CHAPTER 25

WEIGHTED GRAPHS AND APPLICATIONS

Objectives

- To represent weighted edges using adjacency matrices and priority queues (§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

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.

25.2 Representing Weighted Graphs

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 25.1 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}
};
```

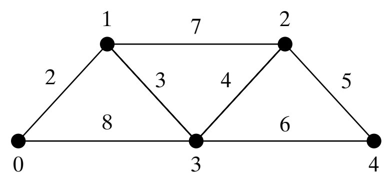


FIGURE 25.1 Each edge is assigned a weight on an edge-weighted graph.

integer weights

Note

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

25.2.2 Priority 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

```
#ifndef WEIGHTEDEDGE_H
#define WEIGHTEDEDGE_H

#include "Edge.h"

class WeightedEdge : public Edge
{
public:
  int weight; // The weight on edge (u, v)

  /** Create a weighted edge on (u, v) */
  WeightedEdge(int u, int v, int weight): Edge(u, v)
  {
    this->weight = weight;
  }

  /** Compare edges based on weights */
  bool operator<(WeightedEdge const &edge) const
  {
    return (*this).weight < edge.weight;
  }

  bool operator<=(const WeightedEdge &edge) const
  {
   return (*this).weight <= edge.weight;
  }

  bool operator>(const WeightedEdge &edge) const
  {
    return (*this).weight > edge.weight;
  }

  bool operator>=(const WeightedEdge &edge) const
  {
    return (*this).weight >= edge.weight;
  }

  bool operator==(const WeightedEdge &edge) const
  {
    return (*this).weight == edge.weight;
  }

  bool operator!=(const WeightedEdge &edge) const
  {
    return (*this).weight != edge.weight;
  }
};
#endif
```

edge weight

constructor

compare edges

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 it is desirable to store a vertex's adjacent edges in a priority queue so that you can remove the edges in increasing order of their weights. For this reason, we define the operator functions (<, <=, ==, !=, >, >=) in the **WeightedEdge** class.

For unweighted graphs, we use adjacency lists to represent edges. For weighted graphs, we still use adjacency lists, but they are priority queues. For example, the adjacency lists for the vertices in the graph in Figure 25.1 can be represented as follows:

```
vector<priority_queue<WeightedEdge, vector<WeightedEdge>,
  greater<WeightedEdge> > > queues;
for (int i = 0; i < numberOfVertices; i++)
{
  queues.push_back(priority_queue<WeightedEdge,
    vector<WeightedEdge>, greater<WeightedEdge> >());
}
```

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

queues[i] stores all edges adjacent to vertex **i**. Be default, the elements are compared using the < operator in a priority queue. The largest value is assigned the highest priority. The **greater<WeightedEdge>** in the constructor reverses the default order so that the smallest value is assigned the highest priority. So, the smallest weight edge will be removed first from the priority queue.

25.3 The WeightedGraph Class

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 25.2.

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**.

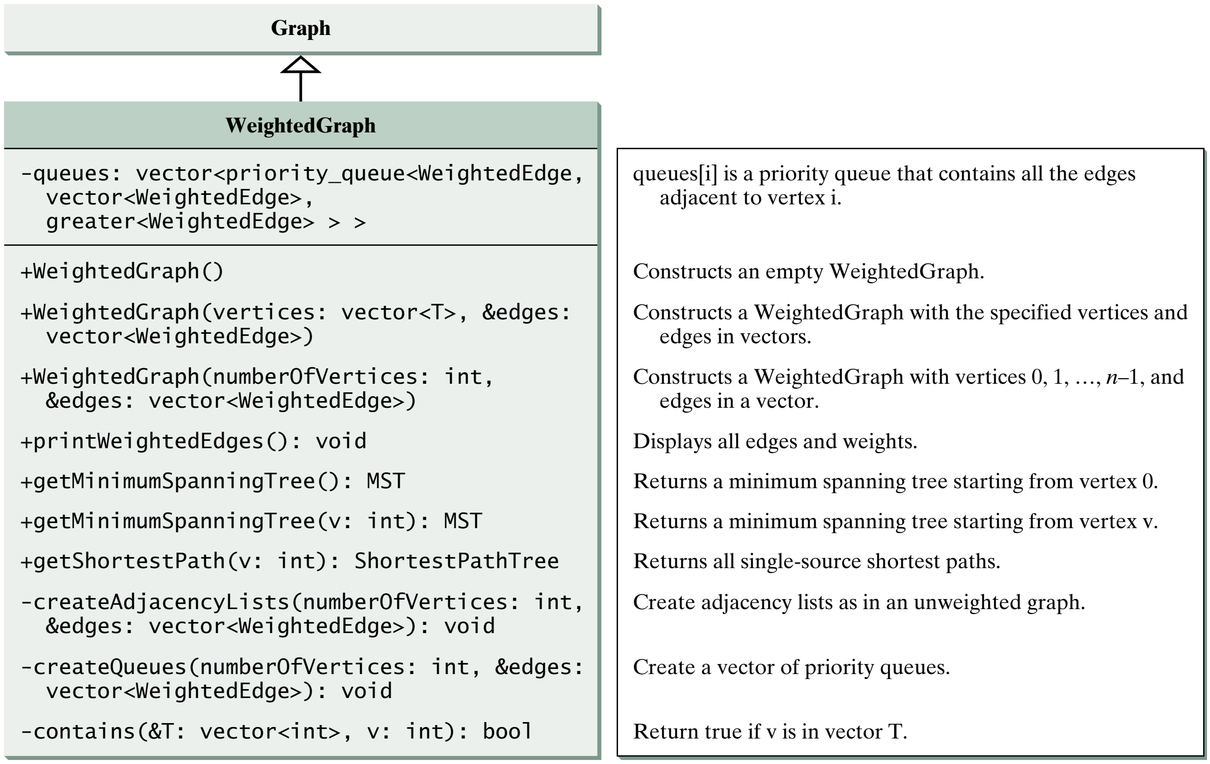


FIGURE 25.2 WeightedGraph extends Graph.

LISTING 25.2 WeightedGraph.h

```
#ifndef WEIGHTEDGRAPH_H
#define WEIGHTEDGRAPH_H

#include "Graph.h"
#include "WeightedEdge.h" // Defined in Listing 25.1
#include "MST.h" // Defined in Listing 25.5
#include "ShortestPathTree.h" // Defined in Listing 25.8
#include <queue> // For priority_queue

template<typename T>
class WeightedGraph : public Graph<T>
{
public:
  /** Construct an empty graph */
  WeightedGraph();

  /** Construct a graph from vertices and edges objects */
  WeightedGraph(vector<T> &vertices, vector<WeightedEdge> &edges);

  /** Construct a graph with vertices 0, 1, ..., n-1 and
    *  edges in a vector */
```

extends Graph

no-arg constructor

constructor

constructor
printWeightedEdges
getMinimumSpanningTree
getShortestPath
vector of priority queues
no-arg constructor
constructor
vertices
adjacency list
priority queues

```
  WeightedGraph(int numberOfVertices, vector<WeightedEdge> &edges);

  /** Print all edges in the weighted tree */
  void WeightedGraph<T>::printWeightedEdges();

  /** Get a minimum spanning tree rooted at vertex 0 */
  MST getMinimumSpanningTree();

  /** Get a minimum spanning tree rooted at a specified vertex */
  MST getMinimumSpanningTree(int startingVertex);

  /** Find single-source shortest paths */
  ShortestPathTree getShortestPath(int sourceVertex);

private:
  /** Priority adjacency lists on edge weights */
  vector<priority_queue<WeightedEdge, vector<WeightedEdge>,
    greater<WeightedEdge> > > queues;

  /** Create adjacency lists as in an unweighted graph */
  void createAdjacencyLists(int numberOfVertices,
    vector<WeightedEdge> &edges);

  /** Create a vector of priority queues */
  void createQueues(int numberOfVertices,
    vector<WeightedEdge> &edges);

  /** Return true if v is in vector T */
  bool contains(vector<int> &T, int v);
};

const int INFINITY = 2147483647;

template<typename T>
WeightedGraph<T>::WeightedGraph()
{
}

template<typename T>
WeightedGraph<T>::WeightedGraph(vector<T> &vertices,
  vector<WeightedEdge> &edges)
{
  // vertices is defined as protected in the Graph class
  this->vertices = vertices;

  // Create the adjacency list neighbors for the Graph class
  createAdjacencyLists(vertices.size(), edges);

  // Create the adjacency priority queues for weighted graph
  createQueues(vertices.size(), edges);
}

template<typename T>
WeightedGraph<T>::WeightedGraph(int numberOfVertices,
  vector<WeightedEdge> &edges)
{
  // vertices is defined as protected in the Graph class
  for (int i = 0; i < numberOfVertices; i++)
    vertices.push_back(i); // vertices is {0, 1, 2, ..., n-1}

```

```
  // Create the adjacency list neighbors for the Graph class
  createAdjacencyLists(numberOfVertices, edges);

  // Create the adjacency priority queues for weighted graph
  createQueues(numberOfVertices, edges);
}

template<typename T>
```

create adjacency list

```
void WeightedGraph<T>::createAdjacencyLists(int numberOfVertices,
  vector<WeightedEdge> &edges)
{
  // neighbors is defined as protected in the Graph class
  for (int i = 0; i < numberOfVertices; i++)
  {
    neighbors.push_back(vector<int>(0));
  }

  for (int i = 0; i < edges.size(); i++)
  {
    int u = edges[i].u;
    int v = edges[i].v;
```

neighbors

```
    neighbors[u].push_back(v);
  }
}

template<typename T>
```

create queues

```
void WeightedGraph<T>::createQueues(int numberOfVertices,
  vector<WeightedEdge> &edges)
{
  for (int i = 0; i < numberOfVertices; i++)
  {
    queues.push_back(priority_queue<WeightedEdge,
    vector<WeightedEdge>, greater<WeightedEdge> >());
  }

  for (int i = 0; i < edges.size(); i++)
  {
    int u = edges[i].u;
    int v = edges[i].v;
    int weight = edges[i].weight;
```

queues

```
    queues[u].push(WeightedEdge(u, v, weight));
  }
}

template<typename T>
```

print edges

```
void WeightedGraph<T>::printWeightedEdges()
{
  for (int i = 0; i < queues.size(); i++)
  {
    // Display all edges adjacent to vertex with index i
    cout << "Vertex " << i << ": ";

    // Get a copy of queues[i], so as to keep original queue intact
    priority_queue<WeightedEdge, vector<WeightedEdge>,
      greater<WeightedEdge> > pQueue = queues[i];
    while (!pQueue.empty())
    {
      WeightedEdge edge = pQueue.top();
      pQueue.pop();
      cout << "(" << edge.u << ", " << edge.v << ", "
```

minimum spanning tree
start from vertex 0
minimum spanning tree
vertices in tree
add to tree
number of vertices
parent vector
initialize parent
total weight
a copy of queues
more vertices?
every u in tree
remove visited vertex
queues[u] is empty
smallest edge to u
update **smallestWeight**

```
        << edge.weight << ") ";
    }
    cout << endl;
  }
}

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

template<typename T>
MST WeightedGraph<T>::getMinimumSpanningTree(int startingVertex)
{
  vector<int> T;
  // T initially contains the startingVertex;
  T.push_back(startingVertex);

  int numberOfVertices = vertices.size(); // Number of vertices
  vector<int> parent(numberOfVertices); // Parent of a vertex
  // Initially set the parent of all vertices to -1
  for (int i = 0; i < parent.size(); i++)
    parent[i] = -1;
  int totalWeight = 0; // Total weight of the tree thus far

  // Clone the queue, so as to keep the original queue intact
  vector<priority_queue<WeightedEdge, vector<WeightedEdge>,
    greater<WeightedEdge> > > queues = this->queues;

  // All vertices are found?
  while (T.size() < numberOfVertices)
  {
    // Search for the vertex with the smallest edge adjacent to
    // a vertex in T
    int v = -1;
    int smallestWeight = INFINITY;
    for (int i = 0; i < T.size(); i++)
    {
      int u = T[i];
      while (!queues[u].empty() && contains(T, queues[u].top().v))
      {
        // Remove the edge from queues[u] if the adjacent
        // vertex of u is already in T
        queues[u].pop();
      }

      if (queues[u].empty())
        continue; // Consider the next vertex in T

      // Current smallest weight on an edge adjacent to u
      WeightedEdge edge = queues[u].top();
      if (edge.weight < smallestWeight)
      {
        v = edge.v;
        smallestWeight = edge.weight;
        // If v is added to the tree, u will be its parent
        parent[v] = u;
      }
```

```
    } // End of for

    T.push_back(v); // Add a new vertex to the tree
    totalWeight += smallestWeight;
  } // End of while

  return MST(startingVertex, parent, T, totalWeight);
}

template<typename T>
bool WeightedGraph<T>::contains(vector<int> &T, int v)
{
  for (int i = 0; i < T.size(); i++)
  {
    if (T[i] == v) return true;
  }

  return false;
}

template<typename T>
ShortestPathTree WeightedGraph<T>::getShortestPath(int sourceVertex)
{
  // T stores the vertices whose path found so far
  vector<int> T;
  // T initially contains the sourceVertex;
  T.push_back(sourceVertex);

  // vertices is defined in the Graph class
  int numberOfVertices = vertices.size();

  // parent[v] stores the previous vertex of v in the path
  vector<int> parent(numberOfVertices);
  parent[sourceVertex] = -1; // The parent of source is set to -1

  // costs[v] stores the cost of the path from v to the source
  vector<int> costs(numberOfVertices);
  for (int i = 0; i < costs.size(); i++)
  {
    costs[i] = INFINITY; // Initial cost set to infinity
  }

  costs[sourceVertex] = 0; // Cost of source is 0

  // Clone the queue, so as to keep the original queue intact
  vector<priority_queue>WeightedEdge, vector<WeightedEdge>,
    greater<WeightedEdge> > > queues = this->queues;

  // Expand verticesFound
  while (T.size() < numberOfVertices)
  {
    int v = -1; // Vertex to be determined
    int smallestCost = INFINITY; // Set to infinity
    for (int i = 0; i < T.size(); i++)
    {
      int u = T[i];
      while (!queues[u].empty() && contains(T, queues[u].top().v))
        queues[u].pop(); // Remove the vertex in verticesFound

```

add to tree
update **totalWeight**
contains
getShortestPath
vertices found
add source
number of vertices
parent vector
parent of root
costs vector
source cost
a copy of queues
more vertices left?
determine one
remove visited vertex

queues[u] is empty

smallest edge to u

update **smallestCost**

v now found

add to T

create a path tree

```
260       if (queues[u].empty())
261         continue; // Consider the next vertex in T
262 
263       WeightedEdge e = queues[u].top();
264       if (costs[u] + e.weight < smallestCost)
265       {
266         v = e.v;
267         smallestCost = costs[u] + e.weight;
268         // If v is added to the tree, u will be its parent
269         parent[v] = u;
270       }
271     } // End of for
272 
273     T.push_back(v); // Add a new vertex to the set
274     costs[v] = smallestCost;
275   } // End of while
276 
277   // Create a ShortestPathTree
278   return ShortestPathTree(sourceVertex, parent, T, costs);
279 }
280 
281 #endif
```

WeightedGraph is derived from Graph (line 11). When you construct a WeightedGraph, the properties vertices and neighbors in the parent class Graph are initialized (lines 65–68, 79–83). Note that vertices and neighbors are defined as protected in Graph, so they can be accessed in the child class WeightedGraph.

A priority queue (lines 38–39) is used internally to store adjacent edges for a vertex. When a WeightedGraph is constructed, its priority adjacency queues are created (lines 71 and 86). The functions getMinimumSpanningTree (lines 148–208) and getShortestPaths (lines 221–279) will be introduced in the upcoming sections.

Listing 25.3 gives a test program that creates a graph for the one in Figure 24.1 and another graph for the one in Figure 25.1.

LISTING 25.3 TestWeightedGraph.cpp

vertices

edge array

```
#include <iostream>
#include <string>
#include <vector>
#include "WeightedGraph.h"
#include "WeightedEdge.h"
using namespace std;

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}
  };
  const int NUMBER_OF_EDGES = 46; // 46 edges in Figure 24.1

  // Create a vector for vertices
  vector<string> vectorOfVertices(12);
  copy(vertices, vertices + 12, vectorOfVertices.begin());

  // Create a vector for edges
  vector<WeightedEdge> edgeVector;
  for (int i = 0; i < NUMBER_OF_EDGES; i++)
    edgeVector.push_back(WeightedEdge(edges[i][0],
      edges[i][1], edges[i][2]));

  WeightedGraph<string> graph1(vectorOfVertices, edgeVector);
  cout << "The number of vertices in graph1: " << graph1.getSize();
  cout << "\nThe vertex with index 1 is " << graph1.getVertex(1);
  cout << "\nThe index for Miami is " << graph1.getIndex("Miami");

  cout << "\nThe edges for graph1: " << endl;
  graph1.printWeightedEdges();

  // Create a graph for Figure 25.1
  int edges2[][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}
  }; // 14 edges in Figure 25.1

  const int NUMBER_OF_EDGES2 = 14; // 14 edges in Figure 25.1

  vector<WeightedEdge> edgeVector2;
  for (int i = 0; i < NUMBER_OF_EDGES2; i++)
    edgeVector2.push_back(WeightedEdge(edges2[i][0],
      edges2[i][1], edges2[i][2]));

  WeightedGraph<int> graph2(5, edgeVector2); // 5 vertices in graph2

  cout << "The number of vertices in graph2: " << graph2.getSize();
  cout << "\nThe edges for graph2: " << endl;
  graph2.printWeightedEdges();

  return 0;
}
```

vector of vertices

vector of edges

create graph

getSize()

getVertex()

getIndex(vertex)

printWeightedEdges

edge array

vector of edges

create graph

printWeightedEdges

```
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 25.1 in lines 55–71. Line 75 invokes the **printWeightedEdges()** function on **graph2** to display all edges in **graph2**.

priority queue

Note

The adjacent edges for each vertex are displayed in increasing order of their weights, because the priority queues are used to store the edges. When you remove an edge from the queue, the one with the smallest weight is removed first.

25.4 Minimum Spanning Trees

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 25.3(b), 25.3(c), 25.3(d) are spanning trees for the graph in Figure 25.3(a). The trees in Figures 25.3(c) and 25.3(d) are minimum spanning trees.

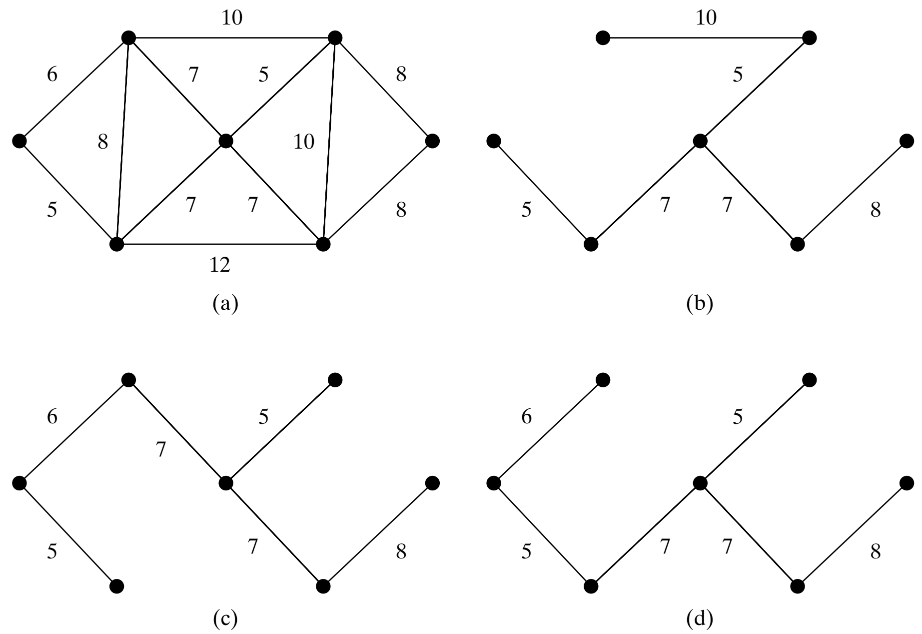


FIGURE 25.3 The tree in (c) and (d) are minimum spanning trees of the graph in (a).

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.

Prim's algorithm

LISTING 25.4 Prim's Minimum Spanning Tree Algorithm

```
minimumSpanningTree()
{
  Let V denote the set of vertices in the graph;
  Let T be a set for the vertices in the spanning tree;
  Initially, add the starting vertex to T;

  while (size of T < n)
  {
    find u in T and v in V - T with the smallest weight
      on the edge (u, v), as shown in Figure 25.4;
    add v to T;
  }
}
```

add initial vertex
more vertices?
find a vertex
add to tree

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 25.4, (**u**, **v**) is the one with the smallest weight.

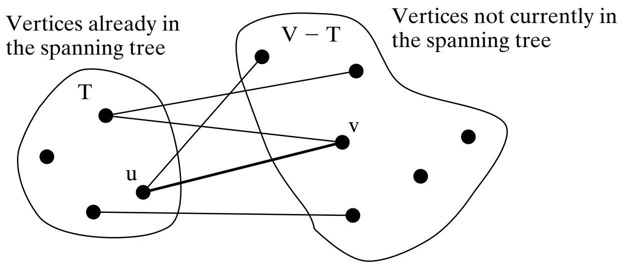


FIGURE 25.4 Find a vertex u in T that connects a vertex v in V − T with the smallest weight.

example

Consider the graph in Figure 25.3(a). The algorithm adds the vertices to `T` in this order:

1. Add vertex `0` to `T`.
2. Add vertex `5` to `T`, since `Edge(5, 0, 5)` has the smallest weight among all edges adjacent to the vertices in `T`, as shown in Figure 25.5(a).

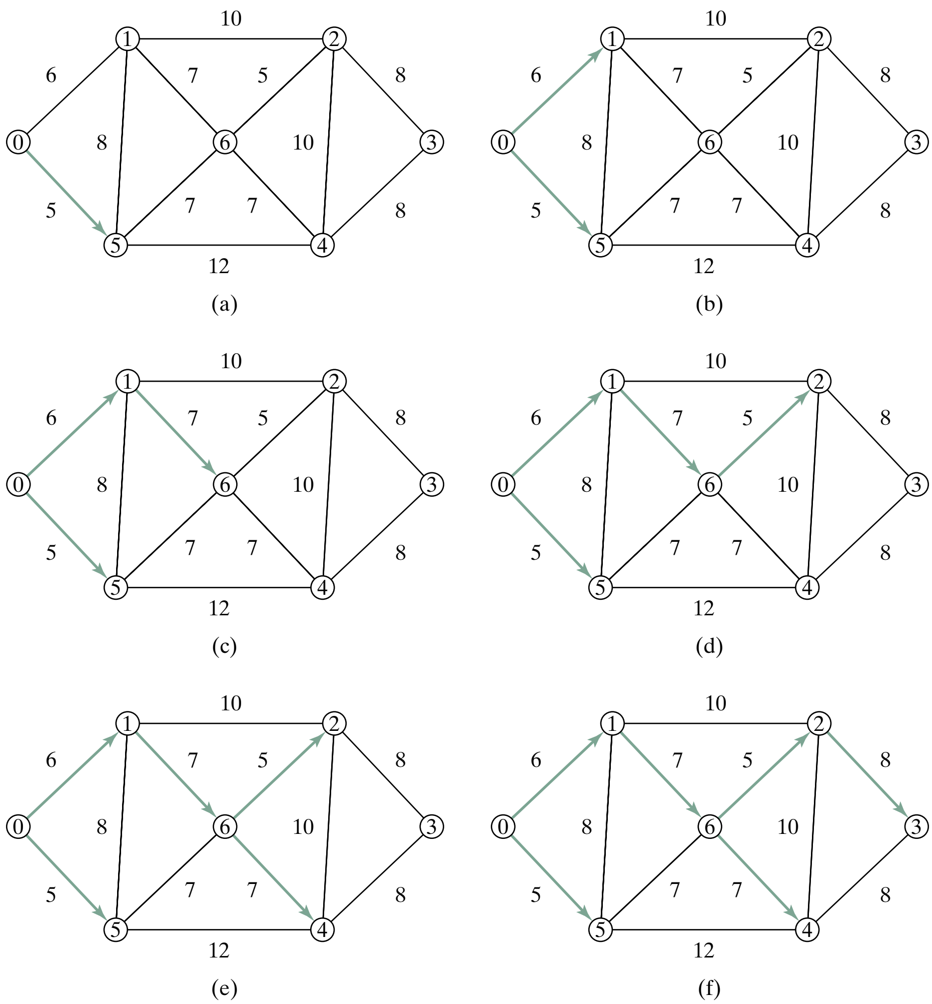


FIGURE 25.5 The adjacent vertices with the smallest weight are added successively to T.

3. Add vertex `1` to `T`, since `Edge(1, 0, 6)` has the smallest weight among all edges adjacent to the vertices in `T`, as shown in Figure 25.5(b).
4. Add vertex `6` to `T`, since `Edge(6, 1, 7)` has the smallest weight among all edges adjacent to the vertices in `T`, as shown in Figure 25.5(c).
5. Add vertex `2` to `T`, since `Edge(2, 6, 5)` has the smallest weight among all edges adjacent to the vertices in T, as shown in Figure 25.5(d).
6. Add vertex `4` to `T`, since `Edge(4, 6, 7)` has the smallest weight among all edges adjacent to the vertices in `T`, as shown in Figure 25.5(e).
7. Add vertex `3` to `T`, since `Edge(3, 2, 8`) has the smallest weight among all edges adjacent to the vertices in `T`, as shown in Figure 25.5(f).

Note

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

unique 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.

connected and undirected

25.4.2 Implementation of the MST Algorithm

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 25.6. The `Tree` class was defined in Listing 24.4. Listing 24.5 implements the `MST` class.

getMinimumSpanning-Tree()

Listing 25.5 MST.h

```
#ifndef MST_H
#define MST_H

#include "Tree.h" // Defined in Listing 24.4

class MST : public Tree
{
public:
  /** Create an empty MST */
  MST()
  {
  }

  /** Construct a tree with root, parent, searchOrders,
    * and total weight */
  MST(int root, vector<int> &parent, vector<int> &searchOrders,
    int totalWeight) : Tree(root, parent, searchOrders)
  {
    this->totalWeight = totalWeight;
  }

  /** Return the total weight of the tree */
```

extends **Tree**

no-arg constructor

constructor

getTotalWeight

```
  int getTotalWeight()
  {
    return totalWeight;
  }

private:
  int totalWeight;
};
#endif
```

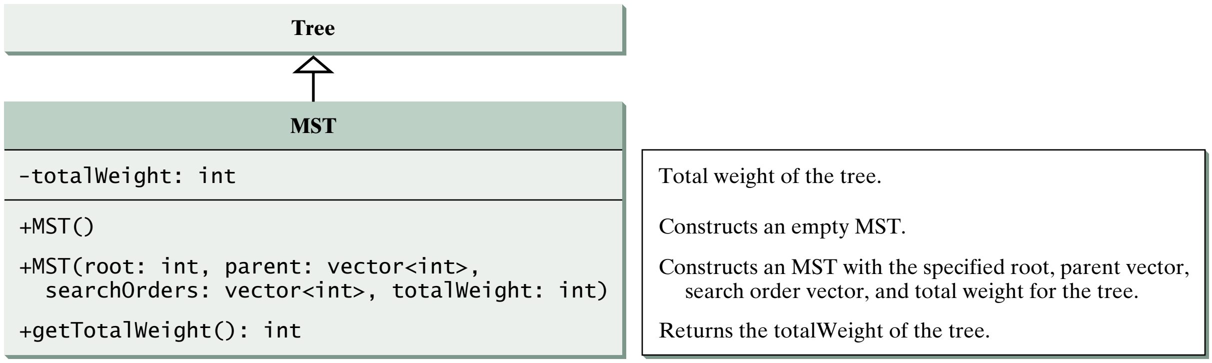


FIGURE 25.6 The MST class extends the Tree class.

The getMinimumSpanningTree function was implemented in lines 148–208 in Listing 25.2. The getMinimumSpanningTree(int startingVertex) function first adds startingVertex to T (line 159). T is a vector that stores the vertices currently in the spanning tree (line 157). Note that T can be implemented using list, set, or vector. vector is most appropriate in this case, because the elements are always added to the end of T and it also maintains the search order.

vertices is defined as a protected data field in the Graph class, which is a vector that stores all vertices in the graph. vertices.size() returns the number of the vertices in the graph (line 161).

A vertex is added to T if it is adjacent to one of the vertices in T with the smallest weight (line 203). Such a vertex is found using the following procedure:

1. For each vertex u in T, find its neighbor with the smallest weight to u. All the neighbors of u are stored in queues[u]. queues[u].top() (line 193) returns the adjacent edge with the smallest weight. If a neighbor is already in T, remove it (line 186). To keep the original queues intact, a copy is created in lines 169–170. After lines 181–190, queues[u].top() (line 193) returns the vertex with the smallest weight to u.

2. Compare all these neighbors and find the one with the smallest weight (lines 194–200).

After a new vertex is added to T (line 203), totalWeight is updated (line 204). Once all vertices are added to T, an instance of MST is created (line 207).

The MST class extends the Tree class. To create an instance of MST, pass root, parent, searchOrders, and totalWeight (lines 207). The data fields root, parent, and searchOrders are defined in the Tree class.

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

time complexity

Listing 25.6 gives a test program that displays minimum spanning trees for the graph in Figure 24.1 and the graph in Figure 25.1, respectively.

Listing 25.6 TestMinimumSpanningTree.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 tree */
template<typename T>
void printTree(Tree &tree, vector<T> &vertices)
{
  cout << "\nThe root is " << vertices[tree.getRoot()];
  cout << "\nThe edges are: ";
  for (int i = 0; i < vertices.size(); i++)
  {
    if (tree.getParent(i) != -1)
      cout << " (" << vertices[i] << ", "
        << vertices[tree.getParent(i)] << ")";
  }
}

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}
  };
  const int NUMBER_OF_EDGES = 46; // 46 edges in Figure 24.1

  // Create a vector for vertices
  vector<string> vectorOfVertices(12);
  copy(vertices, vertices + 12, vectorOfVertices.begin());

  // Create a vector for edges
  vector<WeightedEdge> edgeVector;
```

printTree
getRoot
getParent
vertices
edges
to vector
edge vector

create graph1
minimum spanning tree
getTotalWeight()
print tree
edges
edge vector
create graph2
minimum spanning tree
print tree

```
56    for (int i = 0; i < NUMBER_OF_EDGES; i++)
57      edgeVector.push_back(WeightedEdge(edges[i][0],
58      edges[i][1], edges[i][2]));
59
60    WeightedGraph<string> graph1(vectorOfVertices, edgeVector);
61    MST tree1 = graph1.getMinimumSpanningTree();
62    cout << "The spanning tree weight is " << tree1.getTotalWeight();
63    printTree<string>(tree1, graph1.getVertices());
64
65    // Create a graph for Figure 25.1
66    int edges2[][3] =
67    {
68      {0, 1, 2}, {0, 3, 8},
69      {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
70      {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
71      {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
72      {4, 2, 5}, {4, 3, 6}
73    }; // 14 edges in Figure 25.1
74
75    const int NUMBER_OF_EDGES2 = 14; // 14 edges in Figure 25.1
76
77    vector<WeightedEdge> edgeVector2;
78    for (int i = 0; i < NUMBER_OF_EDGES2; i++)
79      edgeVector2.push_back(WeightedEdge(edges2[i][0],
80      edges2[i][1], edges2[i][2]));
81
82    WeightedGraph<int> graph2(5, edgeVector2); // 5 vertices in graph2
83    MST tree2 = graph2.getMinimumSpanningTree();
84    cout << "\nThe spanning tree weight is " << tree2.getTotalWeight();
85    printTree<int>(tree2, graph2.getVertices());
86
87    return 0;
88  }
```

```
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`.

graphical illustration

The graphical illustration of the minimum spanning tree is shown in Figure 25.7. 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.

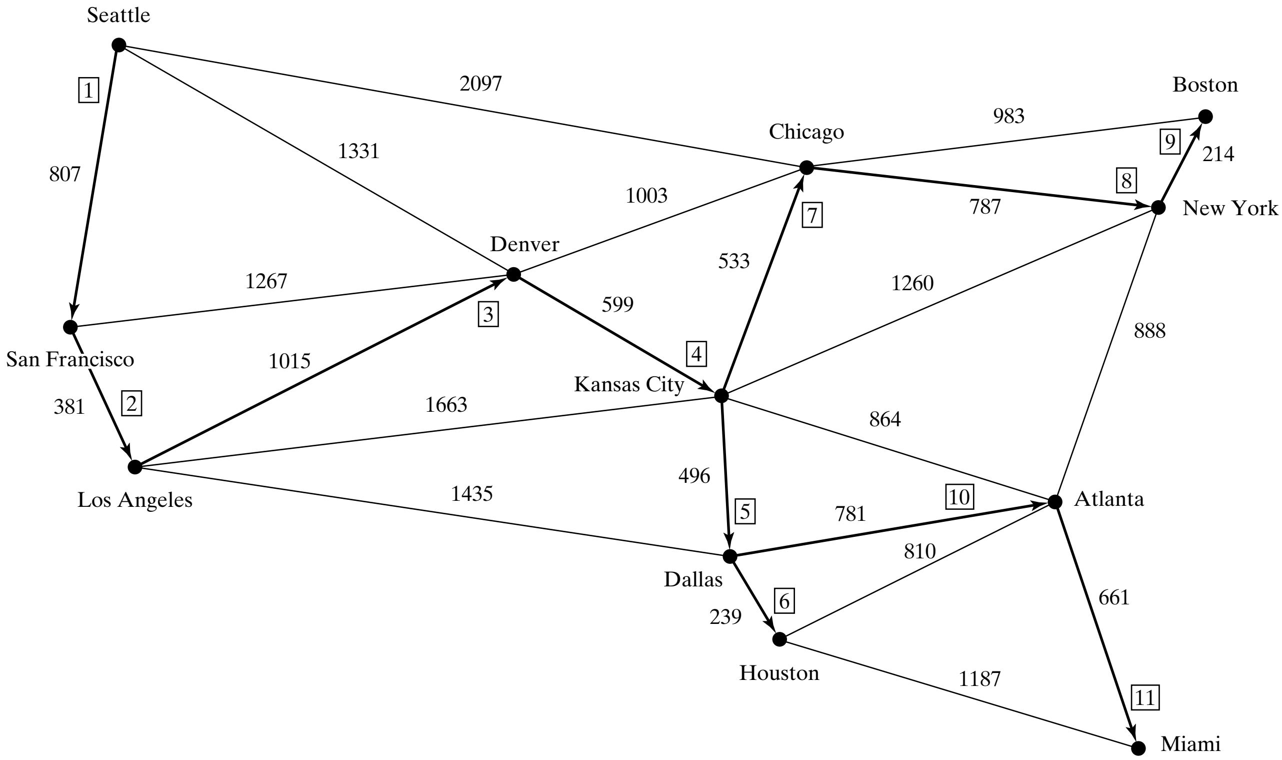


Figure 25.7 The edges in the minimum spanning tree for the cities are highlighted.

25.5 Finding Shortest Paths

§24.1 introduced the problem of finding the shortest distance between two cities for the graph in Figure 24.1. The answer to this problem is to find a shortest path between two vertices in the graph.

25.5.1 Shortest Path Algorithms

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. *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. The algorithm uses `costs[v]` to store the cost of the shortest path from vertex `v` to the source vertex `s`. So `costs[s]` is `0`. Initially assign infinity to `costs[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 have been found so far. 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 `costs[u] + w(u, v)` is the smallest, and moves `v` to `T`. Here `w(u, v)` denotes the weight on edge (`u`, `v`).

The algorithm is described in Listing 25.7.

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

```
shortestPath(s)
{
  Let V denote the set of vertices in the graph;
  Let T be a set that contains the vertices whose
```

add initial vertex

find next vertex

add a vertex

```
 5     paths to s have been found;
 6   Initially T contains source vertex s with costs[s] = 0;
 7
 8   while (size of T < n)
 9   {
10     find v in V - T with the smallest costs[u] + w(u, v) value
11       among all u in T;
12     add v to T and costs[v] = costs[u] + w(u, v);
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 the case of Dijkstra's, the vertex is adjacent to some vertex in the set with the minimum total cost to the source.

The algorithm starts by adding the source vertex s into T and sets costs[s] to 0 (line 6). It then continuously adds a vertex (say v) from V – T into T. v is the vertex that is adjacent to a vertex in T with the smallest costs[u] + w(u, v). For example, there are five edges connecting vertices in T and V – T, as shown in Figure 25.8; (u, v) is the one with the smallest costs[u] + w(u, v). After v is added to T, set costs[v] to costs[u] + w(u, v) (line 12).

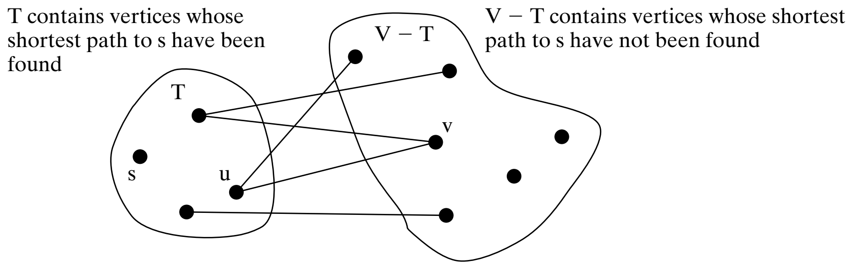


FIGURE 25.8 Find a vertex u in T that connects a vertex v in V – T with the smallest costs[u] + w(u, v).

Let us illustrate Dijkstra's algorithm using the graph in Figure 25.9(a). Suppose the source vertex is 1. So, costs[1] is 0 and the costs for all other vertices are initially ∞, as shown in Figure 25.9(b). 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.

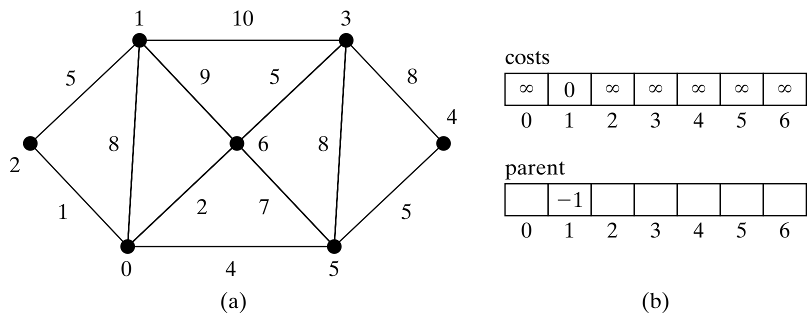


FIGURE 25.9 The algorithm will find all shortest paths from source vertex 1.

Initially set T contains the source vertex. Vertices 2, 0, 6, and 3 are adjacent to the vertices in T, and vertex 2 has the path of smallest cost to source vertex 1. So add 2 to T. `costs[2]` now becomes 5, as shown in Figure 25.10.

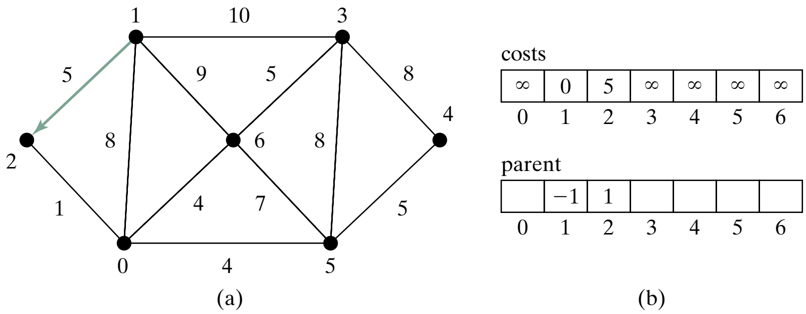


FIGURE 25.10 Now vertices 1 and 2 are in the set T.

Now T contains {1, 2}. Vertices 0, 6, and 3 are adjacent to the vertices in T, and vertex 0 has a path of smallest cost to source vertex 1. So add 1 to T. `costs[0]` now becomes 6, as shown in Figure 25.11.

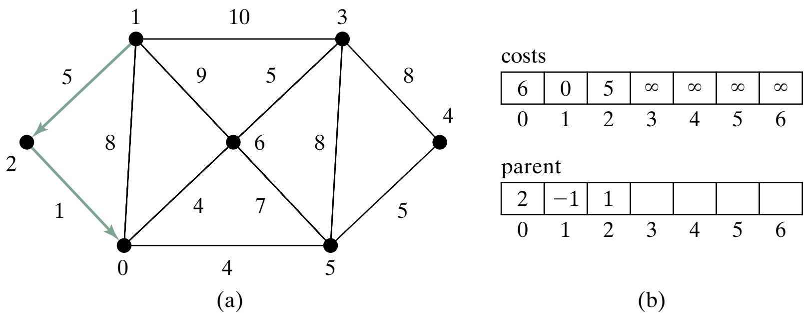


FIGURE 25.11 Now vertices {1, 2, 0} are in the set T.

Now T contains {1, 2, 0}. Vertices 3, 6, and 5 are adjacent to the vertices in T, and vertex 6 has the path of smallest cost to source vertex 1. So add 6 to T. `costs[6]` now becomes 9, as shown in Figure 25.12.

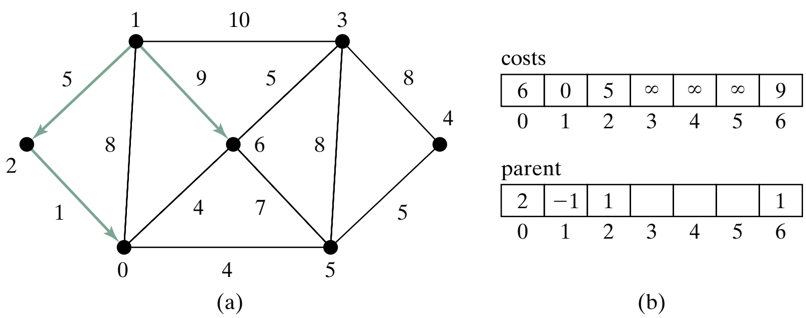
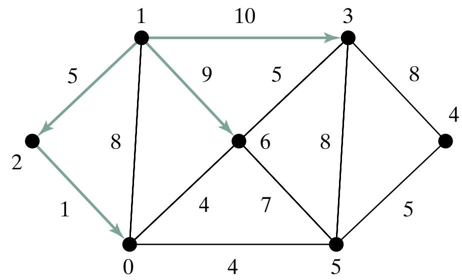


FIGURE 25.12 Now vertices {1, 2, 0, 6} are in the set T.

Now T contains {1, 2, 0, 6}. Vertices 3 and 5 are adjacent to the vertices in T, and both vertices have a path of the same smallest cost to source vertex 1. You can choose either 3 or 5. Let us add 3 to T. `costs[3]` now becomes 10, as shown in Figure 25.13.



(a)

costs

6	0	5	10	∞	∞	9
0	1	2	3	4	5	6

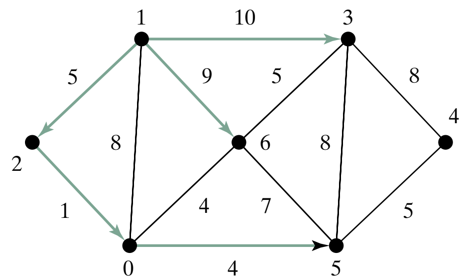
parent

2	−1	1	1			1
0	1	2	3	4	5	6

(b)

FIGURE 25.13 Now vertices {1, 2, 0, 6, 3} are in the set T.

Now T contains {1, 2, 0, 6, 3}. Vertices 4 and 5 are adjacent to the vertices in T, and vertex 5 has the path of smallest cost to source vertex 1. So add 5 to T. `costs[5]` now becomes 10, as shown in Figure 25.14.



(a)

costs

6	0	5	10	∞	10	9
0	1	2	3	4	5	6

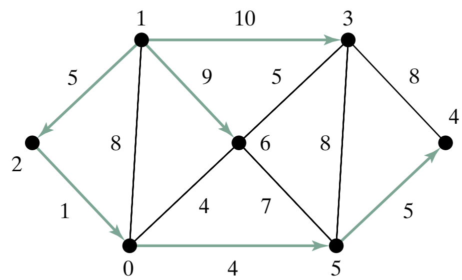
parent

2	−1	1	1		0	1
0	1	2	3	4	5	6

(b)

FIGURE 25.14 Now vertices {1, 2, 0, 6, 3, 5} are in the set T.

Now T contains {1, 2, 0, 6, 3, 5}. The smallest cost for a path to connect 4 with 1 is 15, as shown in Figure 25.15.



(a)

costs

6	0	5	10	15	10	9
0	1	2	3	4	5	6

parent

2	−1	1	1	5	0	1
0	1	2	3	4	5	6

(b)

FIGURE 25.15 Now vertices {1, 2, 6, 0, 3, 5, 4} are in the set T.

25.5.2 Implementation of the shortest-paths algorithm

shortest-path tree

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 25.16. Listing 25.8 implements the `ShortestPathTree` class

LISTING 25.8 ShortestPathTree.h

```
#ifndef SHORTESTPATHTREE_H
#define SHORTESTPATHTREE_H

#include "Tree.h"

class ShortestPathTree : public Tree
{
public:
  /** Create an empty ShortestPathTree */
  ShortestPathTree()
  {
  }

  /** Construct a tree with root, parent, searchOrders,
    * and cost */
  ShortestPathTree(int root, vector<int> &parent, vector<int>
    &searchOrders, vector<int> &costs)
    : Tree(root, parent, searchOrders)
  {
    this->costs = costs;
  }

  /** Return the cost for the path from the source to vertex v. */
  int getCost(int v)
  {
    return costs[v];
  }

private:
  vector<int> costs;
};
#endif
```

extends **Tree**

no-arg constructor

constructor

getCost

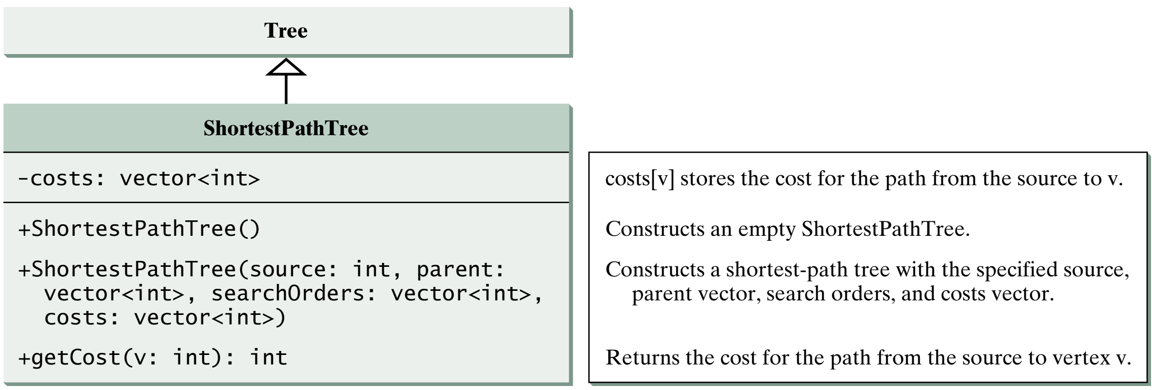


FIGURE 25.16 `ShortestPathTree` extends `Tree`.

The `getShortestPath(int sourceVertex)` function was implemented in lines 221–279 in Listing 25.2, WeightedGraph.h. The function first adds `sourceVertex` to `T` (line 227). `T` is a set that stores the vertices whose path has been found. `vertices` is defined as a protected data field in the `Graph` class, which is a vector that stores all vertices in the graph. `vertices.size()` returns the number of the vertices in the graph (line 230).

Each vertex is assigned a cost. The cost of the source vertex is `0` (line 243). The cost of all other vertices is initially assigned infinity (line 240).

The function needs to remove the elements from the queues in order to find the one with the smallest total cost. To keep the original queues intact, queues are cloned in lines 246–247.

A vertex is added to `T` if it is adjacent to one of the vertices in `T` with the smallest cost (line 273). Such a vertex is found using the following procedure:

1. For each vertex `u` in `T` (line 256), find its incident edge `e` with the smallest weight to `u`. All the incident edges to `u` are stored in `queues[u]`. `queues[u].top()` (line 257) returns the incident edge with the smallest weight. If `e.v` is already in `T`, remove `e` from `queues[u]` (line 258). After lines 257–261, `queues[u].top()` returns the edge `e` such that `e` has the smallest weight to `u` and `e.v` is not in `T` (line 263).

2. Compare all these edges and find the one with the smallest value on `costs[u] + e.getWeight()` (line 267).

After a new vertex is added to `T` (line 273), the cost of this vertex is updated (line 274). Once all vertices are added to `T`, an instance of `ShortestPathTree` is created (line 278).

ShortestPathTree class

The `ShortestPathTree` class extends the `Tree` class. To create an instance of `ShortestPathTree`, pass `sourceVertex`, `parent`, `searchOrders`, and `costs` (lines 278). `sourceVertex` becomes the root in the tree. The data fields `root`, `parent`, and `searchOrders` are defined in the `Tree` class.

Dijkstra's algorithm time complexity

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

Listing 25.9 gives a test program that displays all shortest paths from `Chicago` to all other cities in Figure 24.1 and all shortest paths from vertex `3` to all vertices for the graph in Figure 25.1, respectively.

LISTING 25.9 TestShortestPath.cpp

printAllPath · getRoot · getPath · getCost

```
#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 (int 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 i = path.size() - 1; i >= 0; i--)
    {
      cout << vertices[path[i]] << " ";
    }

    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}
  };
  const int NUMBER_OF_EDGES = 46; // 46 edges in Figure 24.1

  // Create a vector for vertices
  vector<string> vectorOfVertices(12);
  copy(vertices, vertices + 12, vectorOfVertices.begin());

  // Create a vector for edges
  vector<WeightedEdge> edgeVector;
  for (int i = 0; i < NUMBER_OF_EDGES; i++)
    edgeVector.push_back(WeightedEdge(edges[i][0],
    edges[i][1], edges[i][2]));

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

  // Create a graph for Figure 25.1
  int edges2[][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}
  }; // 14 edges in Figure 25.1

  const int NUMBER_OF_EDGES2 = 14; // 14 edges in Figure 25.1

  vector<WeightedEdge> edgeVector2;
  for (int i = 0; i < NUMBER_OF_EDGES2; i++)
    edgeVector2.push_back(WeightedEdge(edges2[i][0],
    edges2[i][1], edges2[i][2]));

```

vertices
edges
to vector
edge vector
create graph1
shortest path
print paths
edges
edge vector

create graph2
shortest path
print paths

```
   WeightedGraph<int> graph2(5, edgeVector2); // 5 vertices in graph2
   ShortestPathTree tree2 = graph2.getShortestPath(3);
   printAllPaths<int>(tree2, graph2.getVertices());

   return 0;
}
```

```
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)
```

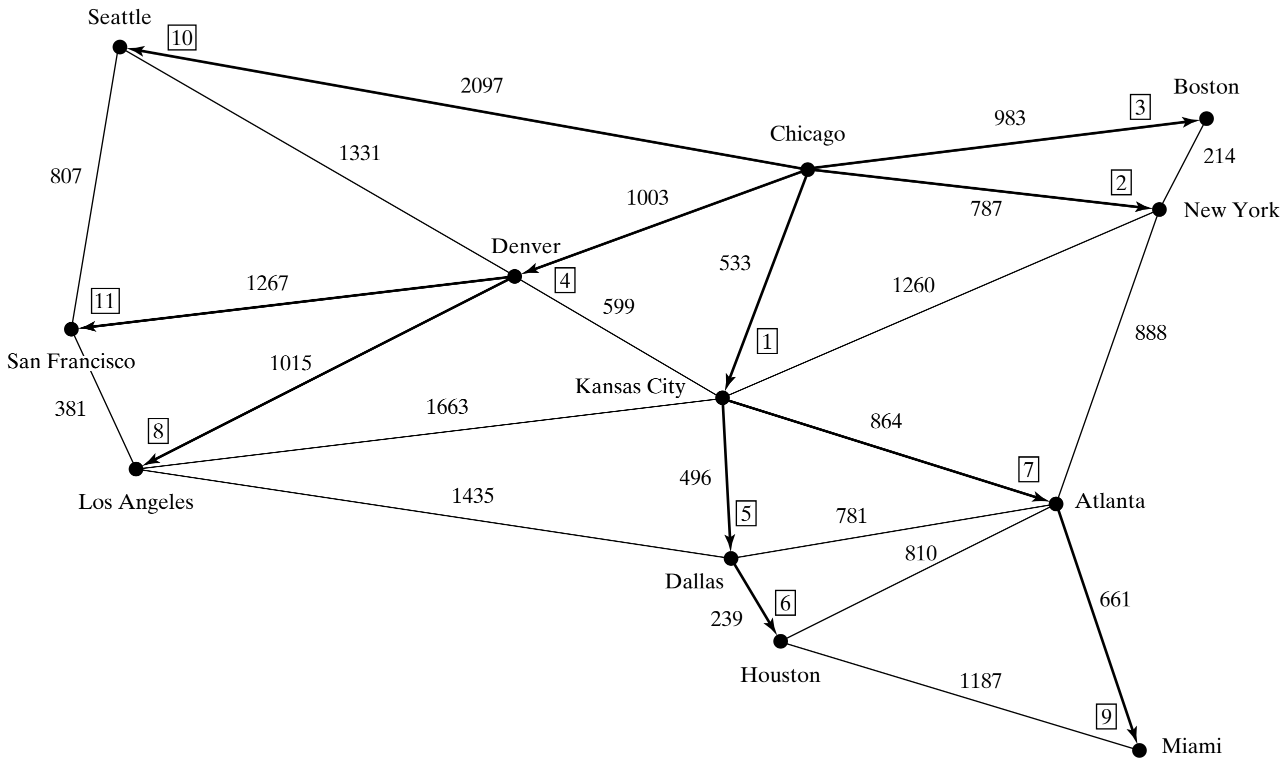


FIGURE 25.17 The shortest paths from Chicago to all other cities are highlighted.

The program creates a weighted graph for Figure 24.1 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 25.17. 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`.

25.6 Case Study: The Weighted Nine Tail Problem

§24.8 presented the nine tail problem and solved it using the BFS algorithm. This section presents a modification 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 25.18(a) to Figure 25.18(b) by flipping the three coins. So the weight for this move is `3`.

H	H	H
T	T	T
H	H	H

(a)

T	T	H
H	T	T
H	H	H

(b)

FIGURE 25.18 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 §24.8 for modeling the nine tail problem. We create a new class named `WeightedNineTailModel` that extends `NineTailModel`, as shown in Figure 25.19.

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

```
#ifndef WEIGHTEDNINETAILMODEL_H
#define WEIGHTEDNINETAILMODEL_H

#include "NineTailModel.h" // Defined in Listing 24.9
#include "WeightedGraph.h" // Defined in Listing 25.2

using namespace std;

class WeightedNineTailModel : public NineTailModel
{
```

extends **NineTailModel**

constructor
getNumberOfFlips
ShortestPathTree
get weighted edges
flips between two nodes
implement constructor
weighted edge vector
create a graph
shortest path tree
tree in **NineTailModel**
implement **getEdges()**
edge vector
number of flips
add an edge

```
public:
  /** Construct a model for the Nine Tail problem */
  WeightedNineTailModel();

  /** Get the number of flips from the target to u */
  int getNumberOfFlips(int u);

private:
  /** ShortestPathTree rooted at the target node 511 */
  ShortestPathTree spTree;

  /** Return a vector of Edge objects for the graph */
  /** Create edges among nodes */
  vector<WeightedEdge> getEdges();

  /** Get the number of flips from u to v */
  int getNumberOfFlips(int u, int v);
};

WeightedNineTailModel::WeightedNineTailModel()
{
  // Create edges
  vector<WeightedEdge> edges = getEdges();

  // Build a graph
  WeightedGraph<int> graph(NUMBER_OF_NODES, edges);

  // Build a shortest path tree rooted at the target node
  spTree = graph.getShortestPath(511);

  // The functions in NineTailModel use the tree property
  tree = spTree;
}

vector<WeightedEdge> WeightedNineTailModel::getEdges()
{
  vector<WeightedEdge> edges;

  for (int u = 0; u < NUMBER_OF_NODES; u++)
  {
    vector<char> node = getNode(u);
    for (int k = 0; k < 9; k++)
    {
      if (node[k] == 'H')
      {
        int v = getFlippedNode(node, k);
        int numberOfFlips = getNumberOfFlips(u, v);
        // Add edge (v, u) for a legal move from node u to node v
        // with weight numberOfFlips
        edges.push_back(WeightedEdge(v, u, numberOfFlips));
      }
    }
  }

  return edges;
}

int WeightedNineTailModel::getNumberOfFlips(int u, int v)
{
  vector<char> node1 = getNode(u);
```

```
71   vector<char> node2 = getNode(v);
72
73   int count = 0; // Count the number of different cells
74   for (int i = 0; i < node1.size(); i++)
75     if (node1[i] != node2[i]) count++;
76
77   return count;
78 }
79
80 int WeightedNineTailModel::getNumberOfFlips(int u)
81 {
82   return spTree.getCost(u);
83 }
84 #endif
```

a different cell?

getCost

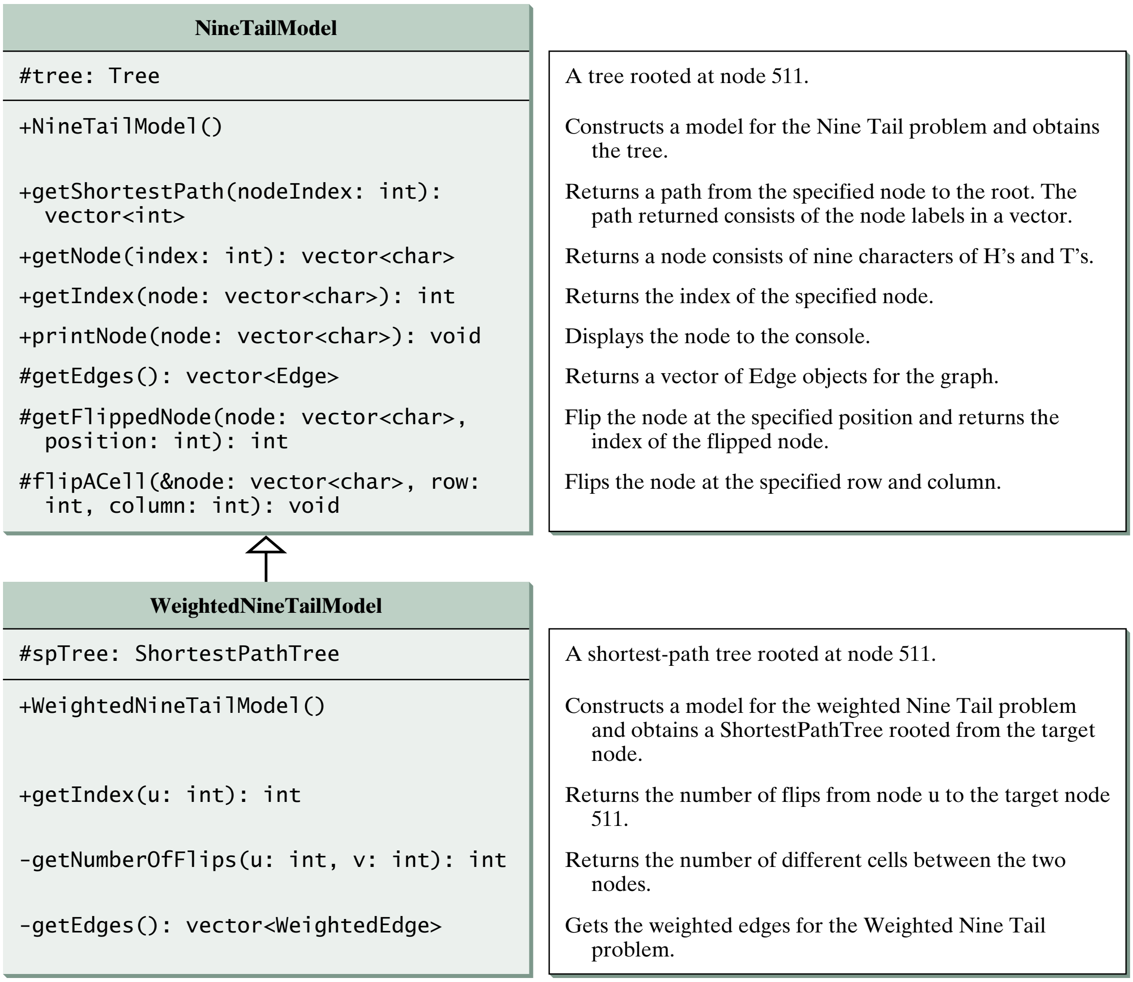


FIGURE 25.19 `WeightedNineTailModel` extends `NineTailModel`.

`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 57). The `getNumberOfFlips(int u, int v)` function returns the number of flips from node `u` to node v (lines 68–78). The number of flips is the number of the different cells between the two nodes (line 75).

The WeightedNineTailModel constructs a WeightedGraph (line 36) and obtains a ShortestPathTree spTree rooted at the target node 511 (line 39). It then assigns spTree to tree (line 42). 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 80–83) 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 82).

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

initial node

create model

get shortest path

print node

```
#include <iostream>
#include <vector>
#include "WeightedNineTailModel.h"
using namespace std;

int main()
{
  // Prompt the user to enter nine coins H's and T's
  cout << "Enter an initial nine coin H's and T's: " << endl;
  vector<char> initialNode(9);

  for (int i = 0; i < 9; i++)
    cin >> initialNode[i];

  cout << "The steps to flip the coins are " << endl;
  WeightedNineTailModel model;
  vector<int> path =
    model.getShortestPath(model.getIndex(initialNode));

  for (int i = 0; i < path.size(); i++)
    model.printNode(model.getNode(path[i]));

  cout << "The number of flips is " <<
    model.getNumberOfFlips(model.getIndex(initialNode)) << endl;

  return 0;
}
```

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

Key Terms

Dijkstra's algorithm 25–19
edge-weighted graph 25–2
minimum spanning tree 25–4
Prim's algorithm 25–13
shortest path 25–4
single-source shortest path 25–19
vertex-weighted graph 25–2

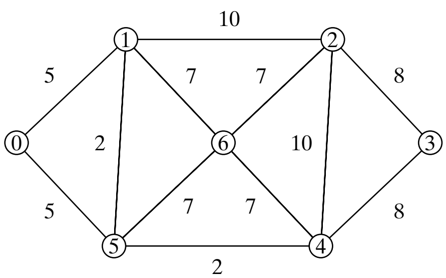
Chapter Summary

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.

Review Questions

Section 25.4

25.1 Find a minimum spanning tree for the following graph.



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

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

Section 25.5

25.4 Trace Dijkstra's algorithm for finding shortest paths from Boston to all other cities in Figure 24.1.

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

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

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* (*Implementing 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* (*Implementing 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* (*Modifying 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.

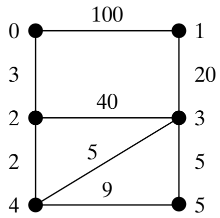
25.5* (*Prove or disprove*) The conjecture is that both `NineTailModel` and `WeightedNineTailModel` result in the same shortest path. Write a program to prove or disprove it.

(*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* (*Finding 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 25.20 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 25.20 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:

```
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* (*Creating 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* (*Finding 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 25.20, a shortest path between `0` and `1` may be displayed as `0 2 4 3 1`.

Here is a sample run of the program:

```
Enter a file name: Exercise25_9.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
```