CHAPTER 25

Weighted Graphs and Applications

Objectives

- To represent weighted edges using adjacency matrices and adjacency lists (§25.2).
- To model weighted graphs using the **`WeightedGraph`** class that extends the **`Graph`** class (§25.3).
- To design and implement the algorithm for finding a minimum spanning tree (§25.4).
- To define the **`MST`** class that extends the **`Tree`** class (§25.4).
- To design and implement the algorithm for finding single-source shortest paths (§25.5).
- To define the **`ShortestPathTree`** class that extends the **`Tree`** class (§25.5).
- To solve the weighted nine tail problem using the shortest-path algorithm (§25.6).

25.1 Introduction

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

Figure 24.1 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 25.1. 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*.

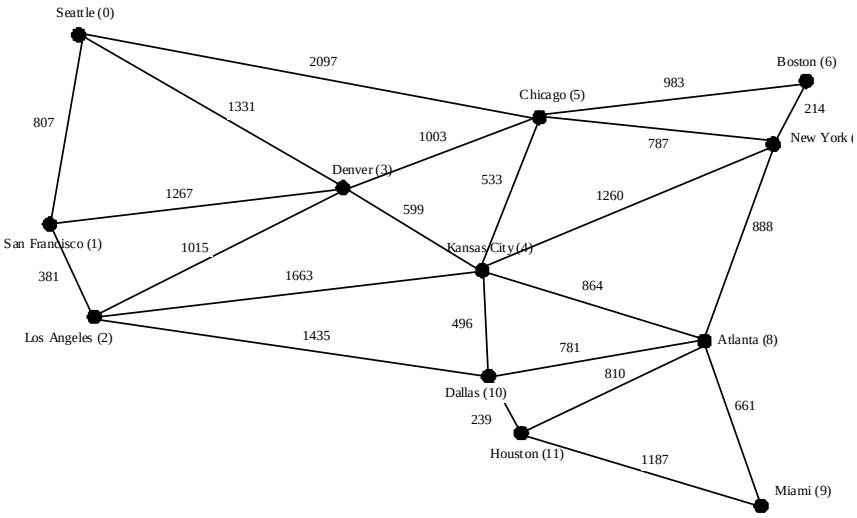


Figure 25.1

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 vectors, adjacency matrices, and adjacency lists, and how to model a graph using the `Graph` class. The 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 §25.4 and the algorithm for finding shortest paths in §25.5.

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. 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 25.2.

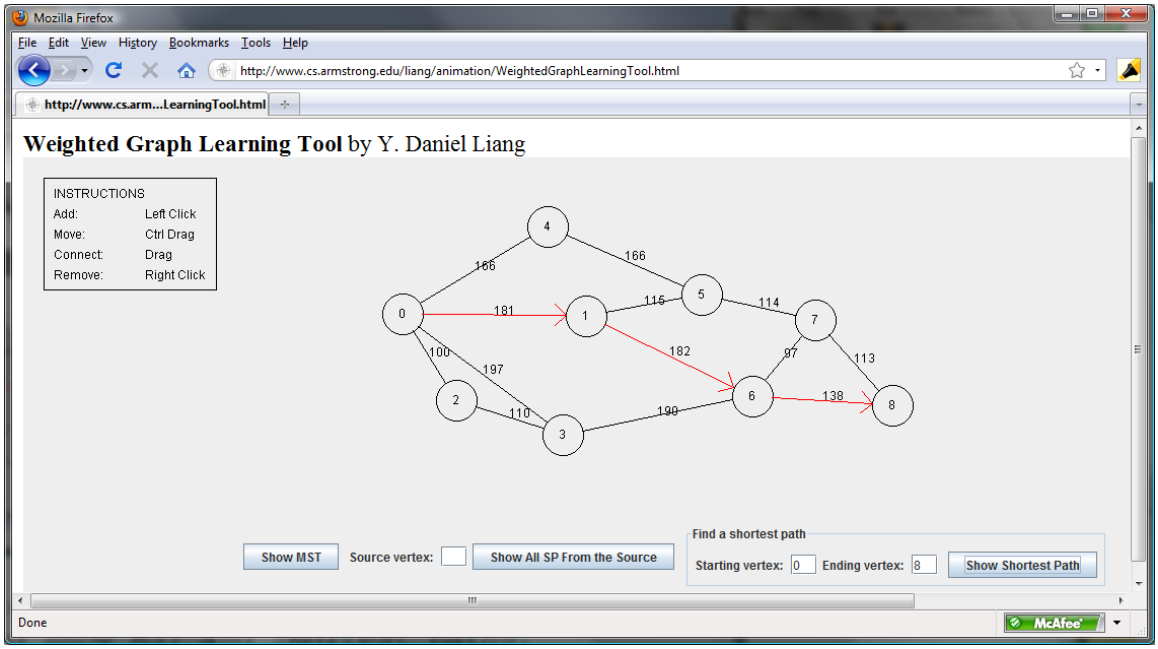


Figure 25.2

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

25.2 Representing Weighted Graphs

Key Point: *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.

25.2.1 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 25.3 using the following array:

```
int edges[][3] =
{
  {0, 1, 2}, {0, 3, 8},
  {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
  {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
  {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
  {4, 2, 5}, {4, 3, 6}
};
```

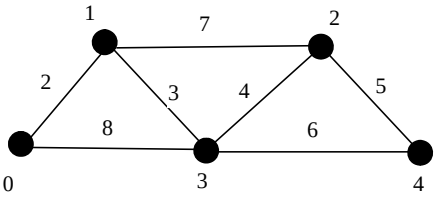


Figure 25.3

Each edge is assigned a weight on an edge-weighted graph.

NOTE

For simplicity, we assume that the *weights are integers*.

25.2.2 Weighted Adjacency Lists

Another way to represent the edges is to define edges as objects. The **`Edge`** class was defined to represent edges in unweighted graphs. For weighted edges, we define the **`WeightedEdge`** class as shown in Listing 25.1.

Listing 25.1 WeightedEdge.cpp

```
1  #ifndef WEIGHTEDEDGE_H
2  #define WEIGHTEDEDGE_H
3
4  #include "Edge.h"
5
6  class WeightedEdge : public Edge
```

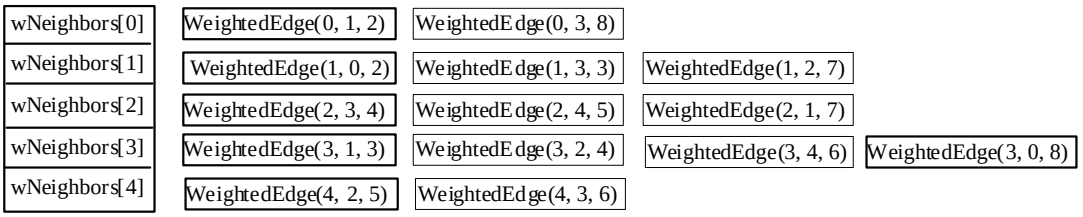
```
 7  {
 8  public:
 9    double weight; // The weight on edge (u, v)
10
11    // Create a weighted edge on (u, v)
12    WeightedEdge(int u, int v, double weight): Edge(u, v)
13    {
14      this->weight = weight;
15    }
16
17    // Compare edges based on weights
18    bool operator<(const WeightedEdge& edge) const
19    {
20      return (*this).weight < edge.weight;
21    }
22
23    bool operator<=(const WeightedEdge& edge) const
24    {
25     return (*this).weight <= edge.weight;
26    }
27
28    bool operator>(const WeightedEdge& edge) const
29    {
30      return (*this).weight > edge.weight;
31    }
32
33    bool operator>=(const WeightedEdge& edge) const
34    {
35      return (*this).weight >= edge.weight;
36    }
37
38    bool operator==(const WeightedEdge& edge) const
39    {
40      return (*this).weight == edge.weight;
41    }
42
43    bool operator!=(const WeightedEdge& edge) const
44    {
45      return (*this).weight != edge.weight;
46    }
47  };
48  #endif
```

The **`Edge`** class, defined in Listing 24.1, represents an edge from vertex **`u`** to **`v`**. **`WeightedEdge`** extends **`Edge`** with a new property **`weight`**.

To create a **`WeightedEdge`** object, use **`WeighedEdge(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, we define the operator functions (**`<`**, **`<=`**, **`==`**, **`!=`**, **`>`**, **`>=`**) in the **`WeightedEdge`** class.

For unweighted graphs, we use adjacency edge lists to represent edges. For weighted graphs, we still use adjacency edge lists. Since **`WeightedEdge`** is derived from **`Edge`**. The adjacency edge list for unweighted graphs can be used for weighted graphs. For example, the adjacency edge lists for the vertices in the graph in Figure 25.3 can be represented as follows:

```
vector<vector<Edge*>> neighbors;
for (unsigned i = 0; i < numberOfVertices; i++)
{
  neighbors.push_back(vector<Edge*>());
}
neighbors[0].push_back(new WeightedEdge(0, 1, 2));
neighbors[0].push_back(new WeightedEdge(0, 3, 8));
...
```



Check point

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

25.2 What is the output of the following code?

```
vector<WeightedEdge> list;
list.push_back(WeightedEdge(1, 2, 3.5));
list.push_back(WeightedEdge(2, 3, 5.5));
list.push_back(WeightedEdge(4, 3, 4.5));
cout << max_element(list.begin(), list.end())->weight << endl;
```

25.3 The `WeightedGraph` Class

Key Point: *The **`WeightedGraph`** class extends **`Graph`**.*

The preceding chapter designed the **`Graph`** class for modeling graphs. Following this pattern, we design **`WeightedGraph`** as a subclass of **`Graph`**, as shown in Figure 25.4.

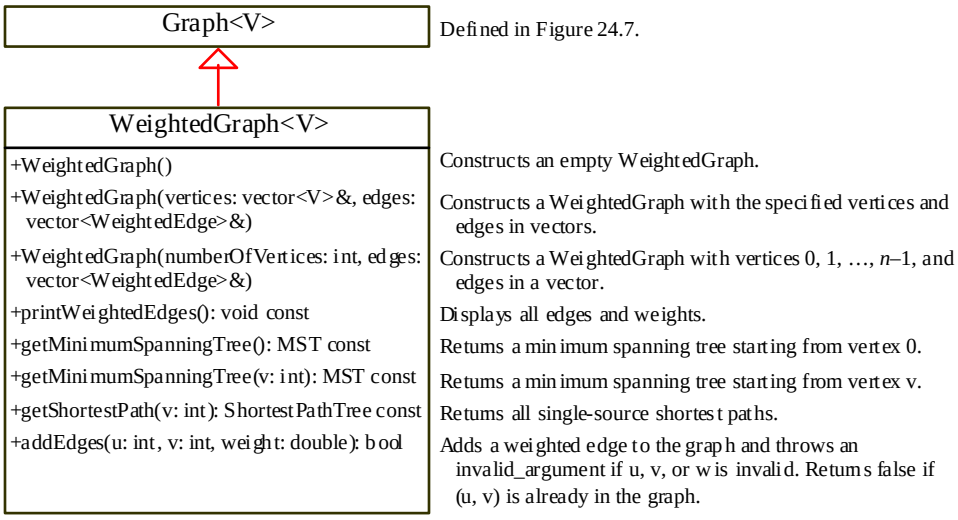


Figure 25.4

`WeightedGraph` *extends* ***`Graph`***.

`WeightedGraph` simply extends **`Graph`**. **`WeightedGraph`** inherits all functions from **`Graph`** and also introduces the new functions for obtaining minimum spanning trees and for finding single-source all shortest paths. Minimum spanning trees and shortest paths will be introduced in §25.4 and §25.5, respectively.

The class contains three constructors. The first is a no-arg constructor to create an empty graph. The second constructs a **`WeightedGraph`** with the specified vertices and edges in vectors. The third constructs a **`WeightedGraph`** with vertices **`0`**, **`1`**, ..., **`n-1`** and an edge vector. The **`printWeightedEdges`** function displays all edges for each vertex. Listing 25.2 implements **`WeightedGraph`**.

Listing 25.2 WeightedGraph.h

```
 1  #ifndef WEIGHTEDGRAPH_H
 2  #define WEIGHTEDGRAPH_H
 3
 4  #include "Graph.h"
 5  #include "WeightedEdge.h" // Defined in Listing 25.1
 6  #include "MST.h" // Defined in Listing 25.3
 7  #include "ShortestPathTree.h" // Defined in Listing 25.8
 8
 9  template<typename V>
10  class WeightedGraph : public Graph<V>
11  {
12  public:
```

```
13    // Construct an empty graph
14    WeightedGraph();
15
16    // Construct a graph from vertices and edges objects
17    WeightedGraph(vector<V>& vertices, vector<WeightedEdge>& edges);
18
19    // Construct a graph with vertices 0, 1, ..., n-1 and
20    // edges in a vector
21    WeightedGraph(int numberOfVertices, vector<WeightedEdge>& edges);
22
23    // Print all edges in the weighted tree
24    void WeightedGraph<V>::printWeightedEdges();
25
26    // Add a weighted edge
27    bool addEdge(int u, int v, double w);
28
29    // Get a minimum spanning tree rooted at vertex 0
30    MST getMinimumSpanningTree();
31
32    // Get a minimum spanning tree rooted at a specified vertex
33    MST getMinimumSpanningTree(int startingVertex);
34
35    // Find single-source shortest paths
36    ShortestPathTree getShortestPath(int sourceVertex);
37  };
38
39  const int INFINITY = 2147483647;
40
41  template<typename V>
42  WeightedGraph<V>::WeightedGraph()
43  {
44  }
45
46  template<typename V>
47  WeightedGraph<V>::WeightedGraph(vector<V>& vertices,
48    vector<WeightedEdge>& edges)
49  {
50    // Add vertices to the graph
51    for (unsigned i = 0; i < vertices.size(); i++)
52    {
53      addVertex(vertices[i]);
54    }
55
56    // Add edges to the graph
57    for (unsigned i = 0; i < edges.size(); i++)
58    {
59      addEdge(edges[i].u, edges[i].v, edges[i].weight);
60    }
61  }
62
63  template<typename V>
64  WeightedGraph<V>::WeightedGraph(int numberOfVertices,
65    vector<WeightedEdge>& edges)
66  {
67    // Add vertices to the graph
68    for (int i = 0; i < numberOfVertices; i++)
69      addVertex(i); // vertices is {0, 1, 2, ..., n-1}
70
71    // Add edges to the graph
```

```
  for (unsigned i = 0; i < edges.size(); i++)
  {
    addEdge(edges[i].u, edges[i].v, edges[i].weight);
  }
}

template<typename V>
void WeightedGraph<V>::printWeightedEdges()
{
  for (int i = 0; i < getSize(); i++)
  {
    // Display all edges adjacent to vertex with index i
    cout << "Vertex " << getVertex(i) << "(" << i << "): ";

    // Display all weighted edges
    for (Edge* e: neighbors[i])
    {
      cout << "(" << e->u << ", " << e->v << ", "
        << static_cast<WeightedEdge*>(e)->weight << ") ";
    }
    cout << endl;
  }
}

template<typename V>
bool WeightedGraph<V>::addEdge(int u, int v, double w)
{
  return createEdge(new WeightedEdge(u, v, w));
}

template<typename V>
MST WeightedGraph<V>::getMinimumSpanningTree()
{
  return getMinimumSpanningTree(0);
}

template<typename V>
MST WeightedGraph<V>::getMinimumSpanningTree(int startingVertex)
{
  vector<int> T; // T contains the vertices in the tree

  vector<int> parent(getSize()); // Parent of a verte
  parent[startingVertex] = -1; // startingVertex is the root
  double totalWeight = 0; // Total weight of the tree thus far

  // cost[v] stores the cost by adding v to the tree
  vector<double> cost(getSize());
  for (unsigned i = 0; i < cost.size(); i++)
  {
    cost[i] = INFINITY; // Initial cost set to infinity
  }
  cost[startingVertex] = 0; // Cost of source is 0

  // Expand T
  while (T.size() < getSize())
  {
    // Find smallest cost v in V - T
    int u = -1; // Vertex to be determined
    double currentMinCost = INFINITY;
```

```
131       for (int i = 0; i < getSize(); i++)
132       {
133         if (find(T.begin(), T.end(), i)== T.end()
134           && cost[i] < currentMinCost)
135         {
136           currentMinCost = cost[i];
137           u = i;
138         }
139       }
140
141       T.push_back(u); // Add a new vertex to T
142       totalWeight += cost[u]; // Add cost[u] to the tree
143
144       // Adjust cost[v] for v that is adjacent to u and v in V - T
145       for (Edge* e: neighbors[u])
146       {
147         if (find(T.begin(), T.end(), e->v) == T.end()
148           && cost[e->v] > static_cast<WeightedEdge*>(e)->weight)
149         {
150           cost[e->v] = static_cast<WeightedEdge*>(e)->weight;
151           parent[e->v] = u;
152         }
153       }
154     } // End of while
155
156     return MST(startingVertex, parent, T, totalWeight);
157   }
158
159   template<typename V>
160   ShortestPathTree WeightedGraph<V>::getShortestPath(int
sourceVertex)
161   {
162     // T stores the vertices whose path found so far
163     vector<int> T;
164
165     // parent[v] stores the previous vertex of v in the path
166     vector<int> parent(getSize());
167     parent[sourceVertex] = -1; // The parent of source is set to -1
168
169     // cost[v] stores the cost of the path from v to the source
170     vector<double> cost(getSize());
171     for (unsigned i = 0; i < cost.size(); i++)
172     {
173       cost[i] = INFINITY; // Initial cost set to infinity
174     }
175     cost[sourceVertex] = 0; // Cost of source is 0
176
177     // Expand T
178     while (T.size() < getSize())
179     {
180       // Find smallest cost v in V - T
181       int u = -1; // Vertex to be determined
182       double currentMinCost = INFINITY;
183       for (int i = 0; i < getSize(); i++)
184       {
185         if (find(T.begin(), T.end(), i)== T.end()
186           && cost[i] < currentMinCost)
187         {
188           currentMinCost = cost[i];
```

```
189           u = i;
190         }
191       }
192
193       if (u == -1) break;
194
195       T.push_back(u); // Add a new vertex to T
196
197       // Adjust cost[v] for v that is adjacent to u and v in V - T
198       for (Edge* e:  neighbors[u])
199       {
200         if (find(T.begin(), T.end(), e->v) == T.end() &&
201           cost[e->v] > cost[u] + static_cast<WeightedEdge*>(e)-
>weight)
202         {
203           cost[e->v] = cost[u] + static_cast<WeightedEdge*>(e)-
>weight;
204           parent[e->v] = u;
205         }
206       }
207     } // End of while
208
209     // Create a ShortestPathTree
210     return ShortestPathTree(sourceVertex, parent, T, cost);
211   }
212
213   #endif
```

WeightedGraph is derived from **Graph** (line 10). The properties **vertices** and **neighbors** are defined as protected in the parent class **Graph**, so they can be accessed in the child class **WeightedGraph**.

When a **WeightedGraph** is constructed, its vertices and adjacency edge lists are created by invoking the **addVertex** function (lines 56, 72) and the **addEdge** function (lines 62, 77). The **addVertex** function is defined the **Graph** class in Listing 24.3. The **addEdge** function invokes the **createEdge** function to add a weighted edge to the graph (lines 96-100). The **createEdge** function is defined as protected in Listing 24.3 Graph.h. This function checks whether the edge is valid and throws an **invalid_argument** exception if the edge is invalid.

Listing 25.3 gives a test program that creates a graph for the one in Figure 25.1 and another graph for the one in Figure 25.3.

Listing 25.3 TestWeightedGraph.cpp

```
 1  #include <iostream>
 2  #include <string>
 3  #include <vector>
 4  #include "WeightedGraph.h"
 5  #include "WeightedEdge.h"
 6  using namespace std;
 7
 8  int main()
 9  {
10    // Vertices for graph in Figure 25.1
11    string vertices[] = {"Seattle", "San Francisco", "Los Angeles",
12      "Denver", "Kansas City", "Chicago", "Boston", "New York",
```

```
13      "Atlanta", "Miami", "Dallas", "Houston"};
14
15    // Edge array for graph in Figure 25.1
16    int edges[][3] = {
17      {0, 1, 807}, {0, 3, 1331}, {0, 5, 2097},
18      {1, 0, 807}, {1, 2, 381}, {1, 3, 1267},
19      {2, 1, 381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},
20      {3, 0, 1331}, {3, 1, 1267}, {3, 2, 1015}, {3, 4, 599},
21        {3, 5, 1003},
22      {4, 2, 1663}, {4, 3, 599}, {4, 5, 533}, {4, 7, 1260},
23        {4, 8, 864}, {4, 10, 496},
24      {5, 0, 2097}, {5, 3, 1003}, {5, 4, 533},
25        {5, 6, 983}, {5, 7, 787},
26      {6, 5, 983}, {6, 7, 214},
27      {7, 4, 1260}, {7, 5, 787}, {7, 6, 214}, {7, 8, 888},
28      {8, 4, 864}, {8, 7, 888}, {8, 9, 661},
29        {8, 10, 781}, {8, 11, 810},
30      {9, 8, 661}, {9, 11, 1187},
31      {10, 2, 1435}, {10, 4, 496}, {10, 8, 781}, {10, 11, 239},
32      {11, 8, 810}, {11, 9, 1187}, {11, 10, 239}
33    };
34
35    // 23 undirected edges in Figure 25.1
36    const int NUMBER_OF_EDGES = 46;
37
38    // Create a vector for vertices
39    vector<string> vectorOfVertices(12);
40    copy(vertices, vertices + 12, vectorOfVertices.begin());
41
42    // Create a vector for edges
43    vector<WeightedEdge> edgeVector;
44    for (int i = 0; i < NUMBER_OF_EDGES; i++)
45      edgeVector.push_back(WeightedEdge(edges[i][0],
46      edges[i][1], edges[i][2]));
47
48    WeightedGraph<string> graph1(vectorOfVertices, edgeVector);
49    cout << "The number of vertices in graph1: " << graph1.getSize();
50    cout << "\nThe vertex with index 1 is " << graph1.getVertex(1);
51    cout << "\nThe index for Miami is " << graph1.getIndex("Miami");
52
53    cout << "\nThe edges for graph1: " << endl;
54    graph1.printWeightedEdges();
55
56    // Create a graph for Figure 25.3
57    int edges2[][3] =
58    {
59      {0, 1, 2}, {0, 3, 8},
60      {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
61      {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
62      {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
63      {4, 2, 5}, {4, 3, 6}
64    }; // 14 edges in Figure 25.3
65
66    // 7 undirected edges in Figure 25.3
67    const int NUMBER_OF_EDGES2 = 14;
68
69    vector<WeightedEdge> edgeVector2;
70    for (int i = 0; i < NUMBER_OF_EDGES2; i++)
71      edgeVector2.push_back(WeightedEdge(edges2[i][0],
```

```
 72      edges2[i][1], edges2[i][2]));
 73
 74    WeightedGraph<int> graph2(5, edgeVector2); // 5 vertices in
graph2
 75
 76    cout << "The number of vertices in graph2: " << graph2.getSize();
 77    cout << "\nThe edges for graph2: " << endl;
 78    graph2.printWeightedEdges();
 79
 80    return 0;
 81  }
```

Sample output

```
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) (2, 4, 1663)
Vertex 3: (3, 4, 599) (3, 5, 1003) (3, 2, 1015)
  (3, 1, 1267) (3, 0, 1331)
Vertex 4: (4, 10, 496) (4, 5, 533) (4, 3, 599) (4, 8, 864)
  (4, 7, 1260) (4, 2, 1663)
Vertex 5: (5, 4, 533) (5, 7, 787) (5, 6, 983)
  (5, 3, 1003) (5, 0, 2097)
Vertex 6: (6, 7, 214) (6, 5, 983)
Vertex 7: (7, 6, 214) (7, 5, 787) (7, 8, 888) (7, 4, 1260)
Vertex 8: (8, 9, 661) (8, 10, 781) (8, 11, 810)
  (8, 4, 864) (8, 7, 888)
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, 8, 810) (11, 9, 1187)

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

The program creates **`graph1`** for the graph in Figure 24.1 in lines 11–46. The vertices for **`graph1`** are defined in lines 11–13. The edges for **`graph1`** are defined in lines 16–33. 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, the first row {**`0`**, **`1`**, **`807`**} represents the edge from vertex **`0`** (**`edges[0][0]`**) to vertex **`1`** (**`edges[0][1]`**) with weight **`807`** (**`edges[0][2]`**). The row {**`0`**, **`5`**, **`2097`**} represents the edge from vertex **`0`** (**`edges[2][0]`**) to vertex **`5`** (**`edges[2][1]`**) with weight

2097 (**edges[2][2]**). To create a **WeightedGraph**, you have to obtain a vector of **WeightedEdge** (lines 41–44).

Line 52 invokes the **printWeightedEdges()** function on **graph1** to display all edges in **graph1**.

The program creates a **WeightedGraph graph2** for the graph in Figure 25.3 in lines 55–71. Line 75 invokes the **printWeightedEdges()** function on **graph2** to display all edges in **graph2**.

Check point

25.3 What is the output of the following code?

```
vector<WeightedEdge> list;
list.push_back(WeightedEdge(1, 2, 3.5));
list.push_back(WeightedEdge(1, 6, 6.5));
list.push_back(WeightedEdge(1, 7, 1.5));
cout << list[0].weight << endl;
q.pop();
cout << list[1].weight << endl;
q.pop();
cout << list[2].weight << endl;
```

25.4 What is the output of the following code?

```
priority_queue<WeightedEdge, vector<WeightedEdge>,
  greater<WeightedEdge>> q;
q.push(WeightedEdge(1, 2, 3.5));
q.push(WeightedEdge(1, 6, 6.5));
q.push(WeightedEdge(1, 7, 1.5));
cout << q.top().weight << endl;
q.pop();
cout << q.top().weight << endl;
q.pop();
cout << q.top().weight << endl;
```

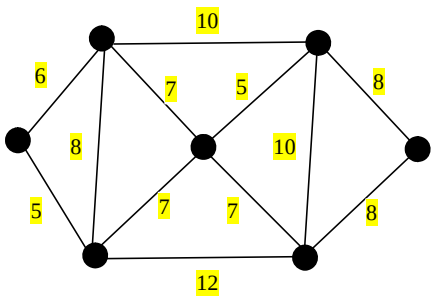
25.5 What is wrong in the following code? Fix it and show the printout.

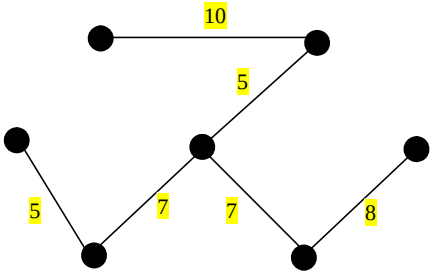
```
vector<vector<WeightedEdge>> neighbors;
neighbors[0].push_back(WeightedEdge(0, 2, 3.5));
neighbors[0].push_back(WeightedEdge(0, 6, 6.5));
neighbors[0].push_back(WeightedEdge(0, 7, 1.5));
neighbors[1].push_back(WeightedEdge(1, 0, 3.5));
neighbors[1].push_back(WeightedEdge(1, 5, 8.5));
neighbors[1].push_back(WeightedEdge(1, 8, 19.5));
cout << neighbors[0][0] << endl;
```

25.4 Minimum Spanning Trees

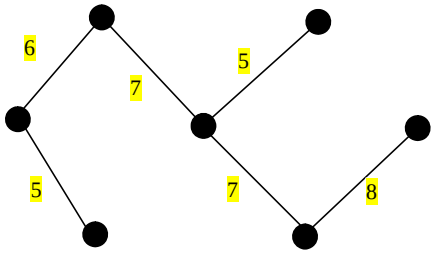
Key Point: *A minimum spanning tree of a graph is a spanning tree with the minimum total weights.*

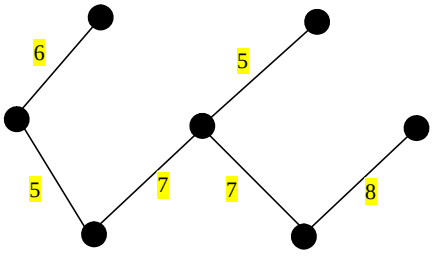
A graph may have many spanning trees. Suppose that the edges are weighted. A minimum spanning tree is a spanning tree with the minimum total weights. For example, the trees in Figures 25.5b, 25.5c, 25.5d are spanning trees for the graph in Figure 25.5a. The trees in Figures 25.5c and 25.5d are minimum spanning trees.





(a) **(b)**





(c) **(d)**

Figure 25.5

The tree 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 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.

25.4.1 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 adding a vertex with the smallest edge incident to a vertex already in the tree. The algorithm is described in Listing 25.4.

Listing 25.4 Prim's Minimum Spanning Tree Algorithm

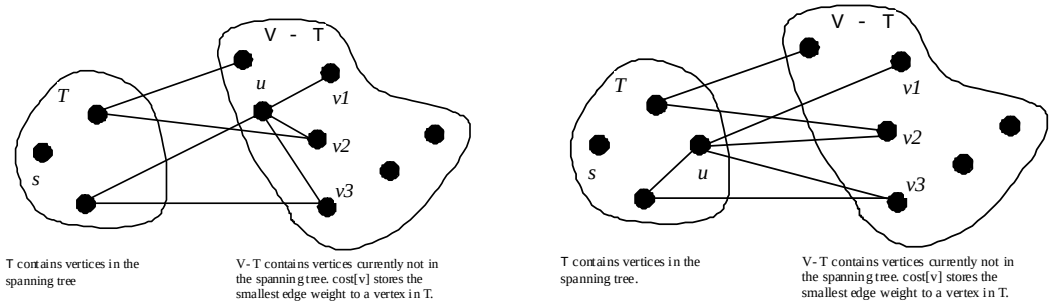
```
 Input: A connected undirected weighted G = (V, E)
Output: MST (a minimum spanning tree)
1  MST minimumSpanningTree()
2  {
3    Let T be a set for the vertices in the spanning tree;
4     Set cost[v] = infinity for each vertex v;
5     Pick an arbitrary vertex, say s and set cost[s] = 0 and parent[s]
= -1;
6
7    while (size of T < n)
8    {
9       Find u not in T with the smallest cost[u];
```

```
10        Add u to T;
11        for (each v not in T and (u, v) in E)
12        {
13          if (cost[v] > w(u, v))
14            cost[v] = w(u, v); parent[v] = u;
15        }
16    }
```

The algorithm starts by selecting an arbitrary vertex **s** and assign **0** to **cost[s]** (line 5) and infinity to **cost[v]** for all other vertices (line 4). It then adds **s** into **T** during the first iteration of the while loop (lines 7-16). After **s** is added into **T**, adjust **cost[v]** for each **v** adjacent to **s**. **cost[v]** is the smallest weight among all **v**'s edges that are adjacent to a vertex in **T**. If **cost[v]** is infinity, it indicates that **v** is not adjacent to any vertex in **T** at the moment. In each subsequent iteration of the while loop, the vertex **u** with the smallest **cost[u]** is picked and added into **T** (lines 9-10), as shown in Figure 25.6a. Afterwards, **cost[v]** is updated to **w(u, v)** and **parent[v]** to **u** for each **v** in **V-T** and **v** is adjacent to **u** if **cost[v] > w(u, v)** (lines 13-14), as shown in Figure 25.6b. Here, **w(u, v)** denotes the weight on edge (**u**, **v**).

Figure 25.6

*(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**.*



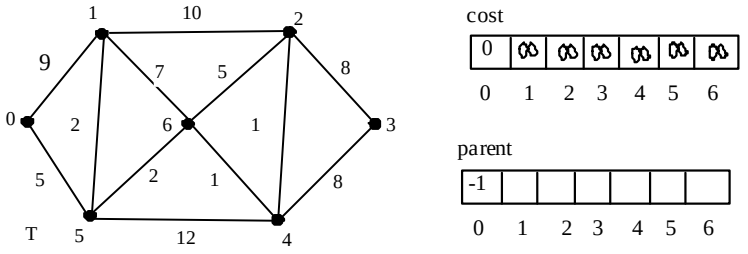
(a) Before moving u to T **(b) After moving u to T**

Consider the graph in Figure 25.7a. Suppose the algorithm selects 0 and assigns **cost[0] = 0** and **parent[0] = -1**. The algorithm adds the vertices to **T** in this order:

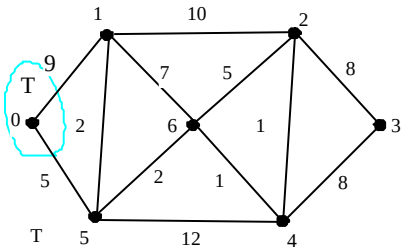
1. Add vertex **0** to **T** and set **cost[1]**, **parent[1]**, **cost[5]**, and **parent[5]**, as shown in Figure 25.7b
2. Add vertex **5** to **T**, since **cost[5]** is the smallest. Update the cost and parent values for vertices **1**, **6**, and **4**, as shown in Figure 25.7c.
3. **cost[1]** and **cost[6]** are now the smallest in **V-T**. Pick either, say vertex **1**. Add vertex **1** to **T**. Update the cost and parent for the adjacent vertices of **1** if applicable, as shown in Figure 25.7d.
4. Add vertex **6** to **T**, since **cost[6]** has the smallest weight among all edges adjacent to the vertices in **T**. Update the cost and parent for the adjacent vertices of **6** if applicable, as shown in Figure 25.7e.
5. Add vertex **4** to **T**, since **cost[4]** has the smallest weight among all edges adjacent to the vertices in **T**. Update the cost and parent for the adjacent vertices of **4** if applicable, as shown in Figure 25.7f.
6. Add vertex **2** to **T**, since **cost[2]** has the smallest weight among all edges adjacent to the vertices in **T**, as shown in Figure 25.7e.
7. Since **3** is the only vertex in **V-T**, add vertex **3** to **T**, as shown in Figure 25.7f.

Figure 25.7

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



(a)



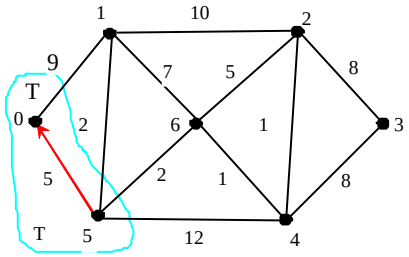
cost

0	9	∞	∞	∞	5	∞
0	1	2	3	4	5	6

parent

-1	0				0	
0	1	2	3	4	5	6

(b)



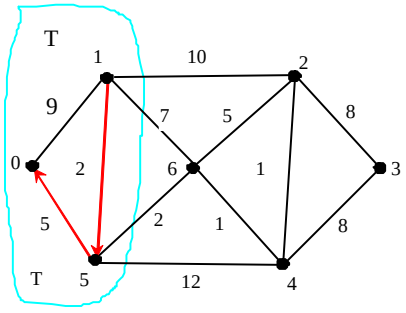
cost

0	2	∞	∞	12	5	2
0	1	2	3	4	5	6

parent

-1	5			5	0	5
0	1	2	3	4	5	6

(c)



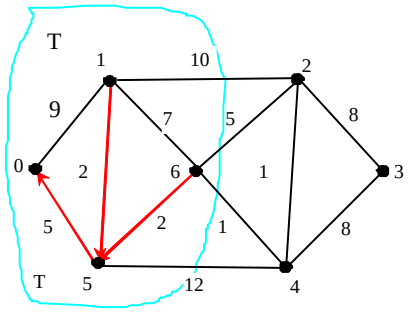
cost

0	2	10	∞	12	5	2
0	1	2	3	4	5	6

parent

-1	5	1		5	0	5
0	1	2	3	4	5	6

(d)



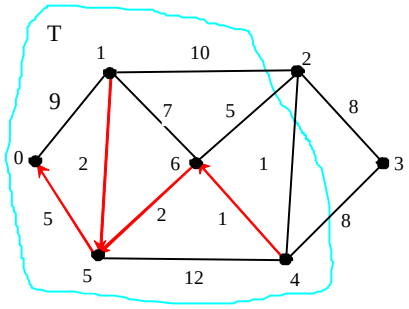
cost

0	2	10	∞	1	5	2
0	1	2	3	4	5	6

parent

-1	5	1		6	0	5
0	1	2	3	4	5	6

(e)



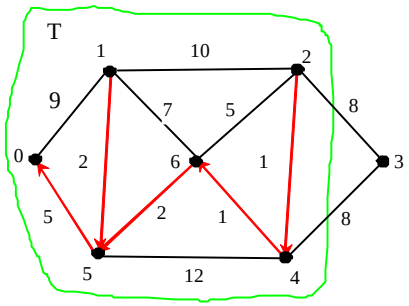
cost

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

parent

-1	5	4	4	6	0	5
0	1	2	3	4	5	6

(e)



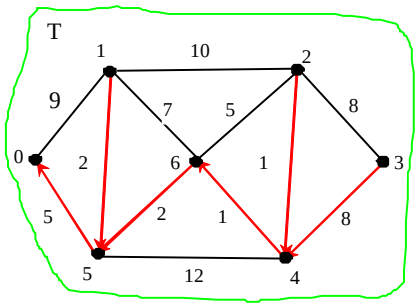
cost

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

parent

-1	5	4	4	6	0	5
0	1	2	3	4	5	6

(f)



cost

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

parent

-1	5	4	4	6	0	5
0	1	2	3	4	5	6

(g)

NOTE:

A minimum spanning tree is not unique. For example, both Figure 25.5c and Figure 25.5d are minimum spanning trees for the graph in Figure 25.7a. However, if the weights are distinct, the graph has a unique minimum spanning tree.

NOTE:

Assume that the graph is connected and undirected. If a graph is not connected or directed, the program may not work. You may modify the program to find a spanning forest for any undirected graph.

The **getMinimumSpanningTree(int v)** function is defined in the **WeightedGraph** class. It returns an instance of the **MST** class. The **MST** class is defined as a child class of **Tree**, as shown in Figure 25.8. The **Tree** class was defined in Listing 24.4. Listing 25.5 implements the **MST** class.

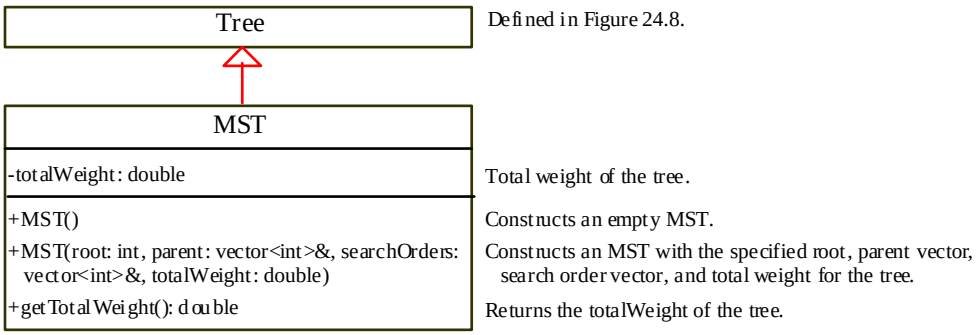


Figure 25.8

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

Listing 25.5 MST.h

```
1  #ifndef MST_H
2  #define MST_H
```

```
 3
 4  #include "Tree.h" // Defined in Listing 24.4
 5
 6  class MST : public Tree
 7  {
 8  public:
 9    // Create an empty MST
10    MST()
11    {
12    }
13
14    // Construct a tree with root, parent, searchOrders,
15    // and total weight
16    MST(int root, vector<int>& parent, vector<int>& searchOrders,
17      double totalWeight) : Tree(root, parent, searchOrders)
18    {
19      this->totalWeight = totalWeight;
20    }
21
22    // Return the total weight of the tree
23    double getTotalWeight()
24    {
25      return totalWeight;
26    }
27
28  private:
29    double totalWeight;
30  };
31  #endif
```

The **getMinimumSpanningTree** function was implemented in lines 127–185 in Listing 25.2. The **getMinimumSpanningTree()** function picks vertex 0 as the starting vertex and invokes the **getMinimumSpanningTree(0)** function to return the MST starting from 0 (line 130).

The **getMinimumSpanningTree(startingVertex)** function sets cost[startingVertex] to 0 (line 150) and cost[v] to infinity for all other vertices (lines 146-149). It then repeatedly finds the vertex with the smallest cost in V-T (lines 156-166) and adds it to T (lines 168. After a vertex u is added to **T**, the cost and parent for each vertex v in V-T adjacent to u may be adjusted (lines 171-182).

After a new vertex is added to **T** (line 193), **totalWeight** is updated (line 169). Once all vertices are added to **T**, an instance of **MST** is created (line 184).

The **MST** class extends the **Tree** class. To create an instance of **MST**, pass **root**, **parent**, **searchOrders**, and **totalWeight** (lines 200). The data fields **root**, **parent**, and **searchOrders** are defined in the **Tree** class.

Since **T** is a vector, testing whether a vertex **k** is in **T** by invoking the STL **find** algorithm (line 168) takes **O(n)** time. The testing time can be reduced to **O(1)** by introducing an array, say **isInT**, to track whether a vertex **k** is in **T**. Whenever a vertex **k** is added to **T**, set **isInT[k]** to **true**. See Programming Exercise 25.11 for implementing the algorithm using **isInT**. So, the total time for finding the vertex to be added into **T** is $O(|V|^2)$. Every time after a vertex **u** is added into **T**, the function updates **cost[v]** and **parent[v]** for each vertex **v** adjacent to **u** and for **v** in **V-T**. vertex **v**'s cost and parent are updated if **cost[v] > w(u, v)**. Each edge incident to u is examined only once. So the total time for examining the edges is O(|E|). Therefore the time complexity of this implementation is $O(|V|^2+|E|)=O(|V|^2)$. Using priority queues, we can implement the algorithm in O(|E|log|V|) time (see Programming Exercise 25.9), which is more efficient for sparse graphs.

Listing 25.6 gives a test program that displays minimum spanning trees for the graph in Figure 25.1 and the graph in Figure 25.3, respectively.

Listing 25.6 TestMinimumSpanningTree.cpp

```
 1  #include <iostream>
 2  #include <string>
 3  #include <vector>
 4  #include "WeightedGraph.h" // Defined in Listing 25.2
 5  #include "WeightedEdge.h" // Defined in Listing 25.1
 6  using namespace std;
 7
 8  // Print tree
 9  template<typename T>
10  void printTree(Tree& tree, vector<T>& vertices)
11  {
12    cout << "\nThe root is " << vertices[tree.getRoot()];
13    cout << "\nThe edges are:";
14    for (unsigned i = 0; i < vertices.size(); i++)
15    {
16      if (tree.getParent(i) != -1)
17        cout << " (" << vertices[i] << ", "
18          << vertices[tree.getParent(i)] << ")";
19    }
20    cout << endl;
21  }
```

```
22
23  int main()
24  {
25    // Vertices for graph in Figure 25.1
26    string vertices[] = {"Seattle", "San Francisco", "Los Angeles",
27      "Denver", "Kansas City", "Chicago", "Boston", "New York",
28      "Atlanta", "Miami", "Dallas", "Houston"};
29
30    // Edge array for graph in Figure 25.1
31    int edges[][3] = {
32      {0, 1, 807}, {0, 3, 1331}, {0, 5, 2097},
33      {1, 0, 807}, {1, 2, 381}, {1, 3, 1267},
34      {2, 1, 381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},
35      {3, 0, 1331}, {3, 1, 1267}, {3, 2, 1015}, {3, 4, 599},
36        {3, 5, 1003},
37      {4, 2, 1663}, {4, 3, 599}, {4, 5, 533}, {4, 7, 1260},
38        {4, 8, 864}, {4, 10, 496},
39      {5, 0, 2097}, {5, 3, 1003}, {5, 4, 533},
40        {5, 6, 983}, {5, 7, 787},
41      {6, 5, 983}, {6, 7, 214},
42      {7, 4, 1260}, {7, 5, 787}, {7, 6, 214}, {7, 8, 888},
43      {8, 4, 864}, {8, 7, 888}, {8, 9, 661},
44        {8, 10, 781}, {8, 11, 810},
45      {9, 8, 661}, {9, 11, 1187},
46      {10, 2, 1435}, {10, 4, 496}, {10, 8, 781}, {10, 11, 239},
47      {11, 8, 810}, {11, 9, 1187}, {11, 10, 239}
48    };
49
50    // 23 undirected edges in Figure 25.1
51    const int NUMBER_OF_EDGES = 46;
52
53    // Create a vector for vertices
54    vector<string> vectorOfVertices(12);
55    copy(vertices, vertices + 12, vectorOfVertices.begin());
56
57    // Create a vector for edges
58    vector<WeightedEdge> edgeVector;
59    for (int i = 0; i < NUMBER_OF_EDGES; i++)
60      edgeVector.push_back(WeightedEdge(edges[i][0],
61      edges[i][1], edges[i][2]));
62
63    WeightedGraph<string> graph1(vectorOfVertices, edgeVector);
64    MST tree1 = graph1.getMinimumSpanningTree();
65    cout << "The spanning tree weight is " << tree1.getTotalWeight();
66    printTree<string>(tree1, graph1.getVertices());
67
68    // Create a graph for Figure 25.3
69    int edges2[][3] =
70    {
71      {0, 1, 2}, {0, 3, 8},
72      {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
73      {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
74      {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
75      {4, 2, 5}, {4, 3, 6}
76    }; // 14 edges in Figure 25.3
77
78    // 7 undirected edges in Figure 25.3
79    const int NUMBER_OF_EDGES2 = 14;
80
```

```
 81    vector<WeightedEdge> edgeVector2;
 82    for (int i = 0; i < NUMBER_OF_EDGES2; i++)
 83      edgeVector2.push_back(WeightedEdge(edges2[i][0],
 84      edges2[i][1], edges2[i][2]));
 85
 86    WeightedGraph<int> graph2(5, edgeVector2); // 5 vertices in
graph2
 87    MST tree2 = graph2.getMinimumSpanningTree();
 88    cout << "\nThe spanning tree weight is " <<
tree2.getTotalWeight();
 89    printTree<int>(tree2, graph2.getVertices());
 90
 91    tree2.printTree();
 92
 93    return 0;
 94  }
```

Sample output

```
The spanning tree weight is 6513
The root is Seattle
The edges are:  (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)

The spanning tree weight is 14
The root is 0
The edges are: (0, 1) (3, 2) (1, 3) (2, 4)
```

The program creates a weighted graph for Figure 24.1 in line 60. It then invokes **`getMinimumSpanningTree()`** (line 61) to return a **`MST`** that represents a minimum spanning tree for the graph. Invoking **`getTotalWeight()`** on the MST object returns the total weight of the minimum spanning tree. The **`printTree()`** function (lines 9–20) on the **`MST`** object displays the edges in the tree. Note that **`MST`** is a subclass of **`Tree`**.

The graphical illustration of the minimum spanning tree is shown in Figure 25.9. 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.

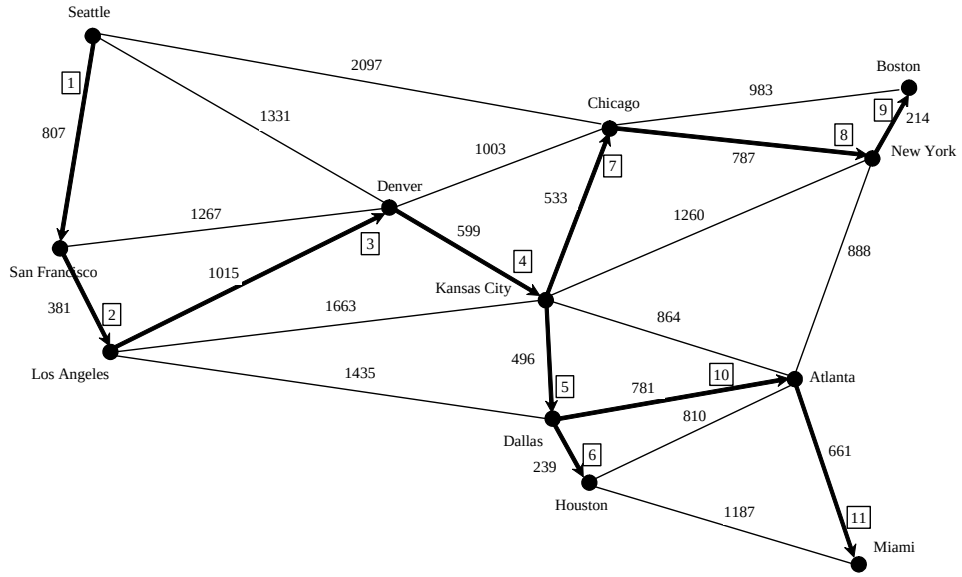
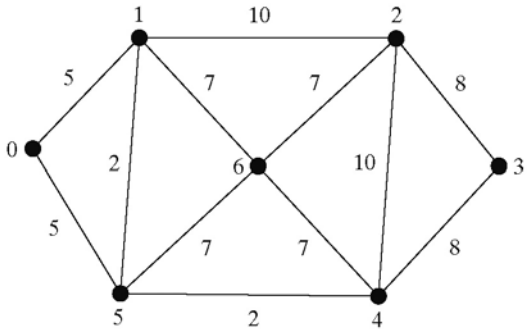


Figure 25.9

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

Check point

25.5 Find a minimum spanning tree for the following graph.



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

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

25.8 What happens to the `getMinimumSpanningTree()` function 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()` function.

<end check point>

25.5 Finding Shortest Paths

<key point>

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

<end key point>

Given a graph with nonnegative weights on the edges, a well-known algorithm for finding a single-source shortest path 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 *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]** for other vertices otherwise. Let **V** denote all vertices in the graph and **T** denote the set of the vertices whose costs are known. Initially, The algorithm repeatedly finds a vertex **u** in **V – T** with the smallest **cost[u]** and moves **u** to **T**. Afterwards, adjust the cost and parent for each vertex **v** in **V-T** if **v** is adjacent to **u** and **cost[v] > cost[u] + w(u, v)**.

*<key term>*shortest path

*<key term>*Dijkstra's algorithm

*<key term>*single-source shortest path

The algorithm is described in Listing 25.7.

Listing 25.7 Dijkstra's Single-Source Shortest-Path Algorithm

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

 1  ShortestPathTree getShortestPath(s)
 2  {
 3    Let T be a set that contains the vertices whose
 4      paths to s are known;
 5    Set cost[v] = infinity for all vertices;
 6    Set cost[s] = 0 and parent[s] = -1;
 7
 8    while (size of T < n)
 9    {
10      Find u not in T with the smallest cost[u];
```

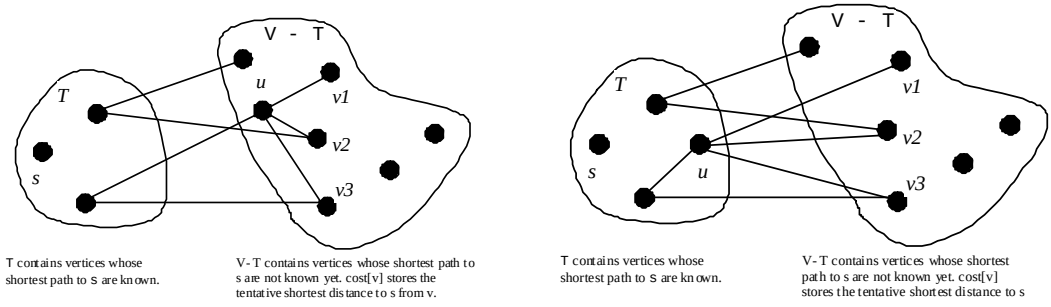
```
11      Add u to T;
12      for (each v not in T and (u, v) in E)
13      {
14        if (cost[v] > cost[u] + w(u, v))
15          cost[v] = cost[u] + w(u, v); parent[v] = u;
16      }
17    }
```

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` and `parent[s]` to `-1` and `cost[v]` to infinity for all other vertices. 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`. The algorithm continuously adds a vertex (say `u`) from `V – T` into `T` with smallest `cost[u]` (lines 10-11), as shown in Figure 25.10a. 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 12-16).

Figure 25.10

(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`.

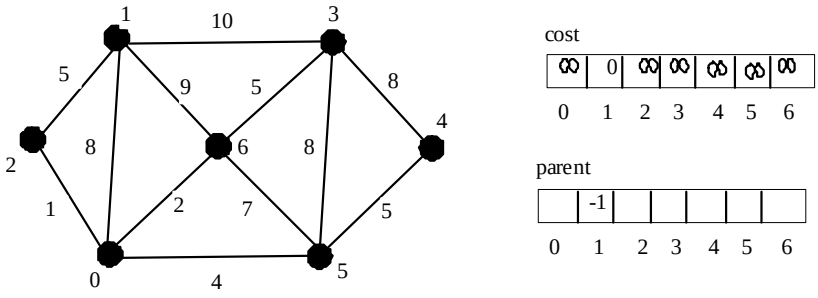


(a) Before moving u to T **(b) After moving u to T**

Let us illustrate Dijkstra's algorithm using the graph in Figure 25.10a. Suppose the source vertex is `1`. Therefore, `cost[1] = 0` and `parent[1] = -1`, and the costs for all other vertices are initially ∞, as shown in Figure 25.10b.

Figure 25.10

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

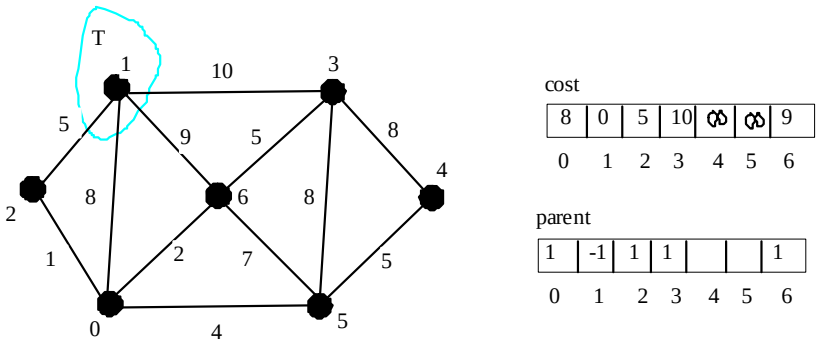


(a) **(b)**

Now `T` contains {`1`}. Vertex `1` is the one in `V-T` with the smallest cost, so add `1` to `T`, as shown in Figure 25.12 and update the cost and parent for vertices in `V-T` and adjacent to `1` if applicable. The cost and parent for vertices 0, 2, 3, and 6 are now updated, as shown in Figure 25.12b.

Figure 25.12

Now vertex `1` is in set `T`.

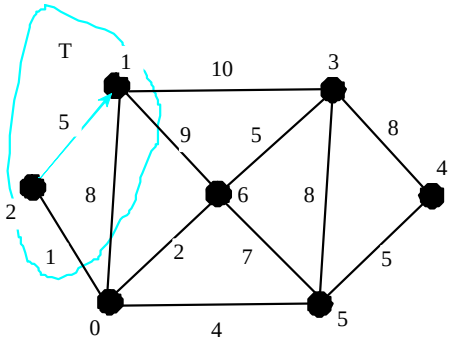


(a) **(b)**

Now `T` contains {`1`}. Vertex `2` is the one in `V-T` with the smallest cost, so add `2` to `T`, as shown in Figure 25.13 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`.

Figure 25.13

Now vertices `1` and `2` are in set `T`.



(a)

cost

6	0	5	10	∞	∞	9
0	1	2	3	4	5	6

parent

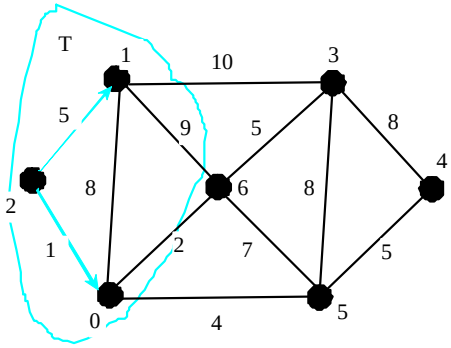
2	-1	1	1			1
0	1	2	3	4	5	6

(b)

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 25.14 and update the cost and parent for vertices in `V-T` and adjacent to `0` if applicable. `cost[5]` is now updated to `10` and its parent is set to `0` and `cost[6]` is now updated to `8` and its parent is set to `0`.

Figure 25.14

Now vertices {`1`, `2`, `0`} are in set `T`.



(a)

cost

6	0	5	10	∞	10	8
0	1	2	3	4	5	6

parent

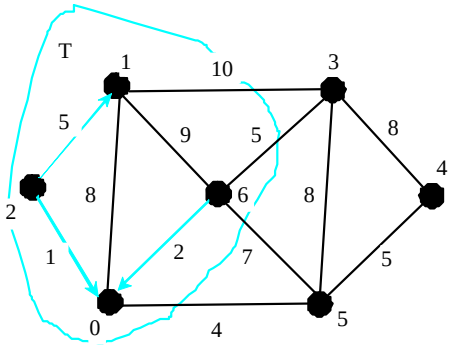
2	-1	1	1		0	1
0	1	2	3	4	5	6

(b)

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 25.15 and update the cost and parent for vertices in `V-T` and adjacent to `6` if applicable.

Figure 25.15

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



(a)

cost

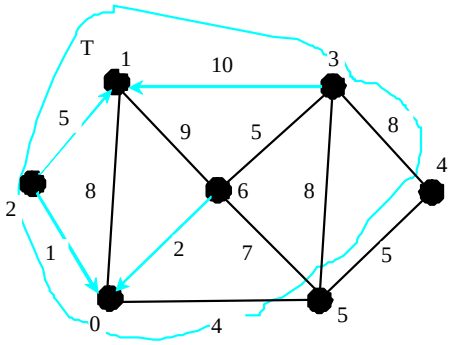
6	0	5	10	∞	10	8
0	1	2	3	4	5	6

parent

2	-1	1	1		0	1
0	1	2	3	4	5	6

(b)

Now `T` contains {`1`, `2`, `0`, `6`}. Vertex `3` or `5` is 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 25.16 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`.



(a)

cost

6	0	5	10	18	10	8
0	1	2	3	4	5	6

parent

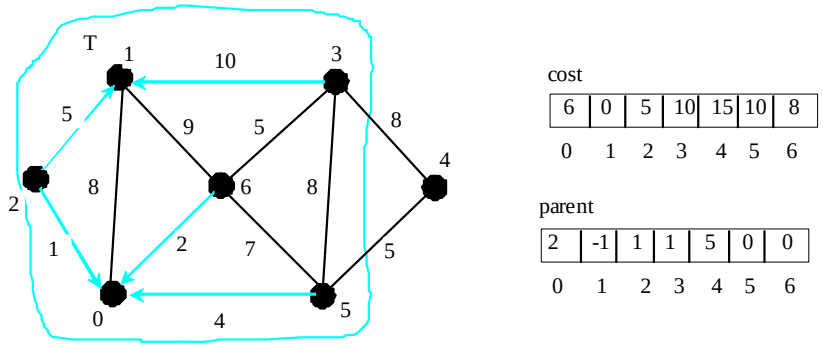
2	-1	1	1	3	0	0
0	1	2	3	4	5	6

(b)

Figure 25.16

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 25.17 and update the cost and parent for vertices in `V-T` and adjacent to `5` if applicable. `cost[4]` is now updated to `10` and its parent is set to `5`.

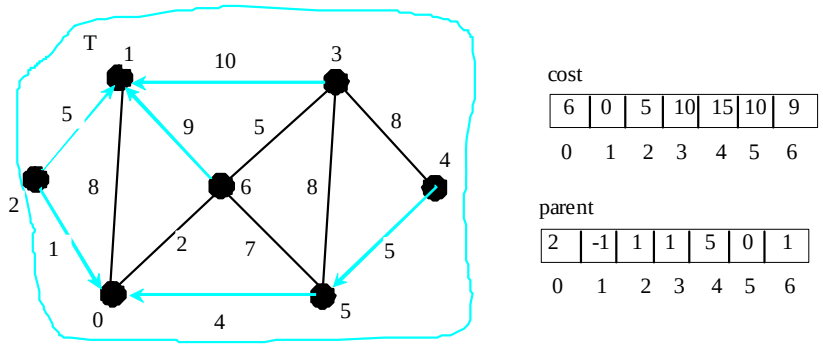


(a) **(b)**

Figure 25.17

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

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 25.18 and update the cost and parent for vertices in `V-T` and adjacent to `4` if applicable.



(a) **(b)**

Figure 25.18

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

As you 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 **Tree** class, as shown in Figure 25.19. Listing 25.8 implements the **ShortestPathTree** class

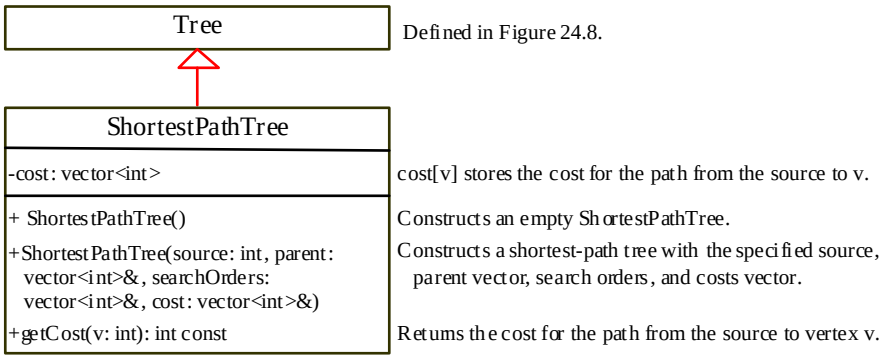


Figure 25.19
***ShortestPathTree** extends **Tree**.*

Listing 25.8 ShortestPathTree.h

```
1  #ifndef SHORTESTPATHTREE_H
2  #define SHORTESTPATHTREE_H
3
4  #include "Tree.h" // Defined in Listing 24.4
5
6  class ShortestPathTree : public Tree
7  {
8  public:
9    // Create an empty ShortestPathTree
10   ShortestPathTree()
11   {
12   }
13
14   // Construct a tree with root, parent, searchOrders,
15   // and cost
16   ShortestPathTree(int root, vector<int>& parent,
17       vector<int>& searchOrders, vector<double>& cost)
18     : Tree(root, parent, searchOrders)
19   {
20     this->cost = cost;
21   }
22
23   // Return the cost for the path from the source to vertex v.
24   double getCost(int v)
25   {
26     return cost[v];
27   }
28
29 private:
30   vector<double> cost;
31 };
32 #endif
```

The **getShortestPath(int sourceVertex)** function was implemented in lines 160–210 in Listing 25.2, WeightedGraph.h. The function first sets **cost[sourceVertex]** to **0** (line 176) and **cost[v]** to infinity for all other vertices (lines 171-176).

The **parent** array stores the parent for each vertex to enable us to identify the path from the source to all other vertices in the graph (line 167). The parent of the root is set to -1 (line 168).

T is a set that stores the vertices whose path has been found (line 164). The function expands **T** by performing the following operations:

1. Find the vertex **u** with the smallest **cost[u]** (lines 182-192) and add it into **T** (line 196). If such a vertex is not found, the graph is not connected and the source vertex cannot reach all vertices in the graph (line 194).
2. 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 199-207).

Once all reachable vertices from **s** are added to **T**, an instance of **ShortestPathTree** is created (line 211).

The **ShortestPathTree** class extends the **Tree** class. To create an instance of **ShortestPathTree**, pass **sourceVertex**, **parent**, **searchOrders**, and **cost** (lines 211). **sourceVertex** becomes the root in the tree. The data fields **root**, **parent**, and **searchOrders** are defined in the **Tree** class.

As discussed early in the **getMinimumSpanningTree** function, we can reduce the time to **O(1)** for testing whether a vertex **k** is in **T** by introducing an array, say **isInT**, to track whether a vertex **k** is in **T**. Whenever a vertex **k** is added to **T**, set **isInT[k]** to **true**. It takes **O(|V|)** time to find a **u** with the smallest **cost[u]** using a linear search (lines 253-260). So, the total time for finding the vertex to be added into **T** is $O(|V|^2)$. Every time after a vertex **u** is added into **T**, the function updates **cost[v]** and **parent[v]** for each vertex **v** adjacent to **u** and for **v** in **V-T**. vertex **v**'s cost and parent are updated if **cost[v] > w(u, v)**. Each edge incident to u is examined only once. So the total time for examining the edges is O(|E|). Therefore the time complexity of this implementation is $O(|V|^2+|E|)=O(|V|^2)$. Using

priority queues, we can implement the algorithm in O(|E|log|V|) time (see Programming Exercise 25.10), which is more efficient for sparse graphs.

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.

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 25.20.

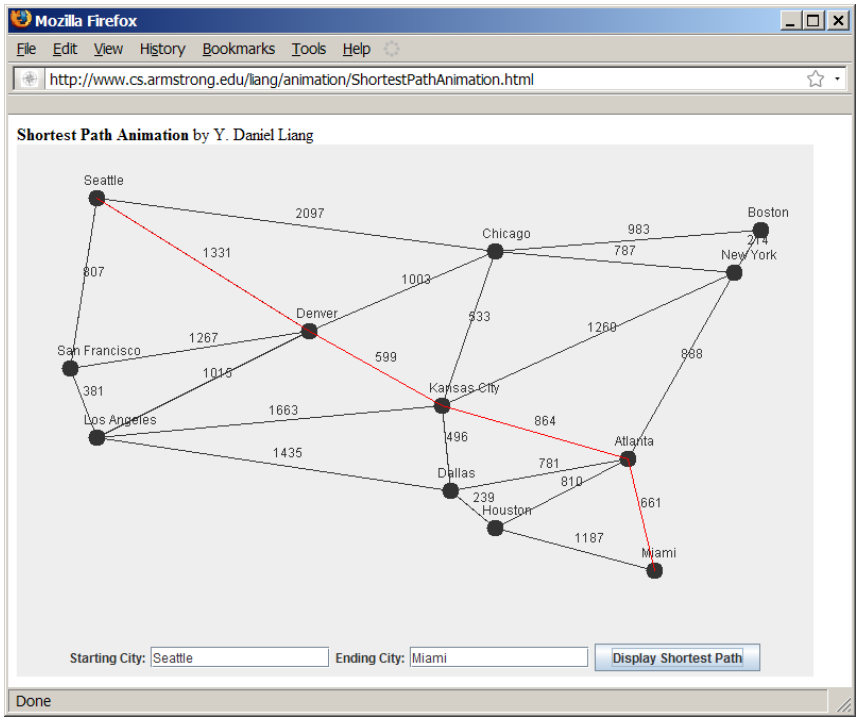


Figure 25.20

The animation tool displays a shortest path between two cities.

Listing 25.9 gives a test program that displays the shortest paths from Chicago to all other cities in Figure 25.1 and the shortest paths from vertex `3` to all vertices for the graph in Figure 25.3a, respectively.

Listing 25.9 TestShortestPath.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include "WeightedGraph.h" // Defined in Listing 25.2
#include "WeightedEdge.h" // Defined in Listing 25.1
using namespace std;

// Print paths from all vertices to the source
template<typename T>
void printAllPaths(ShortestPathTree& tree, vector<T> vertices)
{
  cout << "All shortest paths from " <<
    vertices[tree.getRoot()] << " are:" << endl;
  for (unsigned i = 0; i < vertices.size(); i++)
  {
    cout << "To " << vertices[i] << ": ";

    // Print a path from i to the source
    vector<int> path = tree.getPath(i);
    for (int j = path.size() - 1; j >= 0; j--)
    {
      cout << vertices[path[j]] << " ";
    }

    cout << "(cost: " << tree.getCost(i) << ")" << endl;
  }
}

int main()
{
  // Vertices for graph in Figure 24.1
  string vertices[] = {"Seattle", "San Francisco", "Los Angeles",
    "Denver", "Kansas City", "Chicago", "Boston", "New York",
    "Atlanta", "Miami", "Dallas", "Houston"};

  // Edge array for graph in Figure 24.1
  int edges[][3] = {
    {0, 1, 807}, {0, 3, 1331}, {0, 5, 2097},
    {1, 0, 807}, {1, 2, 381}, {1, 3, 1267},
    {2, 1, 381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},
    {3, 0, 1331}, {3, 1, 1267}, {3, 2, 1015}, {3, 4, 599},
      {3, 5, 1003},
    {4, 2, 1663}, {4, 3, 599}, {4, 5, 533}, {4, 7, 1260},
      {4, 8, 864}, {4, 10, 496},
    {5, 0, 2097}, {5, 3, 1003}, {5, 4, 533},
      {5, 6, 983}, {5, 7, 787},
    {6, 5, 983}, {6, 7, 214},
    {7, 4, 1260}, {7, 5, 787}, {7, 6, 214}, {7, 8, 888},
    {8, 4, 864}, {8, 7, 888}, {8, 9, 661},
      {8, 10, 781}, {8, 11, 810},
    {9, 8, 661}, {9, 11, 1187},
    {10, 2, 1435}, {10, 4, 496}, {10, 8, 781}, {10, 11, 239},
    {11, 8, 810}, {11, 9, 1187}, {11, 10, 239}
  };

  // 23 undirected edges in Figure 25.1
```

```
 57    const int NUMBER_OF_EDGES = 46;
 58
 59    // Create a vector for vertices
 60    vector<string> vectorOfVertices(12);
 61    copy(vertices, vertices + 12, vectorOfVertices.begin());
 62
 63    // Create a vector for edges
 64    vector<WeightedEdge> edgeVector;
 65    for (int i = 0; i < NUMBER_OF_EDGES; i++)
 66      edgeVector.push_back(WeightedEdge(edges[i][0],
 67      edges[i][1], edges[i][2]));
 68
 69    WeightedGraph<string> graph1(vectorOfVertices, edgeVector);
 70    ShortestPathTree tree = graph1.getShortestPath(5);
 71    printAllPaths<string>(tree, graph1.getVertices());
 72
 73    // Create a graph for Figure 25.1
 74    int edges2[][3] =
 75    {
 76      {0, 1, 2}, {0, 3, 8},
 77      {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
 78      {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
 79      {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
 80      {4, 2, 5}, {4, 3, 6}
 81    }; // 7 undirected edges in Figure 25.3
 82
 83    // 7 undirected edges in Figure 25.3
 84    const int NUMBER_OF_EDGES2 = 14;
 85
 86    vector<WeightedEdge> edgeVector2;
 87    for (int i = 0; i < NUMBER_OF_EDGES2; i++)
 88      edgeVector2.push_back(WeightedEdge(edges2[i][0],
 89      edges2[i][1], edges2[i][2]));
 90
 91    WeightedGraph<int> graph2(5, edgeVector2); // 5 vertices in
graph2
 92    ShortestPathTree tree2 = graph2.getShortestPath(3);
 93    printAllPaths<int>(tree2, graph2.getVertices());
 94
 95    return 0;
 96  }
```

Sample output

```
All shortest paths from Chicago are:
To Seattle: Chicago Seattle (cost: 2097)
To San Francisco: Chicago Denver San Francisco (cost: 2270)
To Los Angeles: Chicago Denver Los Angeles (cost: 2018)
To Denver: Chicago Denver (cost: 1003)
To Kansas City: Chicago Kansas City (cost: 533)
To Chicago: Chicago (cost: 0)
To Boston: Chicago Boston (cost: 983)
To New York: Chicago New York (cost: 787)
To Atlanta: Chicago Kansas City Atlanta (cost: 1397)
To Miami: Chicago Kansas City Atlanta Miami (cost: 2058)
To Dallas: Chicago Kansas City Dallas (cost: 1029)
To Houston: Chicago Kansas City Dallas Houston (cost: 1268)

All shortest paths from 3 are:
To 0: 3 1 0 (cost: 5)
```

```
To 1: 3 1 (cost: 3)
To 2: 3 2 (cost: 4)
To 3: 3 (cost: 0)
To 4: 3 4 (cost: 6)
```

The program creates a weighted graph for Figure 25.1 in line 67. It then invokes the **`getShortestPath(5)`** function to return a **`ShortestPathTree`** object that contains all shortest paths from vertex **`5`** (i.e., Chicago) (line 68). The **`printAllPaths`** function displays all the paths (line 69).

The graphical illustration of all shortest paths from **`Chicago`** is shown in Figure 25.20. 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`**.

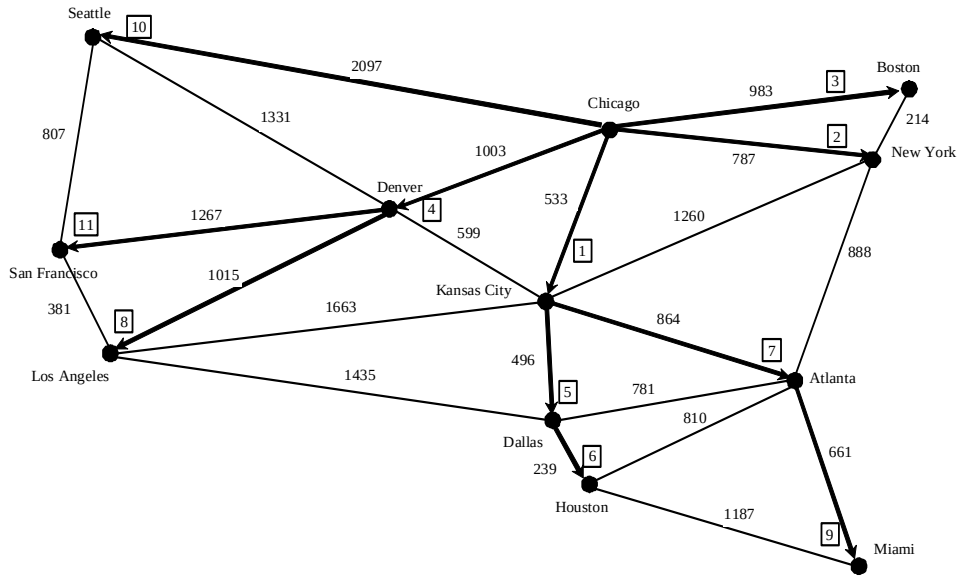


Figure 25.20

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

Check point

25.9 Trace Dijkstra's algorithm for finding shortest paths from Boston to all other cities in Figure 25.1.

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

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

25.12 What happens to the `getShortestPath()` function in `WeightedGraph` if the source vertex cannot reach all vertieces in the graph? Verify your answer by writing a test program that creates an unconnected graph and invoke the `getShortestPath()` function. How do you fix the problem by obtaining a partial shortest path tree?

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

25.14 Assume that the graph is connected; will the `getShortestPath` function find the shortest paths correctly if lines 228-231 in `WeightedGraph` are deleted?

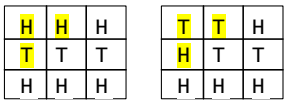
<end check point>

25.6 Case Study: The Weighted Nine Tail Problem

Key Point: *The weighted nine tails problem can be reduced to the weighted shortest path problem.*

Section 24.8 presented the nine tail 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 tail problem is to find the minimum number of the moves that lead to all coins being face down. Each move flips a head coin and its neighbors. The weighted nine tail problem assigns the number of flips as a weight on each move. For example, you can move from the coins in Figure 25.21a to Figure 25.21b by flipping the three coins. So the weight for this move is **3**.



(a) (b)

Figure 25.21

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

The weighted nine tail problem is to find the minimum number of flips that lead to all coins face down. The 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 we created a class **`NineTailModel`** in Section 24.8 for modeling the nine tail problem. We now create a new class named **`WeightedNineTailModel`** that extends **`NineTailModel`**, as shown in Figure 25.22.

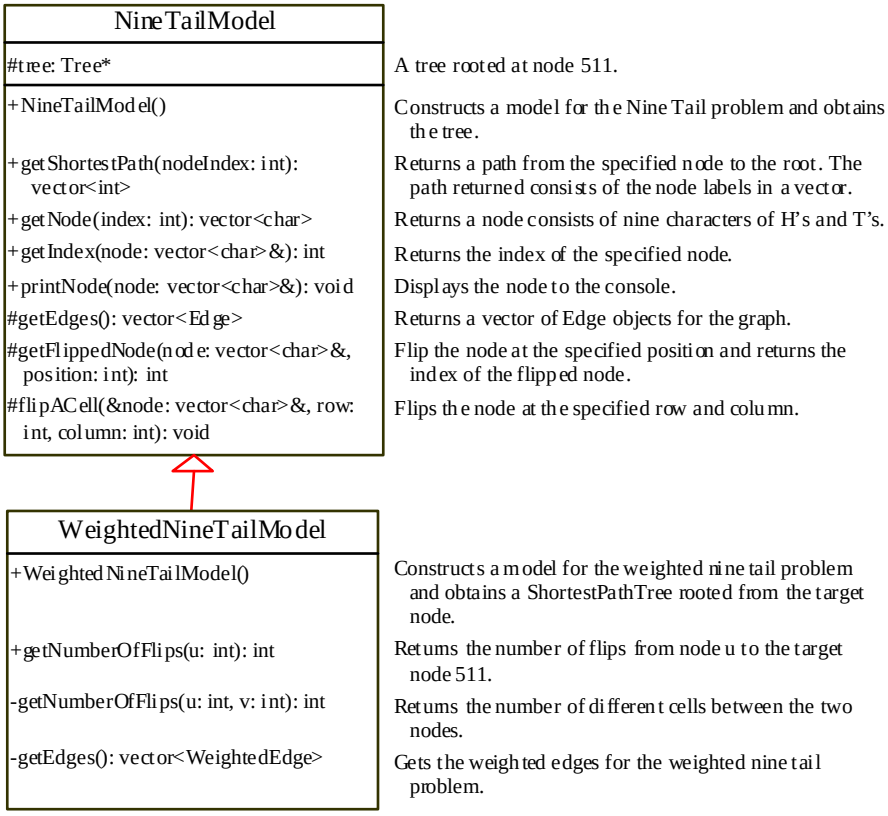


Figure 25.22

`WeightedNineTailModel` *extends* ***`NineTailModel`***.

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 functions **`getShortestPath(int)`**, **`getNode(int)`**, **`getIndex(node)`**, **`getFlippedNode(node, position)`**,

and **flipACell(&node, row, column)** defined in **NineTailModel** are inherited in **WeightedNineTailModel**. The **getNumberOfFlips(int u)** function returns the number of flips from node **u** to the target node.

Listing 25.10 implements **WeightedNineTailModel**.

Listing 25.10 WeightedNineTailModel.h

```
1  #ifndef WEIGHTEDNINETAILMODEL_H
2  #define WEIGHTEDNINETAILMODEL_H
3
4  #include "NineTailModel.h" // Defined in Listing 24.9
5  #include "WeightedGraph.h" // Defined in Listing 25.2
6
7  using namespace std;
8
9  class WeightedNineTailModel : public NineTailModel
10 {
11 public:
12   // Construct a model for the Nine Tail problem
13   WeightedNineTailModel();
14
15   // Get the number of flips from the target to u
16   int getNumberOfFlips(int u);
17
18 private:
19   // Return a vector of Edge objects for the graph
20   // Create edges among nodes
21   vector<WeightedEdge> getEdges();
22
23   // Get the number of flips from u to v
24   int getNumberOfFlips(int u, int v);
25 };
26
27 WeightedNineTailModel::WeightedNineTailModel()
28 {
29   // Create edges
30   vector<WeightedEdge> edges = getEdges();
31
32   // Build a graph
33   WeightedGraph<int> graph(NUMBER_OF_NODES, edges);
34
35   // Build a shortest path tree rooted at the target node
36   tree = new ShortestPathTree(graph.getShortestPath(511));
37 }
38
39 vector<WeightedEdge> WeightedNineTailModel::getEdges()
40 {
41   vector<WeightedEdge> edges;
42
43   for (int u = 0; u < NUMBER_OF_NODES; u++)
44   {
```

```
45      for (int k = 0; k < 9; k++)
46      {
47        vector<char> node = getNode(u);
48        if (node[k] == 'H')
49        {
50          int v = getFlippedNode(node, k);
51          int numberOfFlips = getNumberOfFlips(u, v);
52          // Add edge (v, u) for a legal move from node u to node v
53          // with weight numberOfFlips
54          edges.push_back(WeightedEdge(v, u, numberOfFlips));
55        }
56      }
57    }
58
59    return edges;
60  }
61
62  int WeightedNineTailModel::getNumberOfFlips(int u, int v)
63  {
64    vector<char> node1 = getNode(u);
65    vector<char> node2 = getNode(v);
66
67    int count = 0; // Count the number of different cells
68    for (unsigned i = 0; i < node1.size(); i++)
69      if (node1[i] != node2[i]) count++;
70
71    return count;
72  }
73
74  int WeightedNineTailModel::getNumberOfFlips(int u)
75  {
76    return static_cast<ShortestPathTree*>(tree)->getCost(u);
77  }
78  #endif
```

WeightedNineTailModel extends **NineTailModel** to build a **WeightedGraph** to model the weighted nine tail problem (line 9). For each node **u**, the **getEdges()** function finds a flipped node **v** and assigns the number of flips as the weight for edge (**u**, **v**) (line 50). The **getNumberOfFlips(int u, int v)** function returns the number of flips from node **u** to node **v** (lines 62–72). The number of flips is the number of the different cells between the two nodes (line 69).

The **WeightedNineTailModel** constructs a **WeightedGraph** (line 33) and obtains a **ShortestPathTree** rooted at the target node **511** (line 36). It then assigns it to **tree*** (line 36). **tree**, of the **Tree*** type, is a protected data field defined **NineTailModel**. The functions defined in **NineTailModel** use the **tree** property. Note that **ShortestPathTree** is a child class of **Tree**.

The **getNumberOfFlips(int u)** function (lines 62–72) 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)** function defined in the **ShortestPathTree** class (line 76).

Listing 25.11 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 25.11 WeightedNineTail.cpp

```
 1  #include <iostream>
 2  #include <vector>
 3  #include "WeightedNineTailModel.h"
 4  using namespace std;
 5
 6  int main()
 7  {
 8    // Prompt the user to enter nine coins H and T's
 9    cout << "Enter an initial nine coin H and T's: ";
10    vector<char> initialNode(9);
11
12    for (int i = 0; i < 9; i++)
13      cin >> initialNode[i];
14
15    cout << "The steps to flip the coins are " << endl;
16    WeightedNineTailModel model;
17    vector<int> path =
18      model.getShortestPath(model.getIndex(initialNode));
19
20    for (unsigned i = 0; i < path.size(); i++)
21      model.printNode(model.getNode(path[i]));
22
23    cout << "The number of flips is " <<
24      model.getNumberOfFlips(model.getIndex(initialNode)) << endl;
25
26    return 0;
27  }
```

Sample output

```
Enter an initial nine coin H's and T's:
HHH
TTT
HHH

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 **`H`**'s and **`T`**'s in lines 9–13, creates a model (line 16), obtains a shortest path from the initial node to the target node (lines 17–18), displays the nodes in the path (lines 20–21), and invokes **`getNumberOfFlips`** to get the number of flips needed to reach to the target node (lines 23–24).

Check point

25.15 Why is the **`tree`** data field in **`NineTailModel`** in Listing 24.9 defined protected?

25.16 How are the nodes created for the graph in **`WeightedNineTailModel`**?

25.17 How are the edges created for the graph in **`WeightedNineTailModel`**?

Key Terms

- **Dijkstra's algorithm**
- **edge-weighted graph**
- **minimum spanning tree**
- **Prim's algorithm**
- **shortest path**
- **single-source shortest path**
- **vertex-weighted graph**

Chapter Summary

1. Often a priority queue is used to represent weighted edges, so that the minimum-weight edge can be retrieved first.
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 search from the source vertex and keeps finding vertices that have the shortest path to the source until all vertices are found.

Quiz

Do the quiz for this chapter online at www.cs.armstrong.edu/liang/cpp3e/quiz.html.

Programming Exercises

25.1* (*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 a MST using Kruskal's algorithm.

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

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

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

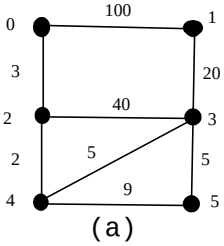
25.5* (*Prove or disprove*) The conjecture is that both **`NineTailModel`** and **`WeightedNineTailModel`** result in the same shortest path. Write a program to prove or disprove it.

(*Hint*: Add a new function named **`depth(int v)`** in the **`Tree`** class to return the depth of the **`v`** in the tree. Let **`tree1`** and **`tree2`** denote the trees 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.)

25.6** (*Traveling salesman problem*) The traveling salesman problem (TSP) is to find a shortest round-trip route that visits each city exactly once and then returns to the starting city. The problem is equivalent to finding a shortest Hamiltonian cycle. Add the following function in the **`WeightedGraph`** class:

```
// Return a shortest cycle
vector<int> getShortestHamiltonianCycle()
```

25.7* (*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 specifies edges with the format **`u1, v1, w1 | u2, v2, w2 | ...`**. Each triplet describes an edge and its weight. Figure 25.23 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.



(a)

```
File
6
0, 1, 100 | 0, 2, 3
1, 3, 20
2, 3, 40 | 2, 4, 2
3, 4, 5 | 3, 5, 5
4, 5, 9
```

(b)

Figure 25.23

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 a file, create an instance **g** of **WeightedGraph**, invoke **g.printWeightedEdges()** to display all edges, invoke **getMinimumSpanningTree()** to obtain an instance **tree** of **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:

Sample output

```
Enter a file name: c:\exercise\Exercise25_7.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)
```

25.8* (*Create a file for graph*) Modify Listing 25.3, TestWeightedGraph.cpp, to create a file for representing **graph1**. The file format is described in Exercise 25.7. Create the file from the array defined in lines 16–33 in Listing 25.3. 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
```

*****25.9** (*Alternative version of Prim's algorithm*) An alternative version of the Prim's algorithm can be described as follows:

```
Input: A connected undirected weighted G = (V, E)
```

```
Output: MST (a minimum spanning tree)
 1  MST minimumSpanningTree()
 2  {
 3    Let T be a set for the vertices in the spanning tree;
 4    Initially, add the starting vertex to T;
 5
 6    while (size of T < n)
 7    {
 8      Find u in T and v in V - T with the smallest weight
 9        on the edge (u, v), as shown in Figure 25.24;
10      Add v to T and set parent[v] = u;
11    }
12  }
```

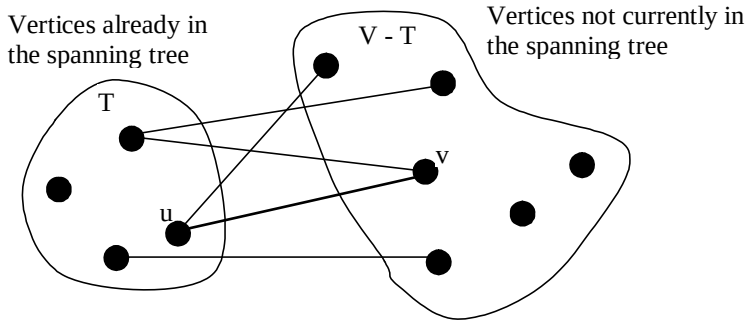


Figure 25.24
Find a vertex u in T that connects a vertex v in V – T with the smallest weight.

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 adjacent to a vertex in **T** with the smallest weight on the edge. For example, five edges connect vertices in **T** and **V – T**, as shown in Figure 25.y, (**u**, **v**) is the one with the smallest weight. This algorithm can be implemented using priority queues to achieve the O(|E|log|V|) time complexity. Use Listing 25.6 TestMinimumSpanningTree.cpp to test your new algorithm.

***25.10** (*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 non-negative 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;
 5    for (each u in V – T)
 6       cost[u] = infinity;
 7
 8    while (size of T < n)
      {
 9      Find v in V – T with the smallest cost[u] + w(u, v) value
10        among all u in T;
11      Add v to T and set cost[v] = cost[u] + w(u, v);
12      parent[v] = u;
13    }
14  }
```

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`**. The shortest path algorithm given in the text coninously update 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`**. This algorithm can be implemented using priority queues to achieve the O(|E|log|V|) time complexity. Use Listing 25.9 TestShortestPath.cpp to test your new algorithm.

***25.11** (*Test if a vertex in T effeciently*) Since **T** is implemented using a list in the **getMinimumSpanningTree** and **getShortestPath** functions in Listing 25.2 WeightedGraph.cpp, testing whethere a vertex **u** is in **T** by invoking the STL **find** algorithm takes **O(n)** time. Modify these two functions by introducing an array named **isInT.** Set **isInT[u]** to **true** when a vertex **u** is added to **T**. Testing whethere a vertex **u** is in **T** can now be done in **O(1)** time. Write a test program using following code, where **graph1**, **printTree**, and **printAllPath** are the same as in Listings 25.6 and 25.9:

```
WeightedGraph<string> graph1(vectorOfVertices, edgeVector);
MST tree1 = graph1.getMinimumSpanningTree();
cout << "The spanning tree weight is " << tree1.getTotalWeight();
printTree<string>(tree1, graph1.getVertices());

ShortestPathTree tree = graph1.getShortestPath(5);
printAllPaths<string>(tree, graph1.getVertices());
```

25.12* (*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 Exercise 25.7. Your program should prompt the user to enter the name of the file, then two vertices, and display the shortest path between the two vertices. For example, for the graph in Figure 25.23, a shortest path between **0** and **1** may be displayed as **0 2 4 3 1**.

Here is a sample run of the program:

Sample output

```
Enter a file name: Exercise25_13.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
```