating concrete **WeightedGraph** instances. **WeightedGraph** inherits all methods from **UnweightedGraph**, 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 29.4 and 29.5, respectively.

Listing 29.2 implements **WeightedGraph**. Edge adjacency lists (lines 38–63) are used internally to store adjacent edges for a vertex. When a **WeightedGraph** is constructed, its edge



FIGURE 29.4 `WeightedGraph` extends `UnweightedGraph`.

adjacency lists are created (lines 47 and 57). The methods `getMinimumSpanningTree()` (lines 99–138) and `getShortestPath()` (lines 156–197) will be introduced in upcoming sections.

LISTING 29.2 `WeightedGraph.java`

```

1  import java.util.*;
2
3  public class WeightedGraph<V> extends UnweightedGraph<V> {
4      /** Construct an empty */
5      public WeightedGraph() {
6      }
7
8      /** Construct a WeightedGraph from vertices and edged in arrays */
9      public WeightedGraph(V[] vertices, int[][] edges) {
10         createWeightedGraph(java.util.Arrays.asList(vertices), edges);
11     }
12
13     /** Construct a WeightedGraph from vertices and edges in list */
14     public WeightedGraph(int[][] edges, int numberOfVertices) {
15         List<V> vertices = new ArrayList<>();
16         for (int i = 0; i < numberOfVertices; i++)
17             vertices.add((V) (Integer.valueOf(i)));
18
19         createWeightedGraph(vertices, edges);
20     }
21

```

no-arg constructor

constructor

constructor

```

22  /** Construct a WeightedGraph for vertices 0, 1, 2 and edge list */
23  public WeightedGraph(List<V> vertices, List<WeightedEdge> edges) {      constructor
24      createWeightedGraph(vertices, edges);
25  }
26
27  /** Construct a WeightedGraph from vertices 0, 1, and edge array */
28  public WeightedGraph(List<WeightedEdge> edges,                          constructor
29      int numberOfVertices) {
30      List<V> vertices = new ArrayList<>();
31      for (int i = 0; i < numberOfVertices; i++)
32          vertices.add((V)(Integer.valueOf(i)));
33
34      createWeightedGraph(vertices, edges);
35  }
36
37  /** Create adjacency lists from edge arrays */
38  private void createWeightedGraph(List<V> vertices, int[][] edges) {
39      this.vertices = vertices;
40
41      for (int i = 0; i < vertices.size(); i++) {
42          neighbors.add(new ArrayList<Edge>()); // Create a list for vertices      create list for vertices
43      }
44
45      for (int i = 0; i < edges.length; i++) {
46          neighbors.get(edges[i][0]).add(
47              new WeightedEdge(edges[i][0], edges[i][1], edges[i][2])); // create a weighted edge
48      }
49  }
50
51  /** Create adjacency lists from edge lists */
52  private void createWeightedGraph(
53      List<V> vertices, List<WeightedEdge> edges) {
54      this.vertices = vertices;
55
56      for (int i = 0; i < vertices.size(); i++) {
57          neighbors.add(new ArrayList<Edge>()); // Create a list for vertices      create list for vertices
58      }
59
60      for (WeightedEdge edge: edges) {
61          neighbors.get(edge.u).add(edge); // Add an edge into the list
62      }
63  }
64
65  /** Return the weight on the edge (u, v) */
66  public double getWeight(int u, int v) throws Exception {              get edge weight
67      for (Edge edge : neighbors.get(u)) {
68          if (edge.v == v) {
69              return ((WeightedEdge)edge).weight;
70          }
71      }
72
73      throw new Exception("Edge does not exist");
74  }
75
76  /** Display edges with weights */
77  public void printWeightedEdges() {                                     print edges
78      for (int i = 0; i < getSize(); i++) {
79          System.out.print(getVertex(i) + " (" + i + "): ");
80          for (Edge edge : neighbors.get(i)) {
81              System.out.print("(" + edge.u +

```

```

82         ", " + edge.v + ", " + ((WeightedEdge)edge).weight + ") ";
83     }
84     System.out.println();
85 }
86 }
87
88 /** Add edges to the weighted graph */
add edge      89 public boolean addEdge(int u, int v, double weight) {
90     return addEdge(new WeightedEdge(u, v, weight));
91 }
92
93 /** Get a minimum spanning tree rooted at vertex 0 */
get an MST    94 public MST getMinimumSpanningTree() {
start from vertex 0 95     return getMinimumSpanningTree(0);
96 }
97
98 /** Get a minimum spanning tree rooted at a specified vertex */
MST from a starting vertex 99 public MST getMinimumSpanningTree(int startingVertex) {
100     // cost[v] stores the cost by adding v to the tree
101     double[] cost = new double[getSize()];
102     for (int i = 0; i < cost.length; i++) {
103         cost[i] = Double.POSITIVE_INFINITY; // Initial cost
104     }
105     cost[startingVertex] = 0; // Cost of source is 0
106
107     int[] parent = new int[getSize()]; // Parent of a vertex
108     parent[startingVertex] = -1; // startingVertex is the root
109     double totalWeight = 0; // Total weight of the tree thus far
110
111     List<Integer> T = new ArrayList<>();
112
113     // Expand T
114     while (T.size() < getSize()) {
115         // Find smallest cost u in V - T
116         int u = -1; // Vertex to be determined
117         double currentMinCost = Double.POSITIVE_INFINITY;
118         for (int i = 0; i < getSize(); i++) {
119             if (!T.contains(i) && cost[i] < currentMinCost) {
120                 currentMinCost = cost[i];
121                 u = i;
122             }
123         }
124
125         if (u == -1) break; else T.add(u); // Add a new vertex to T
126         totalWeight += cost[u]; // Add cost[u] to the tree
127
128         // Adjust cost[v] for v that is adjacent to u and v in V - T
129         for (Edge e: neighbors.get(u)) {
130             if (!T.contains(e.v) && cost[e.v] > ((WeightedEdge)e).weight) {
131                 cost[e.v] = ((WeightedEdge)e).weight;
132                 parent[e.v] = u;
133             }
134         }
135     } // End of while
136
137     return new MST(startingVertex, parent, T, totalWeight);
138 }
139
140 /** MST is an inner class in WeightedGraph */
MST inner class 141 public class MST extends SearchTree {

```



```

142     private double totalWeight; // Total weight of all edges in the tree    total weight in tree
143
144     public MST(int root, int[] parent, List<Integer> searchOrder,
145               double totalWeight) {
146         super(root, parent, searchOrder);
147         this.totalWeight = totalWeight;
148     }
149
150     public double getTotalWeight() {
151         return totalWeight;
152     }
153 }
154
155 /** Find single-source shortest paths */
156 public ShortestPathTree getShortestPath(int sourceVertex) {    getShortestPath
157     // cost[v] stores the cost of the path from v to the source
158     double[] cost = new double[getSize()];                    initialize cost
159     for (int i = 0; i < cost.length; i++) {
160         cost[i] = Double.POSITIVE_INFINITY; // Initial cost set to infinity
161     }
162     cost[sourceVertex] = 0; // Cost of source is 0
163
164     // parent[v] stores the previous vertex of v in the path
165     int[] parent = new int[getSize()];
166     parent[sourceVertex] = -1; // The parent of source is set to -1
167
168     // T stores the vertices whose path found so far
169     List<Integer> T = new ArrayList<>();                        shortest-path tree
170
171     // Expand T
172     while (T.size() < getSize()) {                                expand tree
173         // Find smallest cost u in V - T
174         int u = -1; // Vertex to be determined
175         double currentMinCost = Double.POSITIVE_INFINITY;
176         for (int i = 0; i < getSize(); i++) {
177             if (!T.contains(i) && cost[i] < currentMinCost) {
178                 currentMinCost = cost[i];
179                 u = i;                                            vertex with smallest cost
180             }
181         }
182
183         if (u == -1) break; else T.add(u); // Add a new vertex to T    add to T
184
185         // Adjust cost[v] for v that is adjacent to u and v in V - T
186         for (Edge e: neighbors.get(u)) {
187             if (!T.contains(e.v)
188                 && cost[e.v] > cost[u] + ((WeightedEdge)e).weight) {
189                 cost[e.v] = cost[u] + ((WeightedEdge)e).weight;    adjust cost
190                 parent[e.v] = u;                                     adjust parent
191             }
192         }
193     } // End of while
194
195     // Create a ShortestPathTree
196     return new ShortestPathTree(sourceVertex, parent, T, cost);    create a tree
197 }
198
199 /** ShortestPathTree is an inner class in WeightedGraph */
200 public class ShortestPathTree extends SearchTree {              shortest-path tree
201     private double[] cost; // cost[v] is the cost from v to source    cost

```

```

202
203     /** Construct a path */
204     public ShortestPathTree(int source, int[] parent,
205         List<Integer> searchOrder, double[] cost) {
206         super(source, parent, searchOrder);
207         this.cost = cost;
208     }
209
210     /** Return the cost for a path from the root to vertex v */
211     public double getCost(int v) {
212         return cost[v];
213     }
214
215     /** Print paths from all vertices to the source */
216     public void printAllPaths() {
217         System.out.println("All shortest paths from " +
218             vertices.get(getRoot()) + " are:");
219         for (int i = 0; i < cost.length; i++) {
220             printPath(i); // Print a path from i to the source
221             System.out.println("(cost: " + cost[i] + ")"); // Path cost
222         }
223     }
224 }
225 }

```

constructor

get cost

print all paths

The **WeightedGraph** class extends the **UnweightedGraph** class (line 3). The properties **vertices** and **neighbors** in **UnweightedGraph** are inherited in **WeightedGraph**. **neighbors** is a list. Each element in the list is another list that contains edges. For unweighted graph, each edge is an instance of **Edge**. For a weighted graph, each edge is an instance of **WeightedEdge**. **WeightedEdge** is a subtype of **Edge**. So you can add a weighted edge into **neighbors.get(i)** for a weighted graph (line 47).

The **addEdge(u, v, weight)** method (lines 88–91) adds an edge (**u**, **v**, **weight**) to the graph. If a graph is undirected, you should invoke **addEdge(u, v, weight)** and **addEdge(v, u, weight)** to add an edge between **u** and **v**.

Listing 29.3 gives a test program that creates a graph for the one in Figure 29.1 and another graph for the one in Figure 29.3a.

LISTING 29.3 TestWeightedGraph.java

```

1  public class TestWeightedGraph {
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},

```

vertices

edges

```

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<>(vertices, edges);
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();
36
37     edges = new int[][] {
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 = new WeightedGraph<>(edges, 5);
45     System.out.println("\nThe edges for graph2:");
46     graph2.printWeightedEdges();
47 }
48 }

```

create graph

print edges

edges

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, 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 29.1 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 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 29.3a in lines 37–44. Line 46 invokes the **printWeightedEdges()** method on **graph2** to display all edges in **graph2**.



29.3.1 If a priority queue is used to store weighted edges, what is the output of the following code?

```
PriorityQueue<WeightedEdge> q = new PriorityQueue<>();
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);
```

29.3.2 If a priority queue is used to store weighted edges, what is wrong in the following code? Fix it and show the output.

```
List<PriorityQueue<WeightedEdge>> queues = new ArrayList<>();
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()));
```

29.3.3 Show the output of the following code:

```
public class Test {
    public static void main(String[] args) {
        WeightedGraph<Character> graph = new WeightedGraph<>();
        graph.addVertex('U');
        graph.addVertex('V');
        int indexForU = graph.getIndex('U');
        int indexForV = graph.getIndex('V');
        System.out.println("indexForU is " + indexForU);
        System.out.println("indexForV is " + indexForV);
        graph.addEdge(indexForU, indexForV, 2.5);
        System.out.println("Degree of U is " +
            graph.getDegree(indexForU));
        System.out.println("Degree of V is " +
            graph.getDegree(indexForV));
        System.out.println("Weight of UV is " +
            graph.getWeight(indexForU, indexOfV));
    }
}
```

29.4 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 the edges are weighted. A *minimum spanning tree* has the minimum total weights. For example, the trees in Figures 29.5b, 29.5c, 29.5d are spanning trees for the graph in Figure 29.5a. The trees in Figures 29.5c and 29.5d are minimum spanning trees.

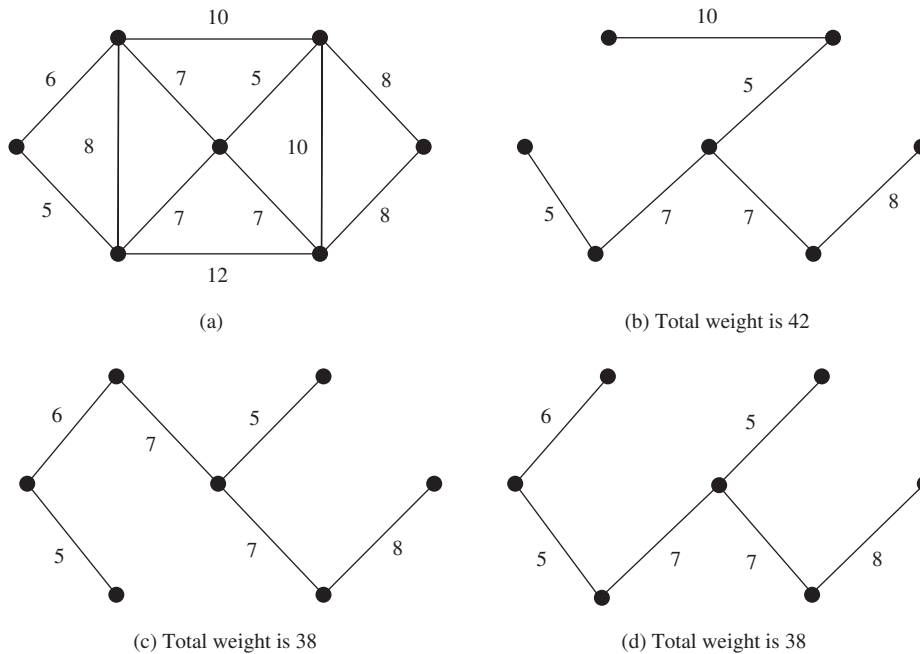


FIGURE 29.5 The trees in (c) and (d) are minimum spanning trees of the graph in (a).

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 rates 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.

29.4.1 Minimum Spanning Tree Algorithms

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 29.4.

Prim's algorithm

LISTING 29.4 Prim's Minimum Spanning Tree Algorithm

Input: A connected undirected weighted $G = (V, E)$ with nonnegative weights
 Output: MST (a minimum spanning tree with vertex s as the root)

```

1  MST getMinimumSpanningTree(s) {
2    Let  $T$  be a set for the vertices in the spanning tree;
```

add initial vertex

more vertices?

find a vertex

add to tree

```
3 Initially, add the starting vertex, s, to T;
4
5 while (size of T < n) {
6     Find x in T and y in V - T with the smallest weight
7     on the edge (x, y), as shown in Figure 29.6;
8     Add y to T and set parent[y] = x;
9 }
10 }
```

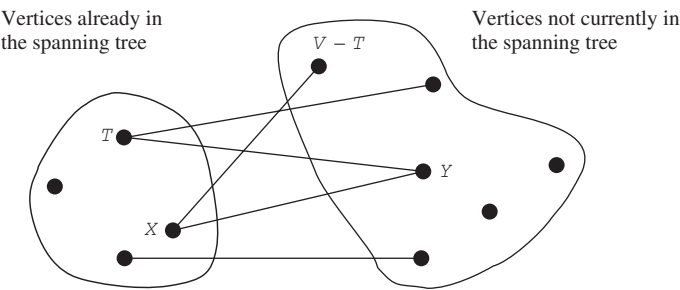


FIGURE 29.6 Find a vertex **x** in **T** that connects a vertex **y** in **V - T** with the smallest weight.

example

The algorithm starts by adding the starting vertex into **T**. It then continuously adds a vertex (say **y**) from **V - T** into **T**. **y** is the vertex that is adjacent to a 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 29.6, and **(x, y)** is the one with the smallest weight. Consider the graph in Figure 29.7. The algorithm adds the vertices to **T** in this order:

1. Add vertex **0** to **T**.
2. Add vertex **5** to **T**, since **WeightedEdge(5, 0, 5)** has the smallest weight among all edges incident to a vertex in **T**, as shown in Figure 29.7a. The arrow line from **0** to **5** indicates that **0** is the parent of **5**.
3. Add vertex **1** to **T**, since **WeightedEdge(1, 0, 6)** has the smallest weight among all edges incident to a vertex in **T**, as shown in Figure 29.7b.
4. Add vertex **6** to **T**, since **WeightedEdge(6, 1, 7)** has the smallest weight among all edges incident to a vertex in **T**, as shown in Figure 29.7c.
5. Add vertex **2** to **T**, since **WeightedEdge(2, 6, 5)** has the smallest weight among all edges incident to a vertex in **T**, as shown in Figure 29.7d.
6. Add vertex **4** to **T**, since **WeightedEdge(4, 6, 7)** has the smallest weight among all edges incident to a vertex in **T**, as shown in Figure 29.7e.
7. Add vertex **3** to **T**, since **WeightedEdge(3, 2, 8)** has the smallest weight among all edges incident to a vertex in **T**, as shown in Figure 29.7f.

unique tree?



Note
A minimum spanning tree is not unique. For example, both (c) and (d) in Figure 29.5 are minimum spanning trees for the graph in Figure 29.5a. However, if the weights are distinct, the graph has a unique minimum spanning tree.

connected and undirected



Note
Assume 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.

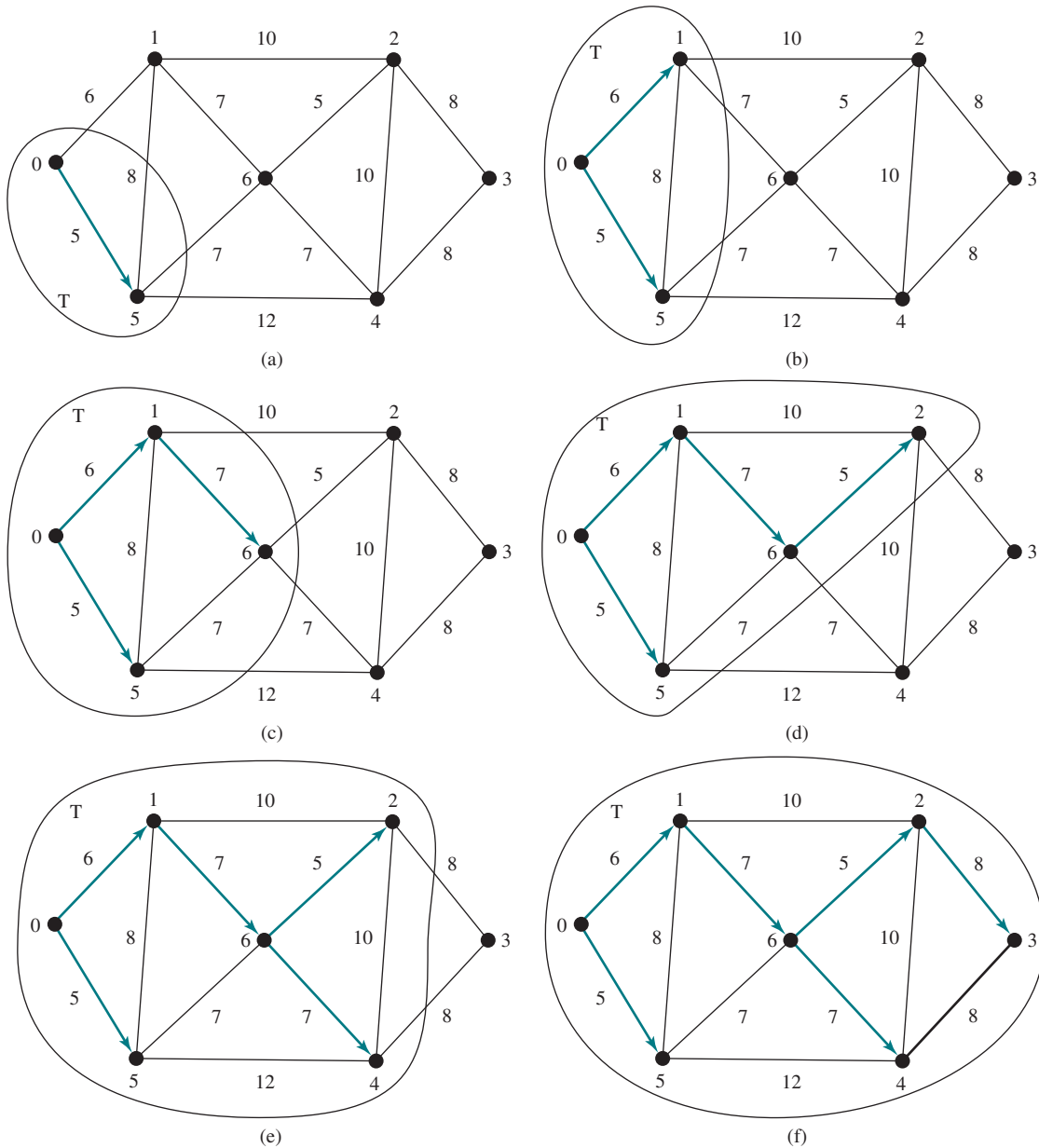


FIGURE 29.7 The adjacent vertices with the smallest weight are added successively to T .

29.4.2 Refining Prim's MST Algorithm

To make it easy to identify the next vertex to add into the tree, we use $\text{cost}[v]$ to store the cost of adding a vertex v to the spanning tree T . Initially, $\text{cost}[s]$ is 0 for a starting vertex and assign infinity to $\text{cost}[v]$ for all other vertices. The algorithm repeatedly finds a vertex u in $V-T$ with the smallest $\text{cost}[u]$ and moves u to T . The refined version of the algorithm is given in Listing 29.5.

LISTING 29.5 Refined Version of Prim's Algorithm

Input: A connected undirected weighted $G = (V, E)$ with nonnegative weights
 Output: a minimum spanning tree with the starting vertex s as the root

1 MST getMinimumSpanngingTree(s) {
2 Let T be a set that contains the vertices in the spanning tree;
3 Initially T is empty;
4 Set cost[s] = 0 and cost[v] = infinity for all other vertices in V;
5
6 while (size of T < n) {
7 Find u not in T with the smallest cost[u];
8 Add u to T;
9 for (each v not in T and (u, v) in E)
10 if (cost[v] > w(u, v)) { // Adjust cost[v]
11 cost[v] = w(u, v); parent[v] = u;
12 }
13 }
14 }

find next vertex
add a vertex to T

adjust cost[v]

For an interactive demo on how the refined Prim’s algorithm works, see liveexample.pearsoncmg.com/dsanimation/RefinedPrim.html.

29.4.3 Implementation of the MST Algorithm

getMinimumSpanningTree()

The `getMinimumSpanningTree(int v)` method is defined in the `WeightedGraph` class, as shown in Figure 29.4. It returns an instance of the `MST` class. The `MST` class is defined as an inner class in the `WeightedGraph` class, which extends the `SearchTree` class, as shown in Figure 29.8. The `SearchTree` class was shown in Figure 28.11. The `MST` class was implemented in lines 141–153 in Listing 29.2.

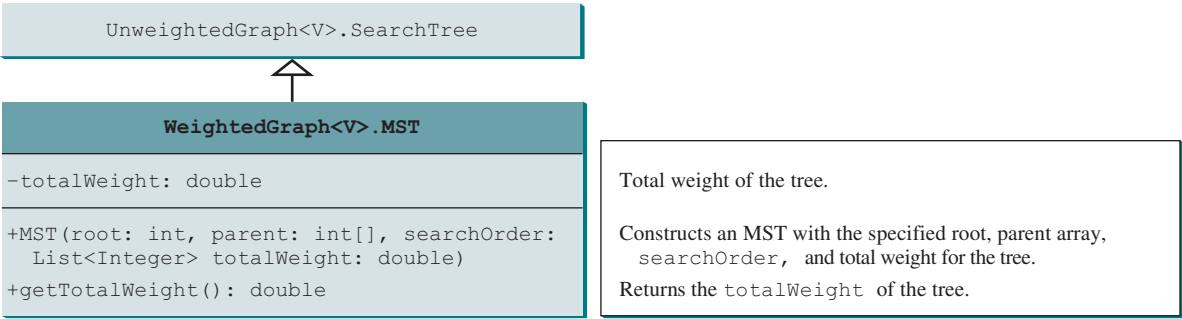


FIGURE 29.8 The `MST` class extends the `SearchTree` class.

The refined version of the Prim’s algorithm greatly simplifies the implementation. The `getMinimumSpanningTree` method was implemented using the refined version of the Prim’s algorithm in lines 99–138 in Listing 29.2. The `getMinimumSpanningTree(int startingVertex)` method sets `cost[startingVertex]` to 0 (line 105) and `cost[v]` to infinity for all other vertices (lines 102–104). The parent of `startingVertex` is set to `-1` (line 108). `T` is a list that stores the vertices added into the spanning tree (line 111). We use a list for `T` rather than a set in order to record the order of the vertices added to `T`. Initially, `T` is empty. To expand `T`, the method performs the following operations:

1. Find the vertex `u` with the smallest `cost[u]` (lines 118–123).
2. If `u` is found, add it to `T` (line 125). Note if `u` is not found (`u == -1`), the graph is not connected. The `break` statement exits the while loop in this case.
3. After adding `u` in `T`, update `cost[v]` and `parent[v]` for each `v` adjacent to `u` in `V-T` if `cost[v] > w(u, v)` (lines 129–134).

After a new vertex is added to `T`, `totalWeight` is updated (line 126). Once all reachable vertices from `s` are added to `T`, an instance of `MST` is created (line 137). Note 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 **SearchTree** class (line 141). To create an instance of **MST**, pass **root**, **parent**, **T**, and **totalWeight** (lines 144–145). The data fields **root**, **parent**, and **searchOrder** are defined in the **SearchTree** class, which is an inner class defined in **UnweightedGraph**.

Note testing whether a vertex **i** is in **T** by invoking **T.contains(i)** takes **O(n)** time, since **T** is a list. Therefore, the overall time complexity for this implementation is $O(n^3)$. Interested readers may see Programming Exercise 29.20 for improving the implementation and reduce the complexity to $O(n^2)$.

time complexity

Listing 29.6 gives a test program that displays minimum spanning trees for the graph in Figure 29.1 and the graph in Figure 29.3a, respectively.

LISTING 29.6 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<>(vertices, edges);
28        WeightedGraph<String>.MST tree1 = graph1.getMinimumSpanningTree();
29        System.out.println("tree1: Total weight is " +
30            tree1.getTotalWeight());
31        tree1.printTree();
32
33        edges = new int[][] {
34            {0, 1, 2}, {0, 3, 8},
35            {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
36            {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
37            {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
38            {4, 2, 5}, {4, 3, 6}
39        };
40
41        WeightedGraph<Integer> graph2 = new WeightedGraph<>(edges, 5);
42        WeightedGraph<Integer>.MST tree2 =
43            graph2.getMinimumSpanningTree(1);
44        System.out.println("\ntree2: Total weight is " +
45            tree2.getTotalWeight());
46        tree2.printTree();

```

create vertices

create edges

create graph1

MST for graph1

total weight

print tree

create edges

create graph2

MST for graph2

total weight

print tree

```
display search order
47
48     System.out.println("\nShow the search order for tree1:");
49     for (int i: tree1.getSearchOrder())
50         System.out.print(graph1.getVertex(i) + " ");
51     }
52 }
```



```
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)

Show the search order for tree1:
Seattle San Francisco Los Angeles Denver Kansas City Dallas
Houston Chicago Atlanta Miami New York Boston
```

The program creates a weighted graph for Figure 29.1 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 31) 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 **SearchTree** class.

graphical illustration

The graphical illustration of the minimum spanning tree is shown in Figure 29.9. The vertices are added to the tree in this order: Seattle, San Francisco, Los Angeles, Denver, Kansas City, Dallas, Houston, Chicago, Atlanta, Miami, New York, and Boston.

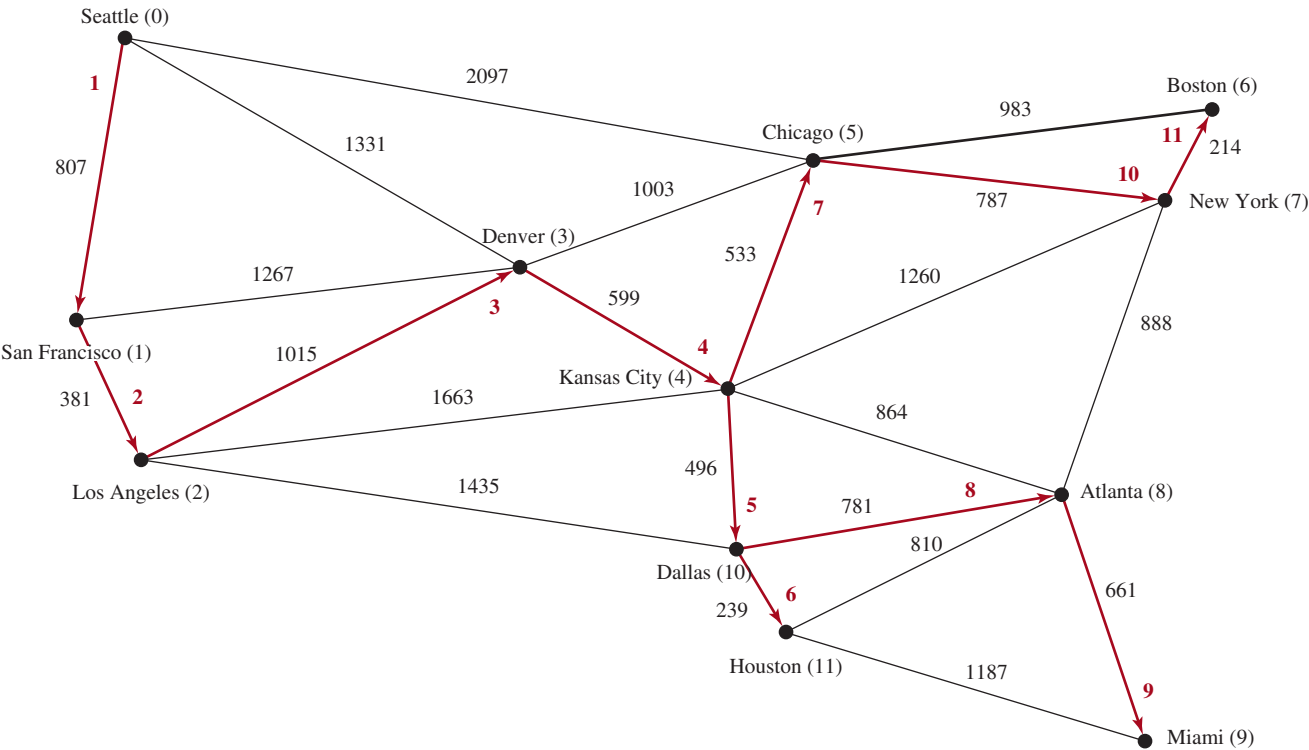
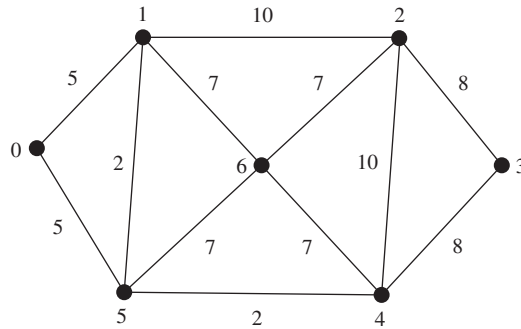


FIGURE 29.9 The edges in a minimum spanning tree for the cities are highlighted.

29.4.1 Find a minimum spanning tree for the following graph:



29.4.2 Is a minimum spanning tree unique if all edges have different weights?

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

29.4.4 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?

29.4.5 Show the output of the following code:

```
public class Test {
    public static void main(String[] args) {
        WeightedGraph<Character> graph = new WeightedGraph<>();
        graph.addVertex('U');
        graph.addVertex('V');
        graph.addVertex('X');
        int indexForU = graph.getIndex('U');
        int indexForV = graph.getIndex('V');
        int indexForX = graph.getIndex('X');
        System.out.println("indexForU is " + indexForU);
        System.out.println("indexForV is " + indexForV);
        System.out.println("indexForX is " + indexForV);
        graph.addEdge(indexForU, indexForV, 3.5);
        graph.addEdge(indexForV, indexForU, 3.5);
        graph.addEdge(indexForU, indexForX, 2.1);
        graph.addEdge(indexForX, indexForU, 2.1);
        graph.addEdge(indexForV, indexForX, 3.1);
        graph.addEdge(indexForX, indexForV, 3.1);
        WeightedGraph<Character>.MST mst
            = graph.getMinimumSpanningTree();
        graph.printWeightedEdges();
        System.out.println(mst.getTotalWeight());
        mst.printTree();
    }
}
```

29.5 Finding Shortest Paths

The shortest path between two vertices is a 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 **s** to vertex **v**, *Dijkstra's algorithm* finds the shortest path from **s** to all vertices. So Dijkstra's algorithm is known as a



Dijkstra's algorithm

single-source shortest path
shortest path

single-source shortest-path algorithm. The algorithm uses `cost[v]` to store the cost of a *shortest path* from vertex `v` to the source vertex `s`. `cost[s]` is 0. Initially assign infinity to `cost[v]` for all other vertices. The algorithm repeatedly finds a vertex `u` in `V-T` with the smallest `cost[u]` and moves `u` to `T`.
The algorithm is described in Listing 29.7.

LISTING 29.7 Dijkstra's Single-Source Shortest-Path Algorithm

Input: a graph $G = (V, E)$ with nonnegative weights
Output: a shortest-path tree with the source vertex `s` as the root

find next vertex
add a vertex to T

adjust cost[v]

```
1 ShortestPathTree getShortestPath(s) {
2   Let T be a set that contains the vertices whose
3   paths to s are known; Initially T is empty;
4   Set cost[s] = 0; and cost[v] = infinity for all other vertices in V;
5
6   while (size of T < n) {
7     Find u not in T with the smallest cost[u];
8     Add u to T;
9     for (each v not in T and (u, v) in E)
10      if (cost[v] > cost[u] + w(u, v)) {
11        cost[v] = cost[u] + w(u, v); parent[v] = u;
12      }
13   }
14 }
```

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 setting `cost[s]` to 0 (line 4), sets `cost[v]` to infinity for all other vertices. It then continuously adds a vertex (say `u`) from `V-T` into `T` with smallest `cost[u]` (lines 7-8), as shown in Figure 29.10a. After adding `u` to `T`, the algorithm updates `cost[v]` and `parent[v]` for each `v` not in `T` if `(u, v)` is in `T` and `cost[v] > cost[u] + w(u, v)` (lines 10-12).

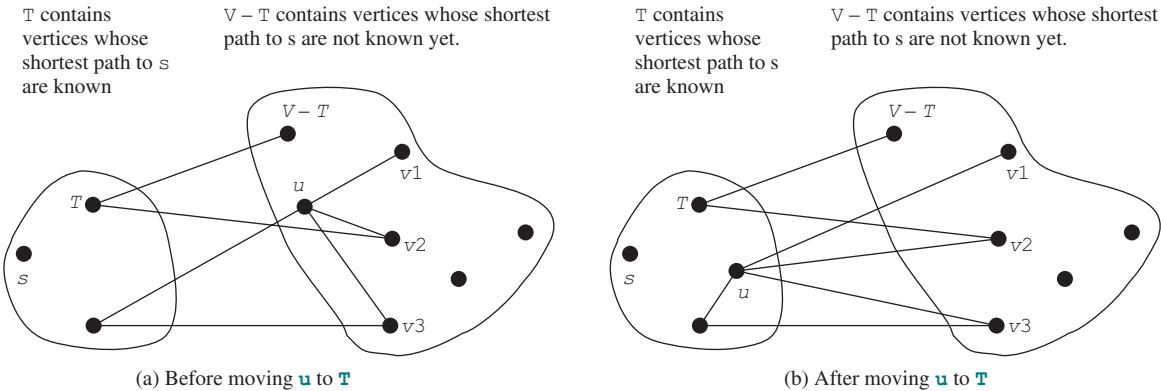


FIGURE 29.10 (a) Find a vertex `u` in `V-T` with the smallest `cost[u]`. (b) Update `cost[v]` for `v` in `V-T` and `v` is adjacent to `u`.

Let us illustrate Dijkstra's algorithm using the graph in Figure 29.11a. Suppose the source vertex is **1**. Therefore, **cost[1] = 0** and the costs for all other vertices are initially ∞ , as shown in Figure 29.11b. 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**.

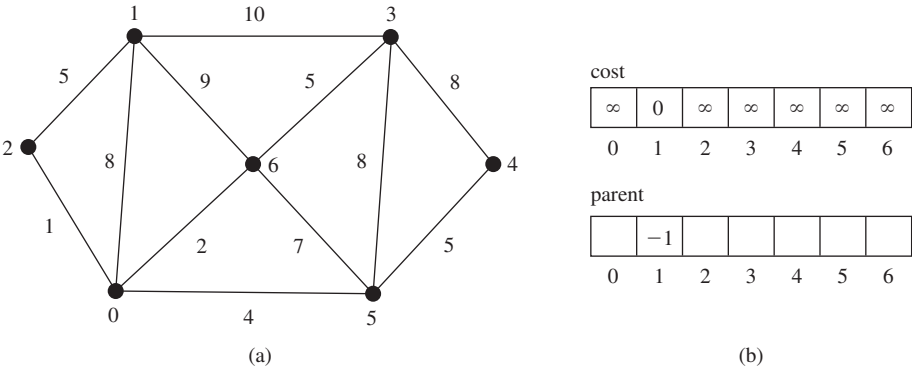


FIGURE 29.11 The algorithm will find all shortest paths from source vertex **1**.

Initially set **T** is empty. The algorithm selects the vertex with the smallest cost. In this case, the vertex is **1**. The algorithm adds **1** to **T**, as shown in Figure 29.12a. Afterward, it adjusts the cost for each vertex adjacent to **1**. The cost for vertices **2**, **0**, **6**, and **3** and their parents are now updated, as shown in Figure 29.12b.

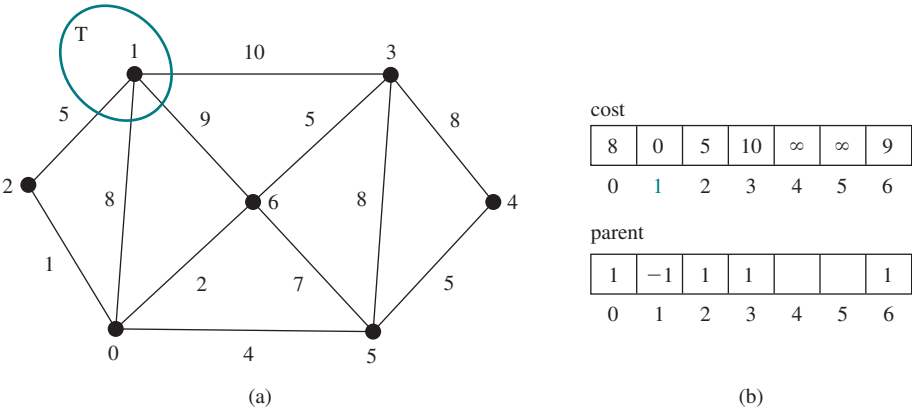


FIGURE 29.12 Now vertex **1** is in set **T**.

Vertices **2**, **0**, **6**, and **3** are adjacent to the source vertex, and vertex **2** is the one in **V-T** with the smallest cost, so add **2** to **T**, as shown in Figure 29.13 and update the cost and parent for vertices in **V-T** and adjacent to **2**. **cost[0]** is now updated to **6** and its parent is set to **2**. The arrow line from **1** to **2** indicates **1** is the parent of **2** after **2** is added into **T**.

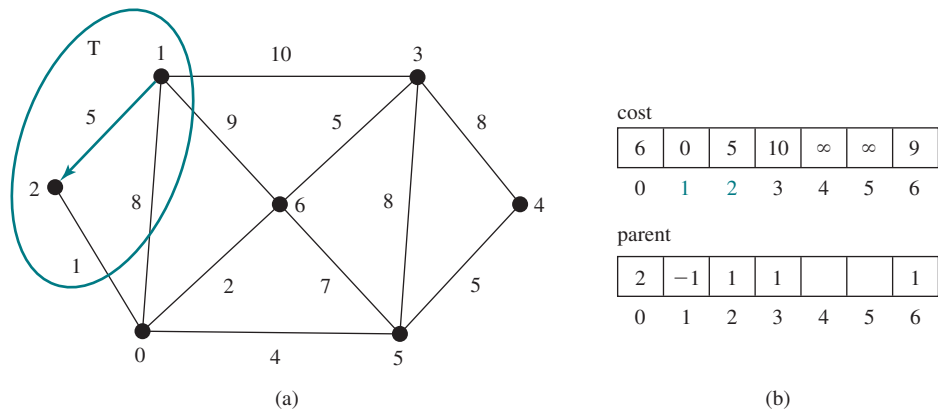


FIGURE 29.13 Now vertices 1 and 2 are in set T .

Now T contains $\{1, 2\}$. Vertex 0 is the one in $V-T$ with the smallest cost, so add 0 to T , as shown in Figure 29.14 and update the cost and parent for vertices in $V-T$ and adjacent to 0 if applicable. $\text{cost}[5]$ is now updated to 10 and its parent is set to 0 and $\text{cost}[6]$ is now updated to 8 and its parent is set to 0.

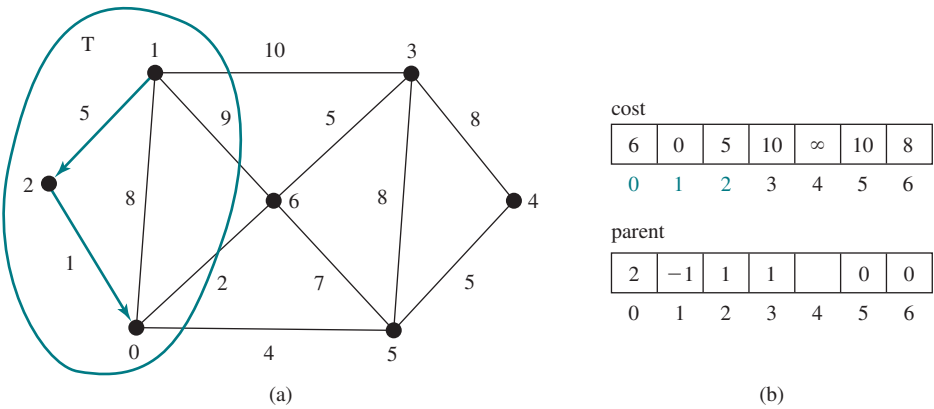


FIGURE 29.14 Now vertices $\{1, 2, 0\}$ are in set T .

Now T contains $\{1, 2, 0\}$. Vertex 6 is the one in $V-T$ with the smallest cost, so add 6 to T , as shown in Figure 29.15 and update the cost and parent for vertices in $V-T$ and adjacent to 6 if applicable.

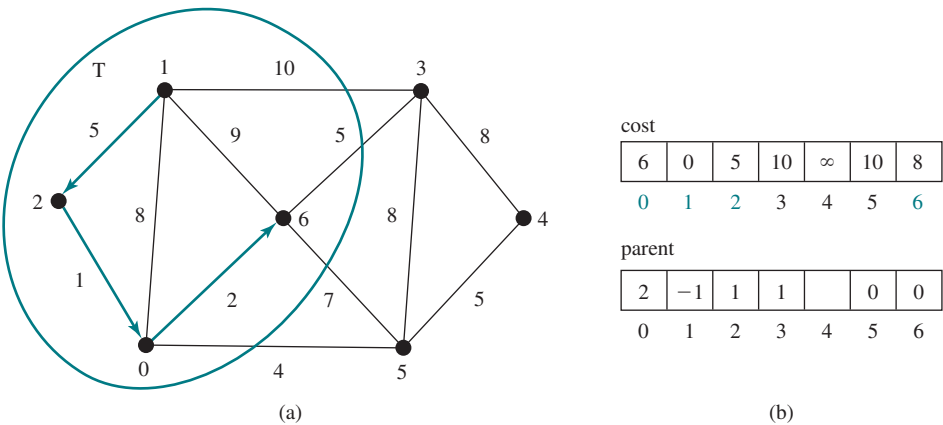


FIGURE 29.15 Now vertices $\{1, 2, 0, 6\}$ are in set T .

Now **T** contains {1, 2, 0, 6}. Vertex 3 or 5 is the one in **V-T** with the smallest cost. You may add either 3 or 5 into **T**. Let us add 3 to **T**, as shown in Figure 29.16 and update the cost and parent for vertices in **V-T** and adjacent to 3 if applicable. **cost**[4] is now updated to 18 and its parent is set to 3.

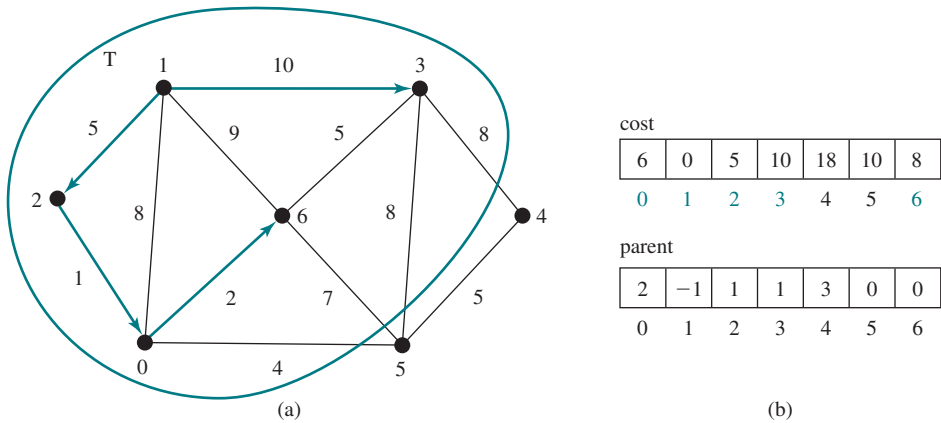


FIGURE 29.16 Now vertices {1, 2, 0, 6, 3} are in set **T**.

Now **T** contains {1, 2, 0, 6, 3}. Vertex 5 is the one in **V-T** with the smallest cost, so add 5 to **T**, as shown in Figure 29.17 and update the cost and parent for vertices in **V-T** and adjacent to 5 if applicable. **cost**[4] is now updated to 15, and its parent is set to 5.

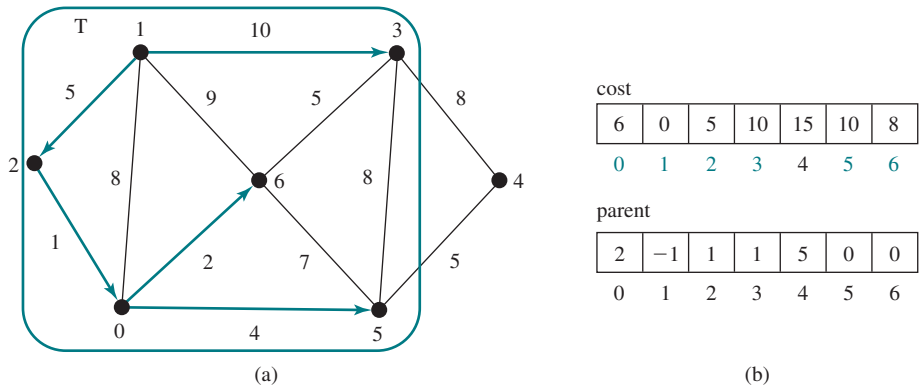


FIGURE 29.17 Now vertices {1, 2, 0, 6, 3, 5} are in set **T**. *Source:* Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

Now **T** contains {1, 2, 0, 6, 3, 5}. Vertex 4 is the one in **V-T** with the smallest cost, so add 4 to **T**, as shown in Figure 29.18.

As you can see, the algorithm essentially finds all 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 **SearchTree** class, as shown in Figure 29.19. **ShortestPathTree** is defined as an inner class in **WeightedGraph** in lines 200–224 in Listing 29.2.

shortest-path tree

The **getShortestPath(int sourceVertex)** method was implemented in lines 156–197 in Listing 29.2. The method sets **cost**[sourceVertex] to 0 (line 162) and **cost**[v] to

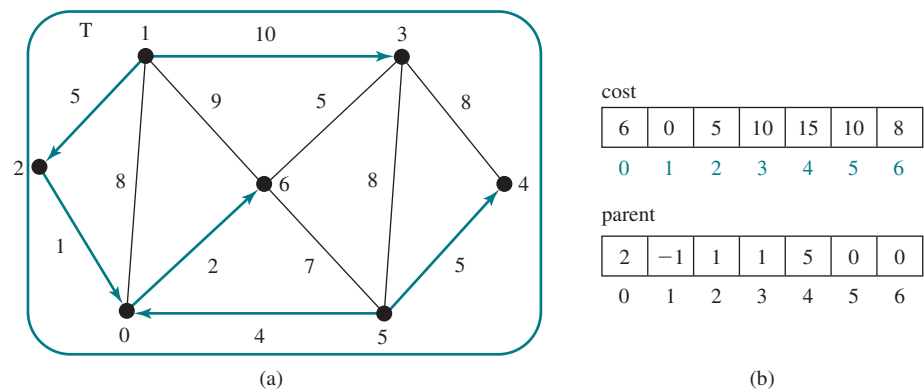


FIGURE 29.18 Now vertices {1, 2, 6, 0, 3, 5, 4} are in set **T**. Source: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

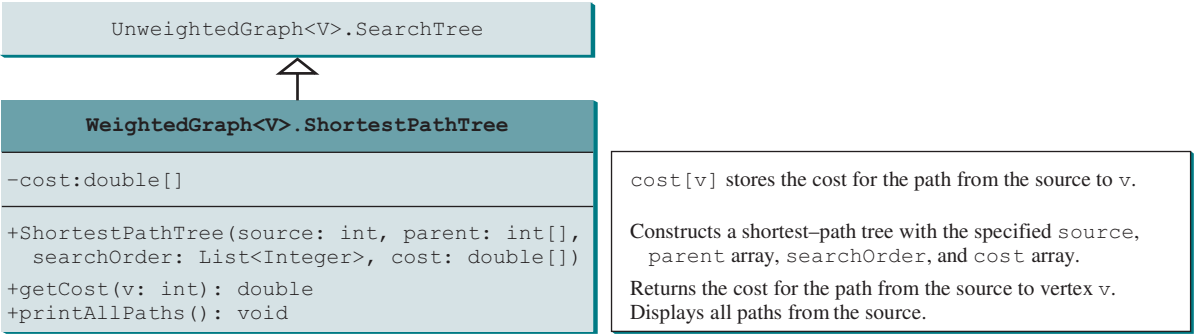


FIGURE 29.19 `WeightedGraph<V>.ShortestPathTree` extends `UnweightedGraph<V>.SearchTree`.

Source: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

infinity for all other vertices (lines 159–161). The parent of `sourceVertex` is set to `-1` (line 166). **T** is a list that stores the vertices added into the shortest-path tree (line 169). We use a list for **T** rather than a set in order to record the order of the vertices added to **T**.

Initially, **T** is empty. To expand **T**, the method performs the following operations:

1. Find the vertex **u** with the smallest `cost[u]` (lines 175–181).
2. If **u** is found, add it to **T** (line 183). Note that if **u** is not found (`u == -1`), the graph is not connected. The `break` statement exits the while loop in this case.
3. After adding **u** in **T**, update `cost[v]` and `parent[v]` for each **v** adjacent to **u** in **V-T** if `cost[v] > cost[u] + w(u, v)` (lines 186–192).

ShortestPathTree class

Once all vertices from **V** are added to **T**, an instance of `ShortestPathTree` is created (line 196).

The `ShortestPathTree` class extends the `SearchTree` class (line 200). To create an instance of `ShortestPathTree`, pass `sourceVertex`, `parent`, **T**, and `cost` (lines 204–205). `sourceVertex` becomes the root in the tree. The data fields `root`, `parent`, and `searchOrder` are defined in the `SearchTree` class, which is an inner class defined in `UnweightedGraph`.

Note testing whether a vertex **i** is in **T** by invoking `T.contains(i)` takes $O(n)$ time, since **T** is a list. Therefore, the overall time complexity for this implementation is $O(n^3)$. Interested readers may see Programming Exercise 29.20 for improving the implementation and reducing the complexity to $O(n^2)$.

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.

Dijkstra’s algorithm time complexity
greedy and dynamic programming

Listing 29.8 gives a test program that displays the shortest paths from Chicago to all other cities in Figure 29.1, and the shortest paths from vertex 3 to all vertices for the graph in Figure 29.3a, respectively.

LISTING 29.8 TestShortestPath.java

```

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<>(vertices, edges);
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
35            = tree1.getPath(graph1.getIndex("Houston"));
36        for (String s: path) {
37            System.out.print(s + " ");
38        }
39
40        edges = new int[][] {
41            {0, 1, 2}, {0, 3, 8},
42            {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
43            {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
44            {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
45            {4, 2, 5}, {4, 3, 6}
46        };
47        WeightedGraph<Integer> graph2 = new WeightedGraph<>(edges, 5);
48        WeightedGraph<Integer>.ShortestPathTree tree2 =
49            graph2.getShortestPath(3);
50        System.out.println("\n");
51        tree2.printAllPaths();
52    }
53 }

```

vertices

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: 2097.0)
A path from Chicago to San Francisco: Chicago Denver San Francisco
(cost: 2270.0)
A path from Chicago to Los Angeles: Chicago Denver Los Angeles
(cost: 2018.0)
A path from Chicago to Denver: Chicago Denver (cost: 1003.0)
A path from Chicago to Kansas City: Chicago Kansas City (cost: 533.0)
A path from Chicago to Chicago: Chicago (cost: 0.0)
A path from Chicago to Boston: Chicago Boston (cost: 983.0)
A path from Chicago to New York: Chicago New York (cost: 787.0)
A path from Chicago to Atlanta: Chicago Kansas City Atlanta
(cost: 1397.0)
A path from Chicago to Miami:
Chicago Kansas City Atlanta Miami (cost: 2058.0)
A path from Chicago to Dallas:
Chicago Kansas City Dallas (cost: 1029.0)
A path from Chicago to Houston:
Chicago Kansas City Dallas Houston (cost: 1268.0)
Shortest path from Houston to Chicago:
Houston Dallas Kansas City Chicago

All shortest paths from 3 are:
A path from 3 to 0: 3 1 0 (cost: 5.0)
A path from 3 to 1: 3 1 (cost: 3.0)
A path from 3 to 2: 3 2 (cost: 4.0)
A path from 3 to 3: 3 (cost: 0.0)
A path from 3 to 4: 3 4 (cost: 6.0)

```

The program creates a weighted graph for Figure 29.1 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 29.20. The shortest paths from Chicago to the cities are found in this order: Kansas City, New York, Boston, Denver, Dallas, Houston, Atlanta, Los Angeles, Miami, Seattle, and San Francisco.



- 29.5.1** Trace Dijkstra's algorithm for finding shortest paths from Boston to all other cities in Figure 29.1.
- 29.5.2** Is a shortest path between two vertices unique if all edges have different weights?
- 29.5.3** If you use an adjacency matrix to represent weighted edges, what would be the time complexity for Dijkstra's algorithm?
- 29.5.4** What happens to the `getShortestPath()` method in `WeightedGraph` if the source vertex cannot reach all vertices in the graph? Verify your answer by writing a test program that creates an unconnected graph and invoke the `getShortestPath()` method. How do you fix the problem by obtaining a partial shortest-path tree?

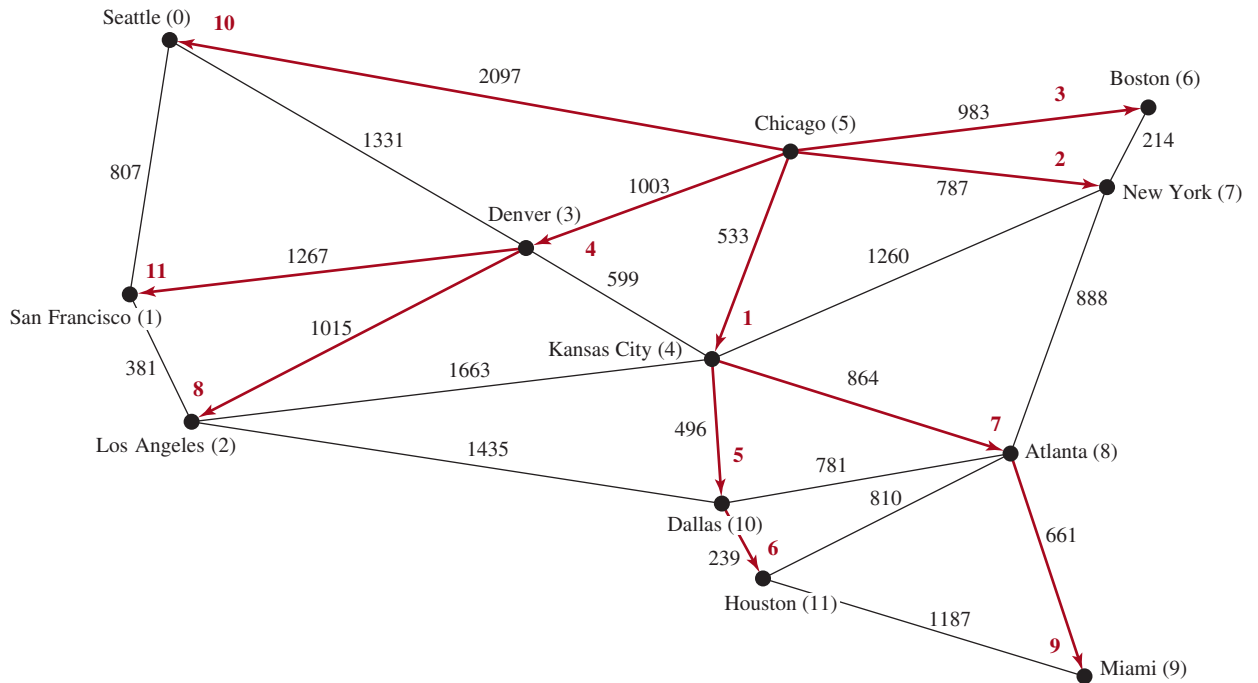


FIGURE 29.20 The shortest paths from Chicago to all other cities are highlighted.

29.5.5 If there is no path from vertex **v** to the source vertex, what will be **cost[v]**?

29.5.6 Assume the graph is connected; will the **getShortestPath** method find the shortest paths correctly if lines 159–161 in **WeightedGraph** are deleted?

29.5.7 Show the output of the following code:

```
public class Test {
    public static void main(String[] args) {
        WeightedGraph<Character> graph = new WeightedGraph<>();
        graph.addVertex('U');
        graph.addVertex('V');
        graph.addVertex('X');
        int indexForU = graph.getIndex('U');
        int indexForV = graph.getIndex('V');
        int indexForX = graph.getIndex('X');
        System.out.println("indexForU is " + indexForU);
        System.out.println("indexForV is " + indexForV);
        System.out.println("indexForX is " + indexForX);
        graph.addEdge(indexForU, indexForV, 3.5);
        graph.addEdge(indexForV, indexForU, 3.5);
        graph.addEdge(indexForU, indexForX, 2.1);
        graph.addEdge(indexForX, indexForU, 2.1);
        graph.addEdge(indexForV, indexForX, 3.1);
        graph.addEdge(indexForX, indexForV, 3.1);
        WeightedGraph<Character>.ShortestPathTree tree =
            graph.getShortestPath(1);
    }
}
```

```
graph.printWeightedEdges();
tree.printTree();
}
```



29.6 Case Study: The Weighted Nine Tails Problem

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

Section 28.10 presented the nine tails problem and solved it using the BFS algorithm. This section presents a variation of the nine tails 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 29.21a to those in Figure 29.21b by flipping the first coin in the first row and its two neighbors. Thus, the weight for this move is 3. You can move from the coins in Figure 29.21c to Figure 29.21d by flipping the five coins. So the weight for this move is 5.

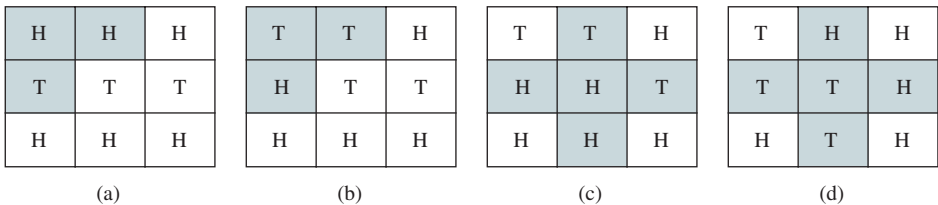


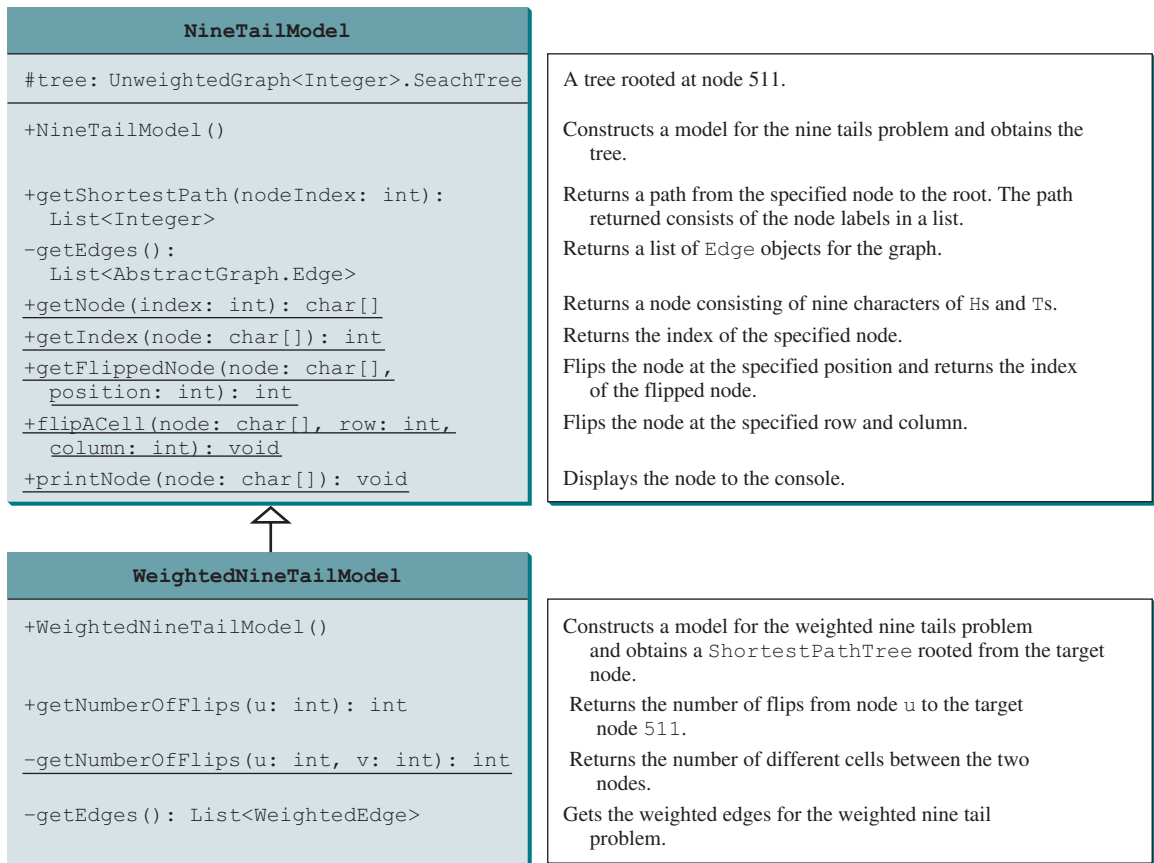
FIGURE 29.21 The weight for each move is the number of flips for the move.

The weighted nine tails problem can be reduced to finding a 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 in Section 28.10, 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 29.22.

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. 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 29.9 implements the **WeightedNineTailModel**.

FIGURE 29.22 The `WeightedNineTailModel` class extends `NineTailModel`.LISTING 29.9 `WeightedNineTailModel.java`

```

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

```

```

21
22     for (int u = 0; u < NUMBER_OF_NODES; u++) {
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 **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 29.10 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 29.10 WeightedNineTail.java

```

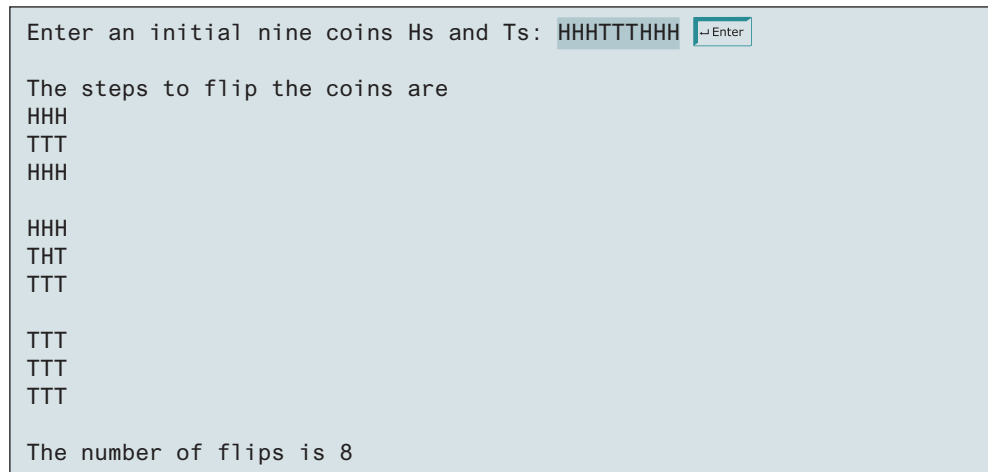
1  import java.util.Scanner;
2
3  public class WeightedNineTail {
4      public static void main(String[] args) {
5          // Prompt the user to enter the nine coins' Hs and Ts

```

```

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(NineTailModel.getNode(path.get(i)));  print node
18
19      System.out.println("The number of flips is " +
20          model.getNumberOfFlips(NineTailModel.getIndex(initialNode)));  number of flips
21  }
22  }

```



```

Enter an initial nine coins Hs and Ts: HHHTTTTHH 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 **Hs** and **Ts** 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 16–17), and invokes **getNumberOfFlips** to get the number of flips needed to reach the target node (line 20).

- 29.6.1** Why is the **tree** data field in **NineTailModel** in Listing 28.13 defined protected?
- 29.6.2** How are the nodes created for the graph in **WeightedNineTailModel**?
- 29.6.3** How are the edges created for the graph in **WeightedNineTailModel**?



KEY TERMS

Dijkstra's algorithm 1131
 edge-weighted graph 1115
 minimum spanning tree 1125
 Prim's algorithm 1125

shortest path 1137
 single-source shortest path 1132
 vertex-weighted graph 1115

CHAPTER SUMMARY

1. You can use adjacency matrices or lists to store 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.
3. Prim's algorithm for finding a minimum spanning tree works as follows: the algorithm starts with a spanning tree **T** that contains an arbitrary vertex. The algorithm expands the tree by adding a vertex with the minimum-weight edge incident to a vertex already in the tree.
4. Dijkstra's algorithm starts the search from the source vertex and keeps finding vertices that have the shortest path to the source until all vertices are found.



Quiz

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

PROGRAMMING EXERCISES

- *29.1** (*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.
- *29.2** (*Implement Prim's algorithm using an adjacency matrix*) The text implements Prim's algorithm using lists for adjacent edges. Implement the algorithm using an adjacency matrix for weighted graphs.
- *29.3** (*Implement Dijkstra's algorithm using an adjacency matrix*) The text implements Dijkstra's algorithm using lists for adjacent edges. Implement the algorithm using an adjacency matrix for weighted graphs.
- *29.4** (*Modify weight in the nine tails problem*) In the text, we assign the number of the flips as the weight for each move. Assuming the weight is three times of the number of flips, revise the program.
- *29.5** (*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.)
- **29.6** (*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**.
- **29.7** (*Weighted 4×4 16 tails*) Revise Listing 29.9, **WeightedNineTail.java**, for the weighted 4×4 16 tails problem. Your program should read the model object created from the preceding exercise.

****29.8** (*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 similar to finding a shortest Hamiltonian cycle in Programming Exercise 28.17. Add the following method in the `WeightedGraph` class:

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

***29.9** (*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 29.23 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 added.

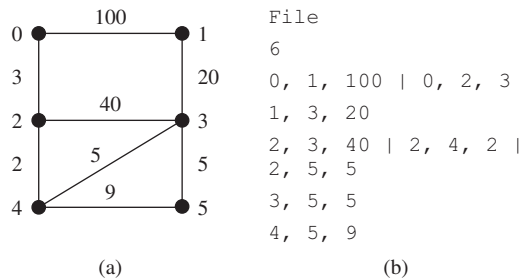


FIGURE 29.23 The vertices and edges of a weighted graph can be stored in a file.

Only one edge is represented in the file. When you construct a graph, both edges need to be added. 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\WeightedGraphSample2.txt Enter
The number of vertices is 6
Vertex 0: (0, 1, 100) (0, 2, 3)
Vertex 1: (1, 0, 100) (1, 3, 20)
Vertex 2: (2, 0, 3) (2, 3, 40) (2, 4, 2) (2, 5, 5)
Vertex 3: (3, 1, 20) (3, 2, 40) (3, 5, 5)
Vertex 4: (4, 2, 2) (4, 5, 9)
Vertex 5: (5, 2, 5) (5, 3, 5) (5, 4, 9)
Total weight in MST is 35
Root is: 0
Edges: (3, 1) (0, 2) (5, 3) (2, 4) (2, 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.)

***29.10** (Create a file for a graph) Modify Listing 29.3, `TestWeightedGraph.java`, to create a file for representing **graph1**. The file format is described in Programming Exercise 29.9. Create the file from the array defined in lines 7–24 in Listing 29.3. 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
```

***29.11** (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 29.9. Your program should prompt the user to enter the name of the file then two vertices, and should display a shortest path between the two vertices. For example, for the graph in Figure 29.23, a shortest path between **0** and **1** can be displayed as **0 2 5 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, 1, 100.0) (0, 2, 3.0)
Vertex 1: (1, 0, 100.0) (1, 3, 20.0)
Vertex 2: (2, 0, 3.0) (2, 3, 40.0) (2, 4, 2.0) (2, 5, 5.0)
Vertex 3: (3, 1, 20.0) (3, 2, 40.0) (3, 5, 5.0)
Vertex 4: (4, 2, 2.0) (4, 5, 9.0)
Vertex 5: (5, 2, 5.0) (5, 3, 5.0) (5, 4, 9.0)
A path from 0 to 1: 0 2 5 3 1
```

- *29.12** (*Display weighted graphs*) Revise **GraphView** in Listing 28.6 to display a weighted graph. Write a program that displays the graph in Figure 29.1 as shown in Figure 29.24. (Instructors may ask students to expand this program by adding new cities with appropriate edges into the graph.)

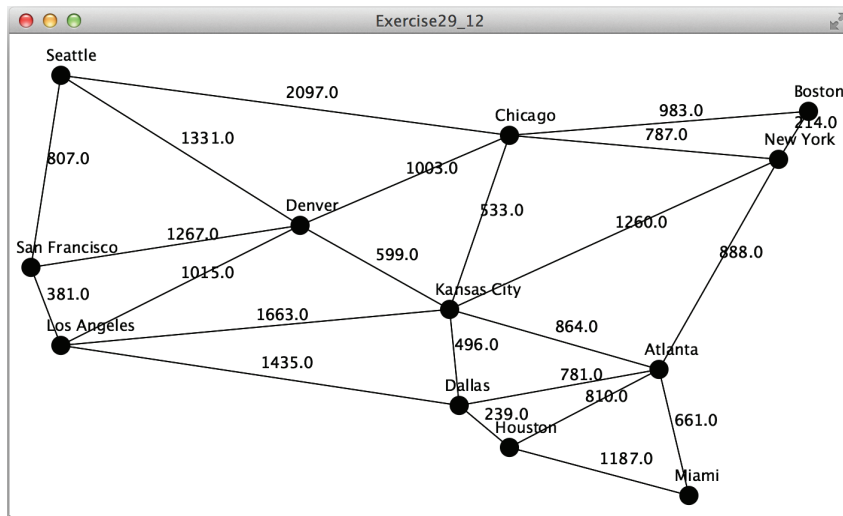


FIGURE 29.24 Programming Exercise 29.12 displays a weighted graph.

- *29.13** (*Display shortest paths*) Revise **GraphView** in Listing 28.6 to display a weighted graph and a shortest path between the two specified cities, as shown in Figure 29.25. 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 text to alert the user.

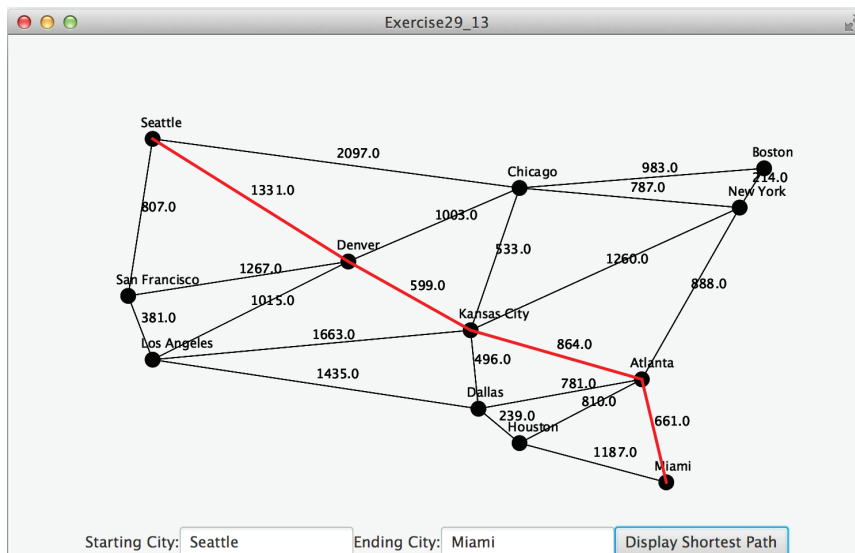


FIGURE 29.25 Programming Exercise 29.13 displays a shortest path.

***29.14** (*Display a minimum spanning tree*) Revise **GraphView** in Listing 28.6 to display a weighted graph and a minimum spanning tree for the graph in Figure 29.1, as shown in Figure 29.26. The edges in the MST are shown in red.

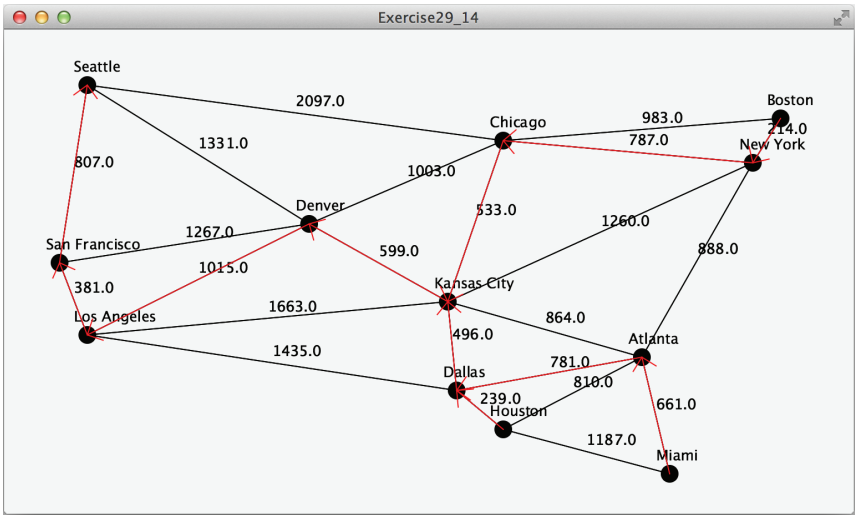


FIGURE 29.26 Programming Exercise 29.14 displays an MST.

*****29.15** (*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 29.27. The user can also create an edge to connect two vertices. To simplify the program, assume 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.

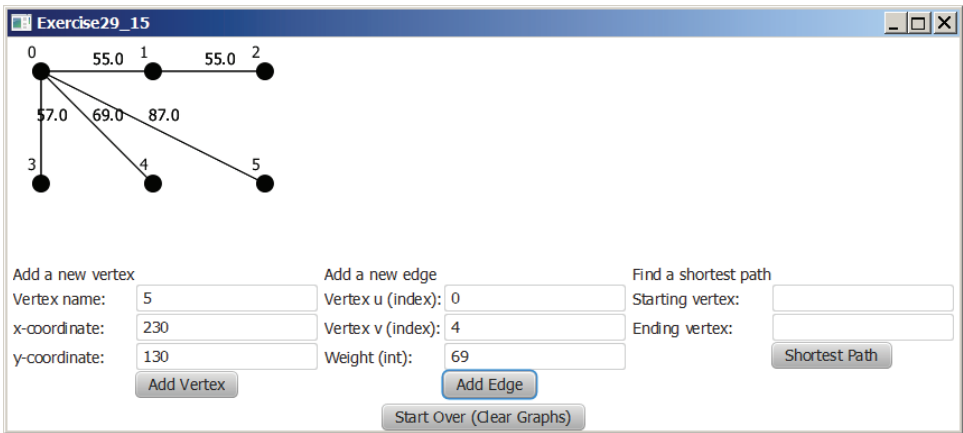


FIGURE 29.27 The program can add vertices and edges and display a shortest path between two specified vertices. *Source:* Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

*****29.16** (*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 29.28. The user can also create an edge to connect two vertices. To simplify the program, assume 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.

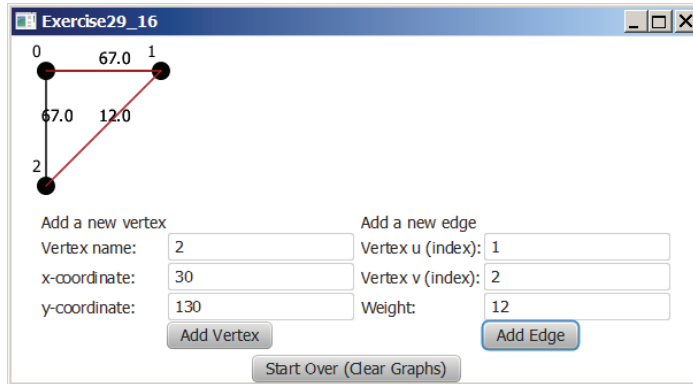


FIGURE 29.28 The program can add vertices and edges and display MST dynamically.

Source: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

*****29.17** (*Weighted graph visualization tool*) Develop a GUI program as shown in Figure 29.2, 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 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 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.

*****29.18** (*Alternative version of Dijkstra algorithm*) An alternative version of the Dijkstra algorithm can be described as follows:

Input: a weighted graph $G = (V, E)$ with nonnegative weights

Output: A shortest-path tree from a source vertex s

```

1 ShortestPathTree getShortestPath(s) {
2   Let T be a set that contains the vertices whose
3   paths to s are known;
4   Initially T contains source vertex s with cost[s] = 0;      add initial vertex
5   for (each u in V - T)
6     cost[u] = infinity;
7
8   while (size of T < n) {                                     more vertex
9     Find v in V - T with the smallest cost[u] + w(u, v) value find next vertex

```

add a vertex

```

10         among all  $u$  in  $T$ ;
11         Add  $v$  to  $T$  and set  $\text{cost}[v] = \text{cost}[u] + w(u, v)$ ;
12          $\text{parent}[v] = u$ ;
13     }
14 }
```

The algorithm uses $\text{cost}[v]$ to store the cost of a shortest path from vertex v to the source vertex s . $\text{cost}[s]$ is 0. Initially assign infinity to $\text{cost}[v]$ 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 $\text{cost}[u] + w(u, v)$ is the smallest, and moves v to T . The shortest-path algorithm given in the text continuously updates the cost and parent for a vertex in $V-T$. This algorithm initializes the cost to infinity for each vertex and then changes the cost for a vertex only once when the vertex is added into T . Implement this algorithm and use Listing 29.7, `TestShortestPath.java`, to test your new algorithm.

*****29.19** (Find u with smallest $\text{cost}[u]$ efficiently) The `getShortestPath` method finds a u with the smallest $\text{cost}[u]$ using a linear search, which takes $O(|V|)$. The search time can be reduced to $O(\log |V|)$ using an AVL tree. Modify the method using an AVL tree to store the vertices in $V-T$. Use Listing 29.7, `TestShortestPath.java`, to test your new implementation.

*****29.20** (Test if a vertex u is in T efficiently) Since T is implemented using a list in the `getMinimumSpanningTree` and `getShortestPath` methods in Listing 29.2, `WeightedGraph.java`, testing whether a vertex u is in T by invoking `T.contains(u)` takes $O(n)$ time. Modify these two methods by introducing an array named `isInT`. Set `isInT[u]` to `true` when a vertex u is added to T . Testing whether a vertex u is in T can now be done in $O(1)$ time. Write a test program using the following code, where `graph1` is created from Figure 29.1:

```

WeightedGraph<String> graph1 = new WeightedGraph<>(edges, vertices);
WeightedGraph<String>.MST tree1 = graph1.getMinimumSpanningTree();
System.out.println("Total weight is " + tree1.getTotalWeight());
tree1.printTree();

WeightedGraph<String>.ShortestPathTree tree2 =
    graph1.getShortestPath(graph1.getIndex("Chicago"));
tree2.printAllPaths();
```