

CHAPTER

13

Graph Applications

Objectives

- To know the Seven Bridges of Königsberg problem and applications of graphs (§13.1).
- To describe the graph terminologies: vertices, edges, simple graphs, weighted/unweighted graphs, and directed/undirected graphs (§13.2).
- To represent vertices and edges using lists, adjacent matrices, and adjacent lists (§13.3).
- To model graphs using the Graph interface, the AbstractGraph class, and the UnweightedGraph class (§13.4).
- To represent the traversal of a graph using the AbstractGraph.Tree class (§13.5).
- To design and implement depth-first search (§13.6).
- To design and implement breadth-first search (§13.7).
- To solve the nine tail problem using breadth-first search (§13.8).

13.1 Introduction

Graphs play an important role in modeling real-world problems. For example, the problem to find the shortest distance between two cities can be modeled using a graph, where the vertices represent cities and edges represent the roads and distances between two adjacent cities, as shown in Figure 13.1. The problem of finding the shortest distance between two cities is reduced to finding a shortest path between two vertices in a graph.

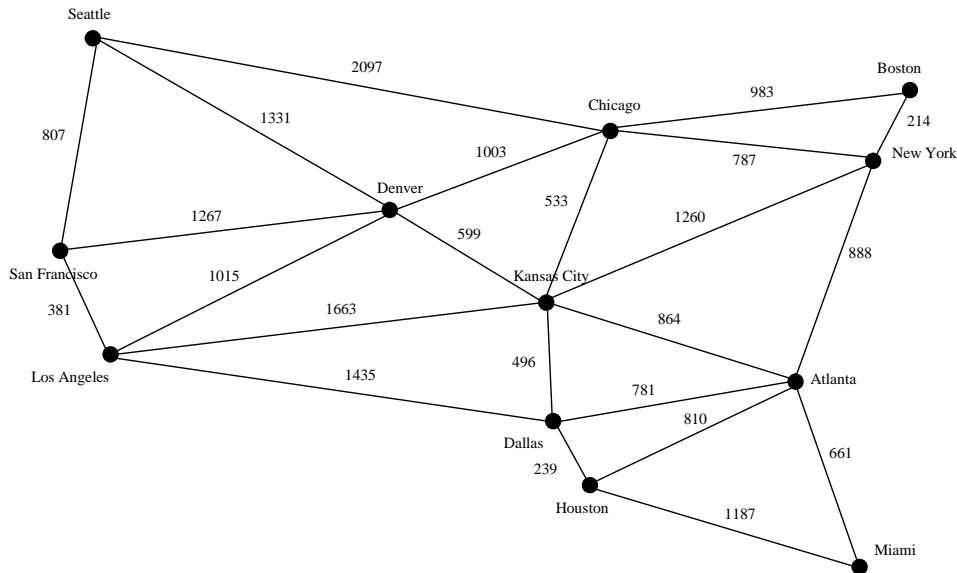


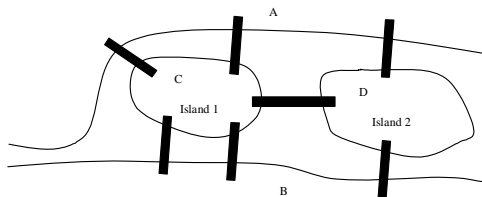
Figure 13.1

A graph can be used to model the distance between the cities.

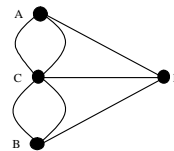
<Side Remark: Seven Bridges of Königsberg>

<Side Remark: graph theory>

The study of graph problems is known as *graph theory*. Graph theory was founded by Leonard Euler in 1736, when he introduced graph terminology to solve the famous *Seven Bridges of Königsberg*. The city of Königsberg, Prussia (now Kaliningrad, Russia) was divided by the Pregel River. There were two islands on the river. The city and islands were connected by seven bridges, as shown in Figure 13.2(a). The question is whether it is possible to walk with a route that crosses each bridge exactly once and return to the starting point. Euler proved that it was not possible.



(a) Seven bridge sketch



(b) Graph model

Figure 13.2

Seven bridges connected islands and land.

To prove it, Euler first abstracted the Königsberg city map into the sketch, as shown in Figure 13.2(a), by eliminating all streets. Second, he replaced each land mass with a dot, called a vertex or a node, and each bridge with a line, called an edge, as shown in Figure 13.2(b). This structure with vertices and edges is called a graph.

Looking at the graph, the question is whether there is a path starting from any vertex, traversing all edges exactly once, and returning to the starting vertex. Euler proved that for such path to exist each vertex must have even number of edges. Therefore, the *Seven Bridges of Königsberg* has no solution.

Graph problems are often solved using algorithms. Graph algorithms have many applications in various areas such as computer science, mathematics, biology, engineering, economics, genetics, and social sciences. This chapter presents the algorithms for depth-first search and breadth-first search, and their applications. The next chapter presents the algorithms for finding shortest paths, finding a minimum spanning tree for weighted graphs, and their applications.

13.2 Basic Graph Terminologies

This chapter does not assume that the reader has any prior knowledge on graph theory nor discrete mathematics. We use plain and simple terms to define graphs.

<Side Remark: what is a graph?>

What is a graph? A graph is a mathematical structure that represents relationships among entities in the real world. For example, the graph in Figure 13.1 represents the roads and their distances among cities, and the graph in Figure 13.2(b) represents the bridges among land masses.

<Side Remark: define a graph>

A graph consists of a nonempty set of vertices, nodes, or points, and a set of edges that connect the vertices. For convenience, we define a graph as $G = (V, E)$, where V represents a set of vertices and E represents a set of edges. For example, V and E for the graph in Figure 13.1 are as follows:

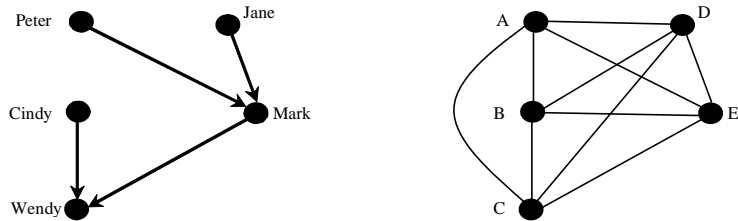
```
V = {"Seattle", "San Francisco", "Los Angeles",  
     "Denver", "Kansas City", "Chicago", "Boston", "New York",  
     "Atlanta", "Miami", "Dallas", "Houston"};  
  
E = {{ "Seattle", "San Francisco"}, {"Seattle", "Chicago"},  
     {"Seattle", "Denver"}, {"San Francisco", "Denver"},  
     ...  
     };
```

<Side Remark: directed vs. undirected>

A graph may be directed or undirected. In a directed graph, each edge has a direction, which indicates that you can move from one vertex to the other through the edge. You may model parent-child relationships

using a directed graph, where an edge from vertex A to B indicates that A is a parent of B.

Figure 13.3(a) shows a directed graph. In an undirected graph, you can move in both directions between vertices. The graph in Figure 13.1 is undirected.



(a) A directed graph

(b) A complete graph

Figure 13.3

Graphs may appear in many forms.

<Side Remark: weighted vs. unweighted>

Edges may be weighted or unweighted. For example, each edge in the graph in Figure 13.1 has a weight that represents the distance between two cities.

<Side Remark: adjacent>

<Side Remark: incident>

<Side Remark: degree>

Two vertices in a graph are said to be *adjacent* if they are connected by the same edge. Similarly two edges are said to be *adjacent* if they connect to the same vertex. An edge in a graph that joins two vertices is said to be *incident* to both vertices. The degree of a vertex is the number of edges incident to it.

<Side Remark: neighbor>

Two vertices are called *neighbors* if they are adjacent. Similarly two edges are called *neighbors* if they are incident.

<Side Remark: loop>

<Side Remark: parallel edge>

<Side Remark: simple graph>

<Side Remark: complete graph>

A *loop* is an edge that links a vertex to itself. If two vertices are connected by two or more edges, these edges are called *parallel edges*. A simple graph is one that has no loops and parallel edges. A complete graph is the one in which every two pairs of vertices are connected, as shown in Figure 13.3(b).

<Side Remark: spanning tree>

Assume that the graph is connected and undirected. A *spanning tree* of a graph is a subgraph that is a tree and connects all vertices in the graph.

13.3 Representing Graphs

To write a program that processes and manipulates graphs, you have to store or represent graphs in computer.

13.3.1 Representing Vertices

The vertices can be stored in an array. For example, you can store all the city names in the graph in Figure 13.1 using the following array:

```
String[] vertices = {"Seattle", "San Francisco", "Los Angeles",  
"Denver", "Kansas City", "Chicago", "Boston", "New York",  
"Atlanta", "Miami", "Dallas", "Houston"};
```

NOTE:

<Side Remark: vertex type>

The vertices can be in fact any type of objects. For example, you may consider cities as objects that contain the information such as name, population, mayor, etc. So, you may define vertices as follows:

```
City city0 = new City("Seattle", 563374, "Greg Nickels");  
...  
City city11 = new City("Houston", 1000203, "Richard Daly");  
Object[] vertices = {city0, city1, ..., city11};  
  
public class City {  
    private String cityName;  
    private int population;  
    private String mayor;  
  
    public City(String cityName, int population, String mayor) {  
        this.cityName = cityName;  
        this.population = population;  
        this.mayor = mayor;  
    }  
  
    public String getCityName() {  
        return cityName;  
    }  
  
    public int getPopulation() {  
        return population;  
    }  
  
    public String getCityName() {  
        return cityName;  
    }  
  
    public void setMayor(String mayor) {  
        this.mayor = mayor;  
    }  
  
    public void setPopulation(int population) {  
        this.population = population;  
    }  
}
```

*****END NOTE**

The vertices can be conveniently labeled using natural numbers 0, 1, 2, ..., n-1, for a graphs for n vertices. So, vertices[0] represents "Seattle", vertices[1] represents "San Francisco", and so on, as shown in Figure 13.4.

vertices[0]	Seattle
vertices[1]	San Francisco
vertices[2]	Los Angeles
vertices[3]	Denver
vertices[4]	Kansas City
vertices[5]	Chicago
vertices[6]	Boston
vertices[7]	New York
vertices[8]	Atlanta
vertices[9]	Miami
vertices[10]	Dallas
vertices[11]	Houston

Figure 13.4

An array stores the vertex names.

NOTE:

<Side Remark: reference vertex>

You can reference a vertex by its name or its index, whichever is convenient. Obviously, it is easy to access a vertex via its index in a program.

*****END NOTE**

13.3.2 Representing Edges: Edge Array

The edges can be represented using a two-dimensional array. For example, you can store all the edges in the graph in Figure 13.1 using the following array:

```
int[][] edges = {  
    {0, 1}, {0, 3}, {0, 5},  
    {1, 0}, {1, 2}, {1, 3},  
    {2, 1}, {2, 3}, {2, 4}, {2, 10},  
    {3, 0}, {3, 2}, {3, 4}, {3, 5},  
    {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},  
    {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},  
    {6, 5}, {6, 7},
```

```

    {7, 4}, {7, 5}, {7, 6}, {7, 8},
    {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
    {9, 8}, {9, 11},
    {10, 2}, {10, 4}, {10, 8}, {10, 11},
    {11, 8}, {11, 9}, {11, 10}
  };
}

```

<Side Remark: array edge>

This is known as the *edge representation using arrays*.

13.3.3 Representing Edges: Edge Objects

Another way to represent the edges is to define edges as objects and stores the edges in a `java.util.ArrayList`. The `Edge` class can be defined as follows:

```

public class Edge {
    int u;
    int v;

    public Edge(int u, int v) {
        this.u = u;
        this.v = v;
    }
}

```

For example, you can store all the edges in the graph in Figure 13.1 using the following list:

```

java.util.ArrayList<Edge> list = new java.util.ArrayList<Edge>();
list.add(new Edge(0, 1));
list.add(new Edge(0, 3));
list.add(new Edge(0, 5));
...

```

Storing `Edge` objects in an `ArrayList` is useful if you don't know the edges in advance.

Representing edges using array or array list is intuitive for input, but not efficient for internal processing. The following two sections introduce how to represent graphs using adjacency matrices and adjacency lists. These two data structures are efficient for processing graphs.

13.3.4 Representing Edges: Adjacency Matrices

Assume that the graph has n vertices. You can use a two-dimensional $n \times n$ matrix, say `adjacencyMatrix`, to represent edges. Each element in the array is 0 or 1. `adjacencyMatrix[i][j]` is 1 if there is an edge from vertex i to vertex j ; otherwise, `adjacencyMatrix[i][j]` is 0. If the graph is undirected, the matrix is symmetric, because `adjacencyMatrix[i][j]` is the same as `adjacencyMatrix[j][i]`. For example, the edges in the graph in Figure 13.1 can be represented using an adjacency matrix as follows:

```

int[][] adjacencyMatrix = {
    {0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0}, // Seattle

```

```

    {1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0}, // San Francisco
    {0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0}, // Los Angeles
    {1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0}, // Denver
    {0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1}, // Kansas City
    {1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0}, // Chicago
    {0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0}, // Boston
    {0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0}, // New York
    {0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1}, // Atlanta
    {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1}, // Miami
    {0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0}, // Dallas
    {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1}, // Houston
};

```

NOTE

<Side Remark: ragged array>

Since the matrix is symmetric for an undirected graph, to save storage, you may use a ragged array.

The adjacency matrix for the directed graph in Figure 13.3(a) can be represented as follows:

```

int[][] a = {{0, 0, 1, 0, 0}, // Peter
             {0, 0, 1, 0, 0}, // Jane
             {0, 0, 0, 0, 1}, // Mark
             {0, 0, 0, 0, 1}, // Cindy
             {0, 0, 0, 0, 0}  // Wendy
            };

```

13.3.5 Representing Edges: Adjacency Lists

To represent edges using adjacency lists, define an array of linked lists. The array has n entries. Each entry represents a vertex. The linked list for vertex i contains all the vertices j such that there is an edge from vertex i to vertex j . For example, to represent the edges in the graph in Figure 13.1, you may create an array of linked list as follows:

```

java.util.LinkedList[] neighbors = new java.util.LinkedList(12);

```

lists[0] contains all vertices adjacent to vertex 0 (i.e., Seattle), lists[1] contains all vertices adjacent to vertex 1 (i.e., San Francisco), and so on, as shown in Figure 13.5.

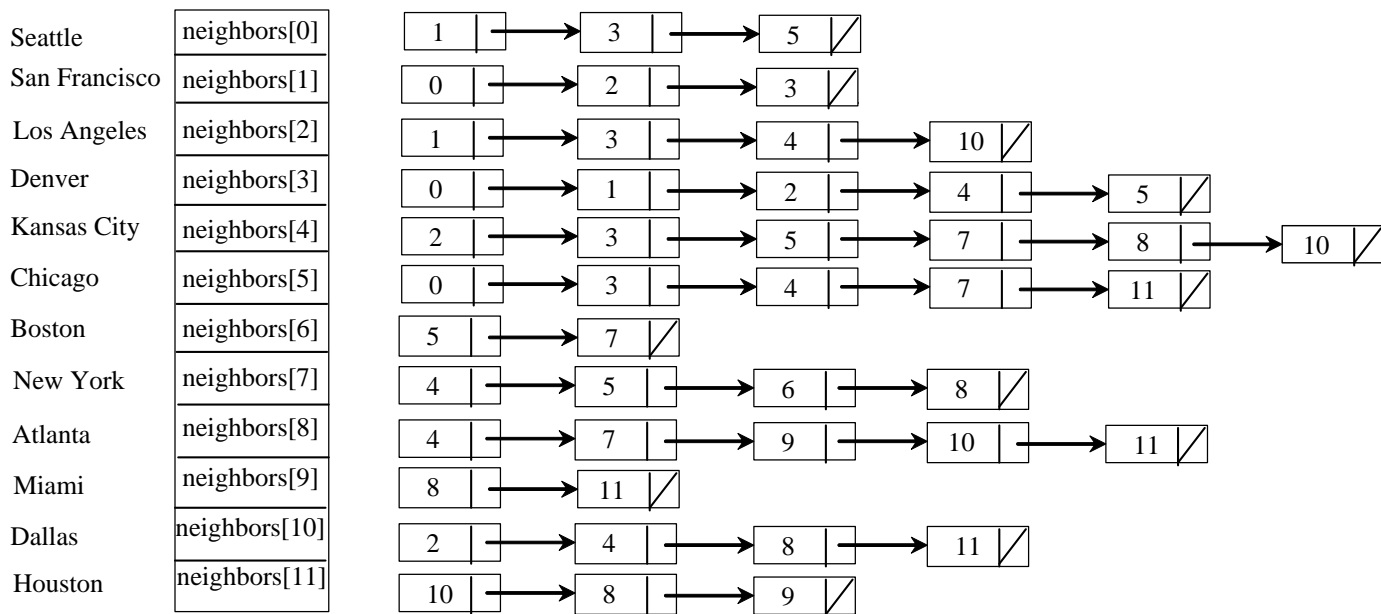


Figure 13.5

Edges in the graph in Figure 13.1 are represented using linked lists.

To represent the edges in the graph in Figure 13.3(a), you may create an array of linked list as follows:

```
java.util.LinkedList[] lists = new java.util.LinkedList(5);
```

`lists[0]` contains all vertices pointed from vertex 0 via directed edges, `lists[1]` contains all vertices pointed from vertex 1 via directed edges, and so on, as shown in Figure 13.6.

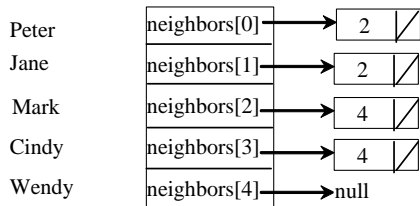


Figure 13.6

Edges in the graph in Figure 13.3(a) are represented using linked lists.

NOTE:

<Side Remark: adjacency matrices vs. adjacency lists>

You can represent a graph using an adjacency matrix or adjacency lists. Which one is better? If the graph is dense (i.e., there are a lot of edges), using an adjacency matrix is preferred. If the graph is very sparse (i.e., very few

edges), using adjacency lists is better, because it would waste a lot of space using an adjacency matrix.

Both adjacency matrices and adjacency lists may be used in the same time to make algorithms more efficient. For example, it takes $O(1)$ constant time to check where two vertices are connected using an adjacency matrix and it takes linear time $O(m)$ to print all edges in a graph using adjacency lists, where m is the number of edges.

*****END NOTE**

NOTE:

<Side Remark: other representations>

Adjacency matrices and adjacency lists are two common representations for graphs, but are not the only ways for representing graphs. There are other ways to represent graphs. For example, you may define a vertex as an object with a method `getNeighbors()` that returns all its neighbors. For simplicity, the text will use adjacency matrices and adjacency lists to represent graphs. Other representations will be explored in the exercises.

*****END NOTE**

13.4 Modeling Graphs

The Java Collections Framework serves a good example for designing complex data structures. The common features of data structures are defined in the interfaces (e.g., `Collection`, `Set`, `List`), as shown in Figure 23.1. Abstract classes (e.g., `AbstractCollection`, `AbstractSet`, `AbstractList`) partially implement the interfaces. Concrete classes (e.g., `HashSet`, `LinkedHashSet`, `TreeSet`, `ArrayList`, `LinkedList`) provide concrete implementations. This design pattern is useful to design graphs. We will define an interface named `Graph` that contains all common operations of graphs and an abstract class named `AbstractGraph` that partially implements the `Graph` interface. Many concrete graphs may be added to the design. For example, we will define such graphs named `UnweightedGraph` and `WeightedGraph`. The relationships of these interfaces and classes are illustrated in Figure 13.7.

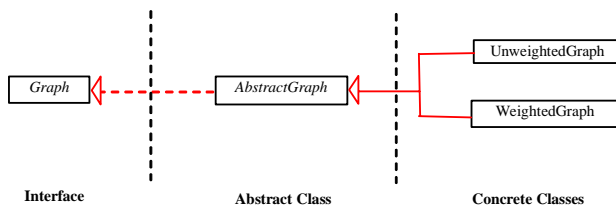


Figure 13.7

Graphs can be modeled using interfaces, abstract classes and concrete classes.

What are common operations for a graph? In general, you need to get the number of vertices in a graph, get all vertices in a graph, get the vertex object with a specified index, get the index of the vertex with a specified name, get the neighbors for a vertex, get the adjacency matrix, get the degree for a vertex, perform a depth-first search, and perform a breadth-first search. Depth-first search and breadth-first search will be introduced in the following section. Figure 13.8 illustrates these methods in the UML diagram.

<PD: UML Class Diagram>

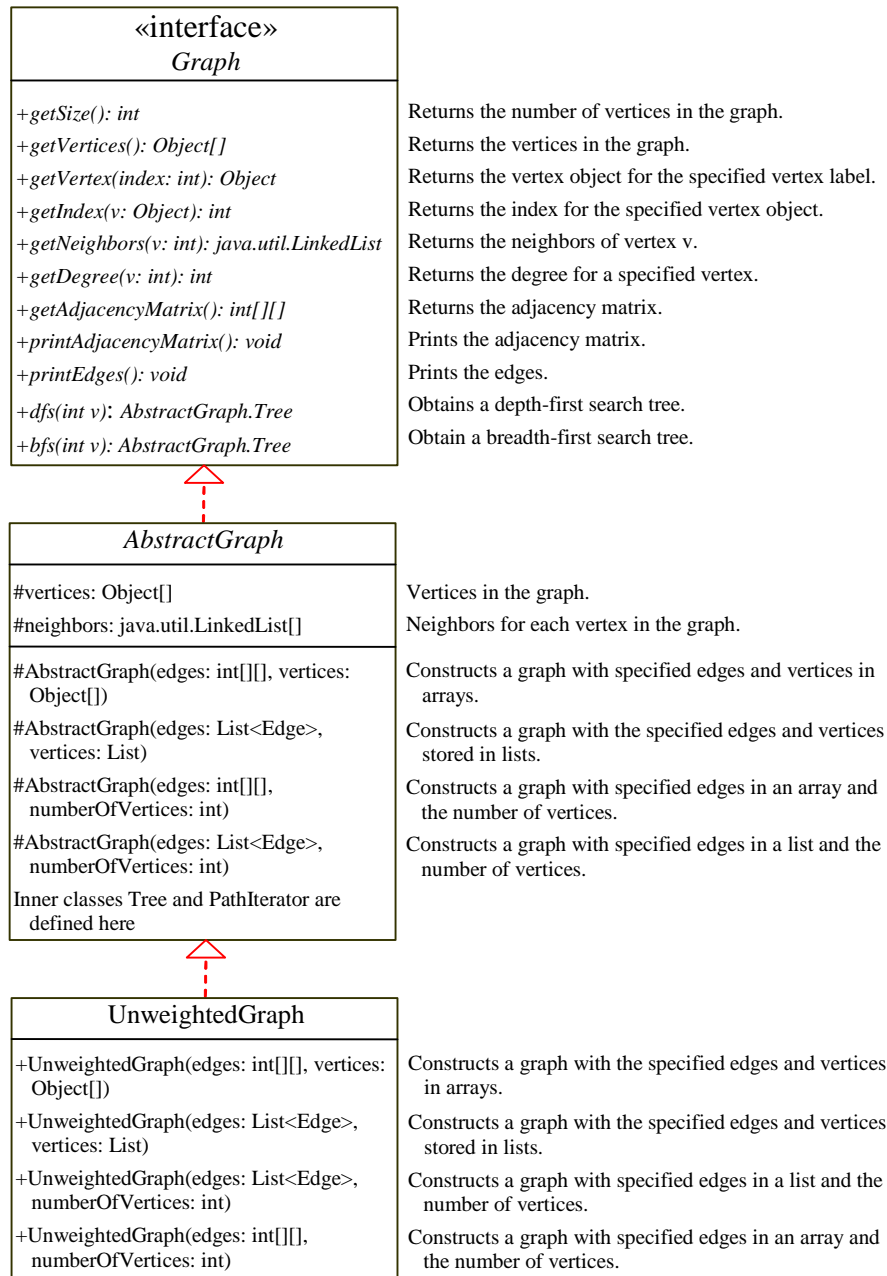


Figure 13.8

The Graph interface defines the common operations for all types of graphs.

AbstractGraph does not introduce any new methods. An array of vertices and an array of adjacency lists for the vertices are defined in the AbstractGraph class. With these data fields, it is sufficient to implement all the methods defined in the Graph interface.

UnweightedGraph simply extends AbstractGraph with four constructors for creating the concrete Graph instances. UnweightedGraph inherits all methods from AbstractGraph and it does not introduce any new methods.

NOTE:

<Side Remark: why AbstractGraph?>

The AbstractGraph class implements all the methods in the Graph interface. So, why is it defined abstract? In the future, you may need to add new methods into the Graph interface that cannot be implemented in AbstractGraph. To make the classes easy to maintain, it is desirable to define the AbstractGraph class abstract.

*****END NOTE**

Assume all these interfaces and classes are available. Listing 13.1 gives a test program that creates a graph for the one in Figure 13.1 and another graph for the one in Figure 13.3(a).

Listing 13.1 TestGraph.java

*****PD: Please add line numbers in the following code*****
*****Layout: Please layout exactly. Don't skip the space. This is true for all source code in the book. Thanks, AU.**
<Side Remark line 3: vertices>
<Side Remark line 7: edges>
<Side Remark line 22: create a graph>
<Side Remark line 24: print adjacency matrix>
<Side Remark line 26: print edges>
<Side Remark line 29: create a graph>
<Side Remark line 31: print adjacency matrix>
<Side Remark line 33: print edges>

```
public class TestGraph {
    public static void main(String[] args) {
        String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
            "Denver", "Kansas City", "Chicago", "Boston", "New York",
            "Atlanta", "Miami", "Dallas", "Houston"};

        int[][] edges = {
            {0, 1}, {0, 3}, {0, 5},
            {1, 0}, {1, 2}, {1, 3},
            {2, 1}, {2, 3}, {2, 4}, {2, 10},
            {3, 0}, {3, 2}, {3, 4}, {3, 5},
            {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
```

```

        {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
        {6, 5}, {6, 7},
        {7, 4}, {7, 5}, {7, 6}, {7, 8},
        {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
        {9, 8}, {9, 11},
        {10, 2}, {10, 4}, {10, 8}, {10, 11},
        {11, 8}, {11, 9}, {11, 10}
    };

    Graph graph1 = new UnweightedGraph(edges, vertices);
    System.out.println("Adjacency matrix for graph1:");
    graph1.printAdjacencyMatrix();
    System.out.println("The edges for graph1:");
    graph1.printEdges();

    edges = new int[][]{{0, 2}, {1, 2}, {2, 4}, {3, 4}};
    Graph graph2 = new UnweightedGraph(edges, 5);
    System.out.println("\nAdjacency matrix for graph2:");
    graph2.printAdjacencyMatrix();
    System.out.println("The edges for graph2:");
    graph2.printEdges();
}
}

```

<Output>

Adjacency matrix for graph1:

```

0 1 0 1 0 1 0 0 0 0 0 0
1 0 1 1 0 0 0 0 0 0 0 0
0 1 0 1 1 0 0 0 0 0 1 0
1 0 1 0 1 1 0 0 0 0 0 0
0 0 1 1 0 1 0 1 1 0 1 0
1 0 0 1 1 0 1 1 0 0 0 0
0 0 0 0 0 1 0 1 0 0 0 0
0 0 0 0 1 1 1 0 1 0 0 0
0 0 0 0 1 0 0 1 0 1 1 1
0 0 0 0 0 0 0 0 1 0 0 1
0 0 1 0 1 0 0 0 1 0 0 1
0 0 0 0 0 0 0 0 1 1 1 0

```

The edges for graph1:

```

Vertex 0: (0, 1) (0, 3) (0, 5)
Vertex 1: (1, 0) (1, 2) (1, 3)
Vertex 2: (2, 1) (2, 3) (2, 4) (2, 10)
Vertex 3: (3, 0) (3, 2) (3, 4) (3, 5)
Vertex 4: (4, 2) (4, 3) (4, 5) (4, 7) (4, 8) (4, 10)
Vertex 5: (5, 0) (5, 3) (5, 4) (5, 6) (5, 7)
Vertex 6: (6, 5) (6, 7)
Vertex 7: (7, 4) (7, 5) (7, 6) (7, 8)
Vertex 8: (8, 4) (8, 7) (8, 9) (8, 10) (8, 11)
Vertex 9: (9, 8) (9, 11)
Vertex 10: (10, 2) (10, 4) (10, 8) (10, 11)

```

Vertex 11: (11, 8) (11, 9) (11, 10)

Adjacency matrix for graph2:

```
0 0 1 0 0
0 0 1 0 0
0 0 0 0 1
0 0 0 0 1
0 0 0 0 0
```

The edges for graph2:

Vertex 0: (0, 2)

Vertex 1: (1, 2)

Vertex 2: (2, 4)

Vertex 3: (3, 4)

Vertex 4:

<End Output>

The program creates graph1 for the graph in Figure 13.1 in lines 1-20. The vertices for graph1 are defined in lines 3-5. The edges for graph1 are defined in 5-11. 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]. For example, the first row {0, 1} represents the edge from vertex 0 (edges[0][0]) to vertex 1 (edges[0][1]). The row {0, 5} represents the edge from vertex 0 (edges[1][0]) to vertex 5 (edges[1][1]). Line 22 invokes the printAdjacencyMatrix() method on graph1 to display the adjacency matrix for graph1. Line 24 invokes the printEdges() method on graph1 to display all edges in graph1.

The program creates graph2 for the graph in Figure 13.3(a) in lines 26-33. The edges for graph2 are defined in line 26. Line 29 invokes the printAdjacencyMatrix() method on graph2 to display the adjacency matrix for graph1. Line 31 invokes the printEdges() method on graph2 to display all edges in graph2.

Now we turn our attention to implementing the interface and classes. Listings 13.2, 13.3, and 13.4 give the Graph interface, the AbstractGraph class, and the UnweightedGraph class, respectively.

Listing 13.2 Graph.java

```
***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don't skip the space.
This is true for all source code in the book. Thanks, AU.
<Side Remark line 3: getSize>
<Side Remark line 6: getVertices>
<Side Remark line 9: getVertex>
<Side Remark line 12: getIndex>
<Side Remark line 15: getNeighbors>
<Side Remark line 18: getDegree>
<Side Remark line 21: getAdjacencyMatrix>
<Side Remark line 24: printAdjacencyMatrix>
<Side Remark line 27: printEdges>
<Side Remark line 30: dfs>
```

<Side Remark line 33: bfs>

```
public interface Graph {  
    /** Return the number of vertices in the graph */  
    public int getSize();  
  
    /** Return the vertices in the graph */  
    public Object[] getVertices();  
  
    /** Return the object for the specified vertex index */  
    public Object getVertex(int index);  
  
    /** Return the index for the specified vertex object */  
    public int getIndex(Object v);  
  
    /** Return the neighbors of vertex v */  
    public java.util.List getNeighbors(int v);  
  
    /** Return the degree for a specified vertex */  
    public int getDegree(int v);  
  
    /** Return the adjacency matrix */  
    public int[][] getAdjacencyMatrix();  
  
    /** Print the adjacency matrix */  
    public void printAdjacencyMatrix();  
  
    /** Print the edges */  
    public void printEdges();  
  
    /** Obtain a depth-first search tree */  
    public AbstractGraph.Tree dfs(int v);  
  
    /** Obtain a breadth-first search tree */  
    public AbstractGraph.Tree bfs(int v);  
}
```

Listing 13.3 AbstractGraph.java

*****PD: Please add line numbers in the following code*****
*****Layout: Please layout exactly. Don't skip the space.**
This is true for all source code in the book. Thanks, AU.

<Side Remark line 8: constructor>
<Side Remark line 14: constructor>
<Side Remark line 20: constructor>
<Side Remark line 29: constructor>
<Side Remark line 68: getSize>
<Side Remark line 73: getVertices>
<Side Remark line 68: getVertex>
<Side Remark line 83: getIndex>
<Side Remark line 94: getNeighbors>
<Side Remark line 99: getDegrees>
<Side Remark line 104: getAdjacencyMatrix>
<Side Remark line 118: printAdjacencyMatrix >

<Side Remark line 130: [printEdges](#)>
 <Side Remark line 142: [Edge inner class](#)>
 <Side Remark line 155: [dfs method](#)>
 <Side Remark line 188: [bfs method](#)>
 <Side Remark line 217: [Tree inner class](#)>

```

import java.util.*;

public abstract class AbstractGraph implements Graph {
    protected Object[] vertices; // Store vertices
    protected List<Integer>[] neighbors; // Adjacency lists

    /** Construct a graph from edges and vertices stored in arrays */
    protected AbstractGraph(int[][] edges, Object[] vertices) {
        this.vertices = vertices;
        createAdjacencyLists(edges, vertices.length);
    }

    /** Construct a graph from edges and vertices stored in ArrayList */
    protected AbstractGraph(List<Edge> edges, List vertices) {
        this.vertices = vertices.toArray();
        createAdjacencyLists(edges, vertices.size());
    }

    /** Construct a graph from edges and vertices in ArrayList */
    protected AbstractGraph(List<Edge> edges, int numberOfVertices) {
        vertices = new Integer[numberOfVertices]; // Create vertices
        for (int i = 0; i < numberOfVertices; i++) {
            vertices[i] = new Integer(i); // vertices is {0, 1, 2, ...}
        }
        createAdjacencyLists(edges, numberOfVertices);
    }

    /** Construct a graph from edges in array */
    protected AbstractGraph(int[][] edges, int numberOfVertices) {
        vertices = new Integer[numberOfVertices]; // Create vertices
        for (int i = 0; i < numberOfVertices; i++) {
            vertices[i] = new Integer(i); // vertices is {0, 1, 2, ...}
        }
        createAdjacencyLists(edges, numberOfVertices);
    }

    /** Create adjacency lists for each vertex */
    private void createAdjacencyLists(
        int[][] edges, int numberOfVertices) {
        // Create a linked list
        neighbors = new LinkedList[numberOfVertices];
        for (int i = 0; i < numberOfVertices; i++) {
            neighbors[i] = new java.util.LinkedList<Integer>();
        }

        for (int i = 0; i < edges.length; i++) {
            int u = edges[i][0];
            int v = edges[i][1];
            neighbors[u].add(v);
        }
    }
}

```

```

    }
}

/** Create adjacency lists for each vertex */
private void createAdjacencyLists(
    List<Edge> edges, int numberOfVertices) {
    // Create a linked list
    neighbors = new LinkedList[numberOfVertices];
    for (int i = 0; i < numberOfVertices; i++) {
        neighbors[i] = new java.util.LinkedList<Integer>();
    }

    for (Edge edge: edges) {
        neighbors[edge.u].add(edge.v);
    }
}

/** Return the number of vertices in the graph */
public int getSize() {
    return vertices.length;
}

/** Return the vertices in the graph */
public Object[] getVertices() {
    return vertices;
}

/** Return the object for the specified vertex */
public Object getVertex(int v) {
    return vertices[v];
}

/** Return the index for the specified vertex object */
public int getIndex(Object vertex) {
    for (int i = 0; i < getSize(); i++) {
        if (vertex.equals(vertices[i])) {
            return i;
        }
    }

    return -1; // If vertex is not in the graph
}

/** Return the neighbors of vertex v */
public java.util.List getNeighbors(int v) {
    return neighbors[v];
}

/** Return the degree for a specified vertex */
public int getDegree(int v) {
    return neighbors[v].size();
}

/** Return the adjacency matrix */
public int[][] getAdjacencyMatrix() {
    int[][] adjacencyMatrix = new int[getSize()][getSize()];
}

```

```

    for (int i = 0; i < neighbors.length; i++) {
        for (int j = 0; j < neighbors[i].size(); j++) {
            int v = neighbors[i].get(j);
            adjacencyMatrix[i][v] = 1;
        }
    }

    return adjacencyMatrix;
}

/** Print the adjacency matrix */
public void printAdjacencyMatrix() {
    int[][] adjacencyMatrix = getAdjacencyMatrix();
    for (int i = 0; i < adjacencyMatrix.length; i++) {
        for (int j = 0; j < adjacencyMatrix[0].length; j++) {
            System.out.print(adjacencyMatrix[i][j] + " ");
        }

        System.out.println();
    }
}

/** Print the edges */
public void printEdges() {
    for (int u = 0; u < neighbors.length; u++) {
        System.out.print("Vertex " + u + ": ");
        for (int j = 0; j < neighbors[u].size(); j++) {
            System.out.print("(" + u + ", " +
                neighbors[u].get(j) + ") ");
        }
        System.out.println();
    }
}

/** Edge inner class inside the AbstractGraph class */
public static class Edge {
    public int u; // Starting vertex of the edge
    public int v; // Ending vertex of the edge

    /** Construct an edge for (u, v) */
    public Edge(int u, int v) {
        this.u = u;
        this.v = v;
    }
}

/** Obtain a DFS tree starting from vertex v */
/** To be discussed in Section 13.6 */
public Tree dfs(int v) {
    List<Integer> searchOrders = new ArrayList<Integer>();
    int[] parent = new int[vertices.length];
    for (int i = 0; i < parent.length; i++)
        parent[i] = -1; // Initialize parent[i] to -1

    // Mark visited vertices

```

```

    boolean[] isVisited = new boolean[vertices.length];

    // Recursively search
    dfs(v, parent, searchOrders, isVisited);

    // Return a search tree
    return new Tree(v, parent, searchOrders);
}

/** Recursive method for DFS search */
private void dfs(int v, int[] parent, List<Integer> searchOrders,
    boolean[] isVisited) {
    // Store the visited vertex
    searchOrders.add(v);
    isVisited[v] = true; // Vertex v visited

    for (int i : neighbors[v]) {
        if (!isVisited[i]) {
            parent[i] = v; // The parent of vertex i is v
            dfs(i, parent, searchOrders, isVisited); // Recursive search
        }
    }
}

/** Starting bfs search from vertex v */
/** To be discussed in Section 13.7 */
public Tree bfs(int v) {
    List<Integer> searchOrders = new ArrayList<Integer>();
    int[] parent = new int[vertices.length];
    for (int i = 0; i < parent.length; i++)
        parent[i] = -1; // Initialize parent[i] to -1

    java.util.LinkedList<Integer> queue =
        new java.util.LinkedList<Integer>(); // list used as a queue
    boolean[] isVisited = new boolean[vertices.length];
    queue.offer(v); // Enqueue v
    isVisited[v] = true; // Mark it visited

    while (!queue.isEmpty()) {
        int u = queue.poll(); // Dequeue to u
        searchOrders.add(u); // u searched
        for (int w : neighbors[u]) {
            if (!isVisited[w]) {
                queue.offer(w); // Enqueue w
                parent[w] = u; // The parent of w is u
                isVisited[w] = true; // Mark it visited
            }
        }
    }

    return new Tree(v, parent, searchOrders);
}

/** Tree inner class inside the AbstractGraph class */
/** To be discussed in Section 13.5 */
public class Tree {

```

```

    private int root; // The root of the tree
    private int[] parent; // Store the parent of each vertex
    private List<Integer> searchOrders; // Store the search order

    /** Construct a tree with root, parent, and searchOrder */
    public Tree(int root, int[] parent, List<Integer> searchOrders) {
        this.root = root;
        this.parent = parent;
        this.searchOrders = searchOrders;
    }

    /** Construct a tree with root and parent without a
     * particular order */
    public Tree(int root, int[] parent) {
        this.root = root;
        this.parent = parent;
    }

    /** Return the root of the tree */
    public int getRoot() {
        return root;
    }

    /** Return the parent of vertex v */
    public int getParent(int v) {
        return parent[v];
    }

    /** Return an array representing search order */
    public List<Integer> getSearchOrders() {
        return searchOrders;
    }

    /** Return number of vertices found */
    public int getNumberOfVerticesFound() {
        return searchOrders.size();
    }

    /** Return the iterator for a path starting from the root to v */
    public java.util.Iterator pathIterator(int v) {
        return new PathIterator(v);
    }

    /** PathIterator inner class inside the tree */
    public class PathIterator implements java.util.Iterator {
        private Stack<Integer> stack;

        /** Construct an iterator for the vertices on the path */
        public PathIterator(int v) {
            stack = new Stack<Integer>();
            do {
                stack.add(v);
                v = parent[v];
            }
            while (v != -1);
        }
    }

```

```

    /** Has next element in the iterator? */
    public boolean hasNext() {
        return !stack.isEmpty();
    }

    /** Get the current element in the iterator and move the
     * iterator to point to the next element */
    public Object next() {
        return vertices[stack.pop()];
    }

    /** This remove method is defined in the Iterator interface
     * do not implement it */
    public void remove() {
        // Do nothing
    }

    /** Print a path from the root to vertex v */
    public void printPath(int v) {
        Iterator iterator = pathIterator(v);
        System.out.print("A path from " + vertices[root] + " to " +
            vertices[v] + ": ");
        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
    }

    /** Print the whole tree */
    public void printTree() {
        System.out.println("Root is: " + vertices[root]);
        System.out.print("Edges: ");
        for (int i = 0; i < parent.length; i++) {
            if (parent[i] != -1) {
                // Display an edge
                System.out.print("(" + vertices[parent[i]] + ", " +
                    vertices[i] + ") ");
            }
        }
        System.out.println();
    }
}

```

Listing 13.4 UnweightedGraph.java

*****PD: Please add line numbers in the following code*****
*****Layout: Please layout exactly. Don't skip the space.**
This is true for all source code in the book. Thanks, AU.
<Side Remark line 7: constructor>
<Side Remark line 14: constructor>
<Side Remark line 21: constructor>
<Side Remark line 28: constructor>

```
import java.util.*;
```

```

public class UnweightedGraph extends AbstractGraph {
    /** Construct a graph with the specified edges and vertices in
    * arrays
    */
    public UnweightedGraph(int[][] edges, Object[] vertices) {
        super(edges, vertices);
    }

    /** Construct a graph with the specified edges and vertices in
    * lists
    */
    public UnweightedGraph(List<Edge> edges, List vertices) {
        super(edges, vertices);
    }

    /** Construct a graph with the specified edges in a list and
    * the number of vertices
    */
    public UnweightedGraph(List<Edge> edges, int numberOfVertices) {
        super(edges, numberOfVertices);
    }

    /** Construct a graph with the specified edges in an array and
    * the number of vertices
    */
    public UnweightedGraph(int[][] edges, int numberOfVertices) {
        super(edges, numberOfVertices);
    }
}

```

The code in the Graph interface in Listing 13.2 and the UnweightedGraph class are straight forward. Let us digest the code in the AbstractGraph class in Listing 13.3.

The AbstractGraph class defines the data field vertices (line 4) to store vertices and neighbors (line 5) to store edges in adjacency lists. neighbors[i] stores all vertices adjacent to vertex i. Four overloaded constructors are defined in lines 8-35 to create graphs from arrays or lists of edges and vertices. The createAdjacencyLists(int[][] edges, int numberOfVertice) method creates adjacency lists from edges in an array (lines 38-51). The createAdjacencyLists(List<Edge> edges, int numberOfVertice) method creates adjacency lists from edges in a list (lines 54-65).

The getAdjacencyMatrix() method (lines 104-115) returns a two-dimensional array for representing an adjacency matrix of the edges. The printAdjacencyMatrix() method (lines 118-127) displays the adjacency matrix. The printEdges() method (lines 130-139) displays all vertices and edges adjacent to each vertex.

The code in lines 155-201 gives the methods for finding a depth-first search tree and a breadth-first search tree, which will be introduced in the following sections.

13.5 Graph Traversals

<Side Remark: depth-first>

<Side Remark: *breadth-first*>

Graph traversal is the process of visiting each vertex in the graph exactly once. There are two popular ways to traverse a graph. They are called *depth-first traversal* (or *depth-first search*) and *breadth-first traversal* (or *breadth-first search*). Both traversals result in a spanning tree, which can be modeled using a class, as shown in Figure 13.9. Note that `Tree` is an inner class defined in the `AbstractGraph` class. `AbstractGraph.Tree` is different from the `Tree` interface defined in §7.2.5. `AbstractGraph.Tree` describes a search order of the nodes in the graph, whereas the `Tree` interface defines common features in a tree.

AbstractGraph.Tree	
-root: int	The root of the tree.
-parent: int[]	The parents of the vertices.
-searchOrders: java.util.List<Integer>	The orders for traversing the vertices.
+Tree(root: int, parent: int[], searchOrders: List<Integer>)	Constructs a tree with the specified root, parent, and searchOrders.
+Tree(root: int, parent: int[])	Constructs a tree with the specified root and parent with no specified searchOrders.
+getRoot(): int	Returns the root of the tree.
+getSearchOrders(): List	Returns the order of vertices searched.
+getParent(v: int): int	Returns the parent of vertex v.
+getNumberOfVerticesFound(): int	Returns the number of vertices searched.
+pathIterator(v: int): java.util.Iterator	Returns an iterator for the path from the root to vertex v.
+printPath(v: int): void	Displays a path from the root to the specified vertex.
+printTree(): void	Displays tree with the root and all edges.

Figure 13.9

The `Tree` class describes a tree with a search order for the nodes.

The `Tree` class is defined as an inner class in the `AbstractGraph` class in lines 219-318 in Listing 13.3.

The `Tree` class defines seven methods. The `getRoot()` method returns the root of the tree. You can get the order of the vertices searched by invoking the `getSearchOrders()` method. You can invoke `getParent(v)` to find the parent of vertex `v` in the search. Invoking `getNumberOfVerticesFound()` returns the number of vertices searched. Invoking `printPath(v)` displays a path from the root to `v`. You can display all edges in the tree using the `printTree()` method.

<Side Remark: *PathIterator* class>

The `Tree` class provides the `pathIterator(int v)` method that returns an iterator for a path from the source vertex to vertex `v`. The iterator is an instance of the `PathIterator` class, which implements `java.util.Iterator` (line 264). The `Iterator` interface defines three methods: `hasNext()`, `next()`, and `remove()`. The `PathIterator` class must implement all these three methods. To avoid actually removing any vertices from the path, the body of the `remove()` method is empty (line 291). The `PathIterator` constructor uses a stack to store vertices from `v` to the root (lines 269-274). `hasNext()` returns true if the stack is not empty (line 279). Invoking `next()` removes a vertex from the stack (line 285). The root is on the top of stack, so it is removed first.

The iterator traverses the vertices for the path from the root to the specified vertex.

Sections 13.6 and 13.7 will introduce depth-first search and breadth-first search, respectively. Both searches will result in an instance of the `Tree` class.

13.6 Depth-First Search

The depth-first search of a graph is like the depth-first search of a tree discussed in §7.2.3, "Tree Traversal." In the case of a tree, the search starts from the root. In a graph, the search can start from any vertex.

A depth-first search of a tree first visits the root, then recursively visits the subtrees of the root. Similarly, the depth-first search of a graph first visits a vertex, then recursively visits all vertices adjacent to that vertex. The difference is that the graph may contain cycles, which may lead to an infinite recursion. To avoid this problem, you need to track the vertices that have already been visited.

The search is called depth-first, because it searches "deeper" in the graph as much as possible. The search starts from some vertex v . After visiting v , the next vertex to be visited is first unvisited neighbor of v . If v has no unvisited neighbor, backtrack to the vertex from which we reached v .

13.6.1 Depth-First Search Algorithm

The algorithm for the depth-first search can be described in Listing 13.7.

Listing 13.7 Depth-first Search Algorithm

```
***PD: Please add line numbers in the following code***  
***Layout: Please layout exactly. Don't skip the space.  
This is true for all source code in the book. Thanks, AU.  
<Side Remark line 2: visit v>  
<Side Remark line 4: check a neighbor>  
<Side Remark line 5: recursive search>
```

```
dfs(vertex v) {  
    visit v;  
    for each neighbor w of v  
        if (w has not been visited) {  
            dfs(w);  
        }  
}
```

You may use an array named `isVisited` to denote whether a vertex has been visited. Initially, `isVisited[i]` is `false` for each vertex i . Once a vertex, say v , is visited, `isVisited[v]` is set to `true`.

Consider the graph in Figure 13.10(a). Suppose you start the depth-first search from vertex 0. First visit 0, then any of its neighbors, say 1. Now 1 is visited, as shown in Figure 13.10(b). Vertex 1 has three neighbors 0, 2, and 4. Since 0 has already been visited, you will

visit either 2 or 4. Let us pick 2. Now 2 is visited, as shown in Figure 13.10(c). 2 has three neighbors 0, 1, and 3. Since 0 and 1 have already been visited, pick 3. 3 is now visited, as shown in Figure 13.10(d). At this point, the vertices have been visited are in this order:

0, 1, 2, 3

Since all the neighbors of 3 have been visited, back track to 2, since all the vertices of 2 have been visited, back track to 1. 4 is adjacent to 1, but 4 has not been visited. So visit 4, as shown in Figure 13.9(e). Since all the neighbors of 4 have been visited, back track to 1. Since all the neighbors of 1 have been visited, back track to 0. Since all the neighbors of 0 have been visited, the search ends.

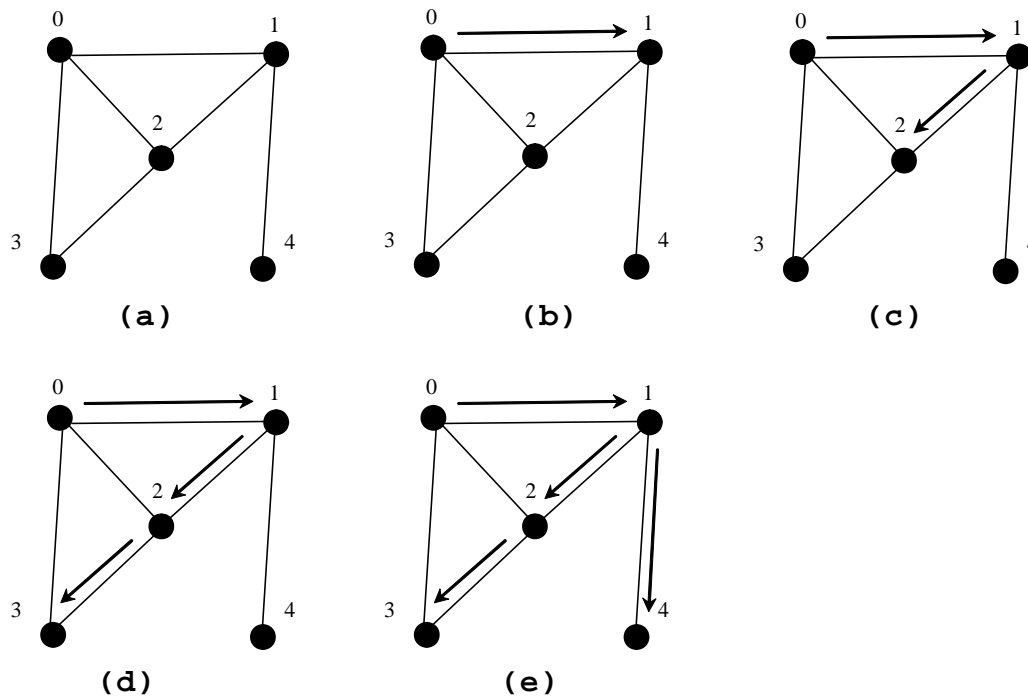


Figure 13.10

Depth-first search visits a node and its neighbors recursively.

<Side Remark: DFS time complexity>

Since each edge and vertex is visited only once, so the time complexity of the `dfs` method is $O(|E| + |V|)$, where $|E|$ denotes the number of edges and $|V|$ denotes the number of vertices.

13.6.2 Implementation of Depth-First Search

The algorithm is described in Listing 13.5 using recursion. It is natural to implement it using recursion. Alternatively, you can implement it using a stack (see Exercise 13.1).

The `dfs(int v)` method is implemented in lines 157-186 in Listing 13.3. It returns an instance of the `Tree` class with vertex `v` as the root. The

method stores the vertices searched in a list `searchOrders` (line 158), the parent of each vertex in an array `parent` (line 159), and uses the `isVisited` array to indicate whether a vertex has been visited (line 164). It invokes the helper method `dfs(v, parent, searchOrders, isVisited)` to perform a depth-first search (line 167).

In the recursive helper method, the search starts from vertex `v`. `v` is added to `searchOrders` in line 177 and is marked visited (line 178). For each unvisited neighbor of `v`, the method is recursively invoked to perform a depth-first search. When a vertex `i` is visited, the parent of `i` is stored in `parent[i]` (line 182). The method returns when all vertices are visited for a connected graph, or in a connected component.

Listing 13.7 gives a test program that displays a DFS for the graph in Figure 13.1 starting from Chicago.

Listing 13.7 TestDFS.java

*****PD: Please add line numbers in the following code*****
*****Layout: Please layout exactly. Don't skip the space.**
This is true for all source code in the book. Thanks, AU.
<Side Remark line 3: vertices>
<Side Remark line 7: edges>
<Side Remark line 22: crate a graph>
<Side Remark line 23: get DFS>
<Side Remark line 25: get search order>

```
public class TestDFS {
    public static void main(String[] args) {
        String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
            "Denver", "Kansas City", "Chicago", "Boston", "New York",
            "Atlanta", "Miami", "Dallas", "Houston"};

        int[][] edges = {
            {0, 1}, {0, 3}, {0, 5},
            {1, 0}, {1, 2}, {1, 3},
            {2, 1}, {2, 3}, {2, 4}, {2, 10},
            {3, 0}, {3, 2}, {3, 4}, {3, 5},
            {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
            {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
            {6, 5}, {6, 7},
            {7, 4}, {7, 5}, {7, 6}, {7, 8},
            {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
            {9, 8}, {9, 11},
            {10, 2}, {10, 4}, {10, 8}, {10, 11},
            {11, 8}, {11, 9}, {11, 10}
        };

        Graph graph = new UnweightedGraph(edges, vertices);
        AbstractGraph.Tree dfs = graph.dfs(5); // Vertex 5 is Chicago

        java.util.List<Integer> searchOrders = dfs.getSearchOrders();
        System.out.println(dfs.getNumberOfVerticesFound() +
            " vertices are searched in this DFS order:");
        for (int i = 0; i < searchOrders.size(); i++)
            System.out.print(graph.getVertex(searchOrders.get(i)) + " ");
    }
}
```

```

    System.out.println();

    for (int i = 0; i < searchOrders.size(); i++)
        if (dfs.getParent(i) != -1)
            System.out.println("parent of " + graph.getVertex(i) +
                " is " + graph.getVertex(dfs.getParent(i)));
    }
}

```

<Output>

12 vertices are searched in this DFS order:
 Chicago Seattle San Francisco Los Angeles Denver Kansas City
 New York Boston Atlanta Miami Houston Dallas

parent of Seattle is Chicago
 parent of San Francisco is Seattle
 parent of Los Angeles is San Francisco
 parent of Denver is Los Angeles
 parent of Kansas City is Denver
 parent of Chicago is Chicago
 parent of Boston is New York
 parent of New York is Kansas City
 parent of Atlanta is New York
 parent of Miami is Atlanta
 parent of Dallas is Houston
 parent of Houston is Miami

<End Output>

The graphical illustration of the DFS starting from Chicago is shown in Figure 13.12.

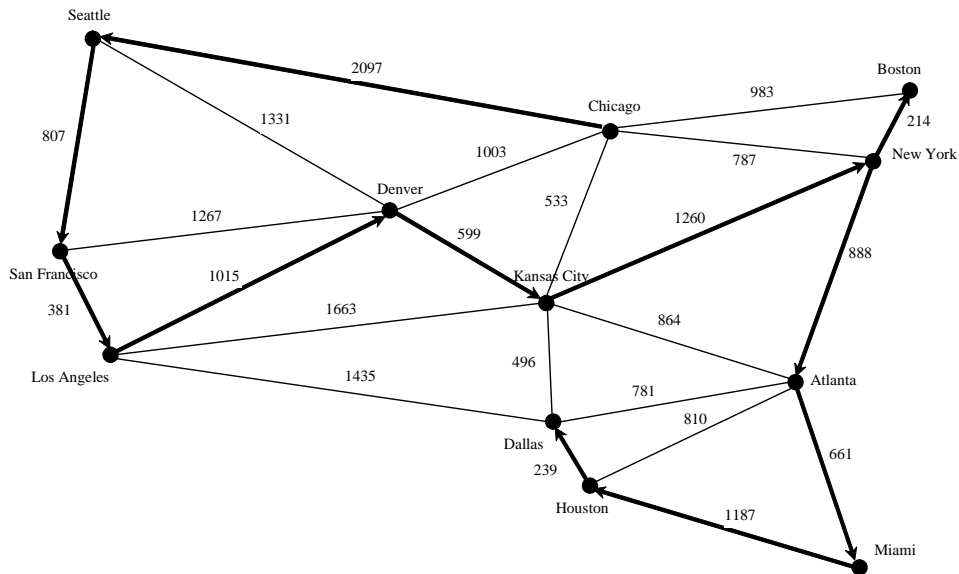


Figure 13.12
 DFS search starts from Chicago.

13.6.4 Applications of the DFS

The depth-first search can be used to solve the following problems:

- Detecting whether a graph is connected. A graph is connected if there is a path between any two vertices in the graph. (See Exercise 13.1)
- Detecting whether there is a path between two vertices. (See Exercise 13.2)
- Finding a path between two vertices. (See Exercise 13.3)
- Finding all connected components. A connected component is a maximal connected subgraph in which every pair of vertices are connected by a path. (See Exercise 13.4)
- Detecting whether there is a cycle in the graph. (See Exercise 13.5)
- Finding a cycle in the graph. (See Exercise 13.5)

The first four problems can be easily solved using the `DFS` class in Listing 13.6. To detect or find a cycle in the graph, you have to slightly modify the `dfs` method.

13.7 Breadth-First Search

The breadth-first traversal of a graph is like the breadth-first traversal of a tree discussed in §7.2.3, "Tree Traversal." With breadth-first traversal of a tree, the nodes are visited level by level. First the root is visited, then all the children of the root, then the grandchildren of the root from left to right, and so on. Similarly, the breadth-first search of a graph first visits a vertex, then all its adjacent vertices, then all the vertices adjacent to those vertices, and so on. To ensure that each vertex is visited only once, skip a vertex if it has already been visited.

13.7.1 Breadth-First Search Algorithm

The algorithm for the breadth-first search starting from vertex v in a graph can be described in Listing 13.8.

Listing 13.8 Breadth-first Search Algorithm

```
***PD: Please add line numbers in the following code***  
***Layout: Please layout exactly. Don't skip the space.  
This is true for all source code in the book. Thanks, AU.  
<Side Remark line 2: create a queue>  
<Side Remark line 3: enqueue v>  
<Side Remark line 7: dequeue into u>  
<Side Remark line 8: visit u>  
<Side Remark line 9: check a neighbor w>  
<Side Remark line 10: is w visited?>  
<Side Remark line 11: enqueue w>
```

```
    bfs(vertex v) {
```

```

    create an empty queue for storing vertices to be visited;
    add v into the queue;
    mark v visited;

    while the queue is not empty {
        dequeue a vertex, say u, from the queue
        visit u;
        for each neighbor w of u
            if w has not been visited {
                add w into the queue;
                mark w visited;
            }
    }
}

```

Consider the graph in Figure 13.13(a). Suppose you start the depth-first search from vertex 0. First visit 0, then all its all visited neighbors 1, 2, and 3, as shown in 13.13(b). Vertex 1 has three neighbors 0, 2, and 4. Since 0 and 2 have already been visited, you will visit just 4 now, as shown in Figure 13.13(c). Vertex 2 has three neighbors 0, 1, and 3, which are all visited. Vertex 3 has three neighbors 0, 2, and 4, which are all visited. Vertex 4 has two neighbors 1 and 3, which are all visited. So, the search ends.

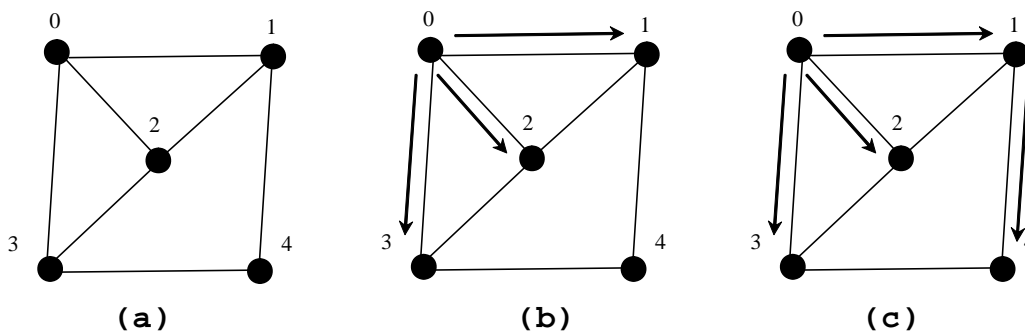


Figure 13.13

Breadth-first search visits a node, then its neighbors, and then its neighbours' neighbors, and so on.

<Side Remark: BFS time complexity>

Since each edge and vertex is visited only once, so the time complexity of the `bfs` method is $O(|E| + |V|)$, where $|E|$ denotes the number of edges and $|V|$ denotes the number of vertices.

13.7.2 Implementation of Breadth-First Search

The `bfs(int v)` method is defined in the `Graph` interface and implemented in the `AbstractGraph` class (lines 190-215). It returns an instance of the `Tree` class with vertex `v` as the root. The method stores the vertices searched in a list `searchOrders` (line 191), the parent of each vertex in an array `parent` (line 192), uses a linked list for a queue (line 196), and uses the `isVisited` array to indicate whether a vertex has been visited (line 198). The search starts from vertex `v`. `v` is added to the queue in line 199 and is marked visited (line 200). The method now examines each vertex `u` in the queue (line 202) and adds it

to `searchOrders` (line 204). The method adds each unvisited neighbor `w` of `u` to the queue (line 207), set its parent to `u` (line 208), and mark it visited (line 209).

Listing 13.10 gives a test program that displays a BFS for the graph in Figure 13.1 starting from Chicago.

Listing 13.10 TestBFS.java

*****PD: Please add line numbers in the following code*****
*****Layout: Please layout exactly. Don't skip the space.**
This is true for all source code in the book. Thanks, AU.
<Side Remark line 3: vertices>
<Side Remark line 7: edges>
<Side Remark line 22: create a graph>
<Side Remark line 23: create a bfs tree>
<Side Remark line 25: get search order>

```
public class TestBFS {
    public static void main(String[] args) {
        String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
            "Denver", "Kansas City", "Chicago", "Boston", "New York",
            "Atlanta", "Miami", "Dallas", "Houston"};

        int[][] edges = {
            {0, 1}, {0, 3}, {0, 5},
            {1, 0}, {1, 2}, {1, 3},
            {2, 1}, {2, 3}, {2, 4}, {2, 10},
            {3, 0}, {3, 2}, {3, 4}, {3, 5},
            {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
            {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
            {6, 5}, {6, 7},
            {7, 4}, {7, 5}, {7, 6}, {7, 8},
            {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
            {9, 8}, {9, 11},
            {10, 2}, {10, 4}, {10, 8}, {10, 11},
            {11, 8}, {11, 9}, {11, 10}
        };

        Graph graph = new UnweightedGraph(edges, vertices);
        AbstractGraph.Tree bfs = graph.bfs(5); // Vertex 5 is Chicago

        java.util.List<Integer> searchOrders = bfs.getSearchOrders();
        System.out.println(bfs.getNumberOfVerticesFound() +
            " vertices are searched in this order:");
        for (int i = 0; i < searchOrders.size(); i++)
            System.out.println(graph.getVertex(searchOrders.get(i)));

        for (int i = 0; i < searchOrders.size(); i++)
            if (bfs.getParent(i) != -1)
                System.out.println("parent of " + graph.getVertex(i) +
                    " is " + graph.getVertex(bfs.getParent(i)));
    }
}
```

<Output>

12 vertices are searched in this BFS order:

Chicago Seattle Denver Kansas City Boston New York San Francisco
Los Angeles Atlanta Dallas Miami Houston

parent of Seattle is Chicago

parent of San Francisco is Seattle

parent of Los Angeles is Denver

parent of Denver is Chicago

parent of Kansas City is Chicago

parent of Chicago is Seattle

parent of Boston is Chicago

parent of New York is Chicago

parent of Atlanta is Kansas City

parent of Miami is Atlanta

parent of Dallas is Kansas City

parent of Houston is Atlanta

<End Output>

The graphical illustration of the BFS starting from Chicago is shown in Figure 13.15.

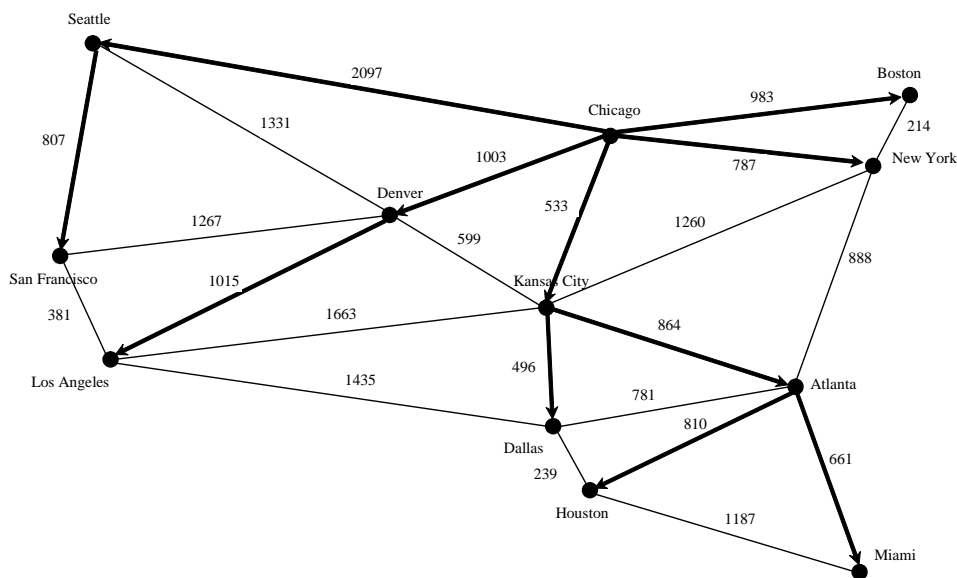


Figure 13.15

BFS search starts from Chicago.

13.7.3 Applications of the BFS

Many of the problems solved by the DFS can also be solved using the breadth-first search. Specifically, the BFS can be used to solve the following problems:

- Detecting whether a graph is connected. A graph is connected if there is a path between any two vertices in the graph.
- Detecting whether there is a path between two vertices.

- Finding a shortest path between two vertices. You can prove that the path between the root and any node in the BFS tree is the shortest path between the root and the node (see Review Question 13.10).
- Finding all connected components. A connected component is a maximal connected subgraph in which every pair of vertices are connected by a path.
- Detecting whether there is a cycle in the graph. (See Exercise 13.4)
- Finding a cycle in the graph. (See Exercise 13.5)
- Testing whether a graph is bipartite. A graph is bipartite if the vertices of the graph can be divided into two disjoint sets such that no edges exist between vertices in the same set. (See Exercise 13.8)

13.8 Case Study: The Nine Tail Problem

The DFS and BFS algorithms have many applications. This section applies the BFS to solve the nine tail problem.

The problem is stated as follows. Nine coins are placed in a three by three matrix with some face up and some face down. A legal move is to take any coin that is face up and reverse it, together with the coins adjacent to it (this does not include coins that are diagonally adjacent). Your task is to find the minimum number of the moves that lead to all coins face down. For example, you start with the nine coins as shown in Figure 13.16(a). After flipping the second coin in the last row, the nine coins are now shown in Figure 13.16(b). After flipping the second coin in the first row, the nine coins are all face down as shown in Figure 13.16(c).

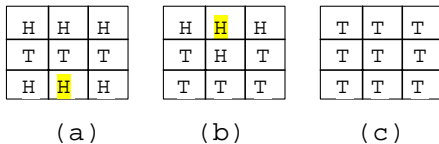
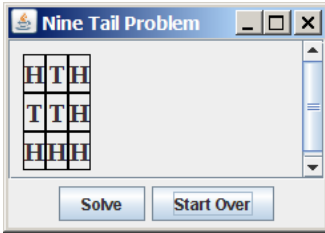


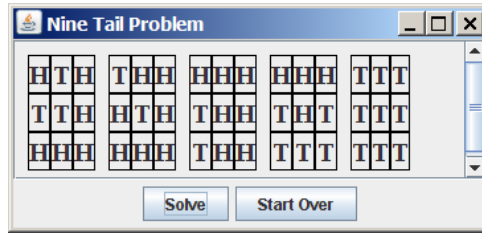
Figure 13.16

The problem is solved when all coins are faced down.

We will write a Java applet that lets the user set an initial state of the nine coins (see Figure 13.17(a)) and click the *Solve* button to display the solution, as shown in Figure 13.17(b). Initially, the user can click the mouse button to flip a coin.



(a)



(b)

Figure 13.17

The applet solves the nine tail problem.

We will create two classes: `NineTailApp` and `NineTailModel`. The `NineTailApp` class is responsible for user interaction and for displaying the solution and the `NineTailModel` class creates a graph for modeling this problem, as shown in Figure 13.18.

<PD: UML Class Diagram>

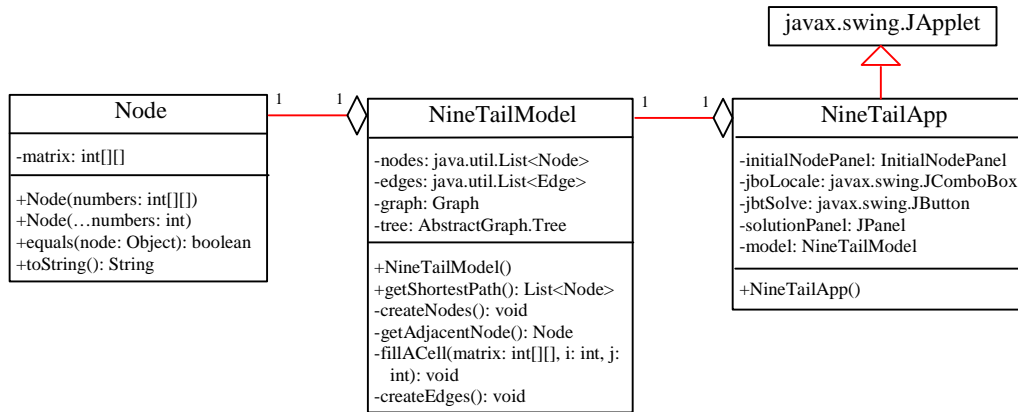


Figure 13.18

`NineTailModel` creates a graph for modeling the problem and `NineTailApp` presents the model in a view.

Each state of the nine coins represents a node in the graph. For example, the three states in Figure 13.16 correspond to three nodes in the graph. For convenience, we use a 3×3 matrix to represent all nodes and use 0 for head and 1 for tail. Since there are nine cells and each cell is either 0 or 1, there are total 2^9 (512) nodes labeled 0, 1, ..., and 511, as shown in Figure 13.18.

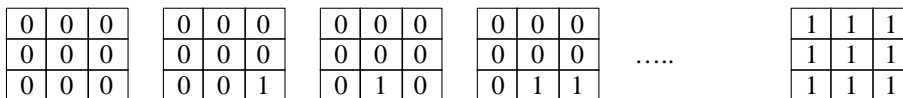


Figure 13.18

There are total 512 nodes and labeled in this order as 0, 1, 2, ..., and 511.

We assign an edge from node u to v if there is a legal move from v to u. The last node in Figure 13.18 represents the state of nine face-down coins. For convenience, we call this last node the *target node*. So the target node is labeled 511. Suppose the initial state of the nine tail problem corresponds to the node u. The nine tail problem is reduced to finding the shortest path from node u to the target node, which is equivalent to finding the path from node u to the target node in a BFS tree rooted at the target node.

The `NineTailModel` class is given in Listing 13.11 to create the nodes and the edges, and obtains a BFS tree rooted from the target node. The `Node` class defines a node (lines 49-94). You can create a `Node` by passing nine numbers as a variable-length argument (line 53) or as a two dimensional array (line 63). The `createNodes()` method (lines 25-46) creates all nodes and adds them to the `nodes` list. The `createEdges()` method (lines 97-113) creates all edges and adds them to the `edges` list. For each node u, you get a new node v (line 106) by flipping a head cell and its neighboring cells in u, and add an edge (v, u) to the edges list (line 108). The `getShortestPath(Node node)` method (lines 146-156) returns a list that contains all vertices on a path from the specified node to the target node in this order.

Listing 13.11 `NineTailModel.java`

*****PD: Please add line numbers in the following code*****
*****Layout: Please layout exactly. Don't skip the space.**
This is true for all source code in the book. Thanks, AU.

<Side Remark line 4: vertices>
 <Side Remark line 5: edges>
 <Side Remark line 7: declare a graph>
 <Side Remark line 8: declare a tree>
 <Side Remark line 12: create nodes>
 <Side Remark line 13: create edges>
 <Side Remark line 16: create graph>
 <Side Remark line 19: create tree>
 <Side Remark line 23: create 512 nodes>
 <Side Remark line 47: Node inner class>
 <Side Remark line 51: construct a node>
 <Side Remark line 61: construct a node>
 <Side Remark line 66: compare two nodes>
 <Side Remark line 80: string representation>
 <Side Remark line 95: create edges>
 <Side Remark line 114: get adjacent node>
 <Side Remark line 132: flip a cell>
 <Side Remark line 144: shortest path>

```
import java.util.*;

public class NineTailModel {
    private ArrayList<Node> nodes = new ArrayList<Node>(); // Vertices
    private ArrayList<AbstractGraph.Edge> edges =
        new ArrayList<AbstractGraph.Edge>(); // Store edges
    private UnweightedGraph graph; // Define a graph
    private AbstractGraph.Tree tree; // Define a tree
```

```

    /** Construct a model */
    public NineTailModel() {
        createNodes(); // Create nodes
        createEdges(); // Create edges

        // Create a graph
        graph = new UnweightedGraph(edges, nodes);

        // Obtain a tree rooted at the target node
        tree = graph.bfs(511);
    }

    /** Create all nodes for the graph */
    private void createNodes() {
        for (int k1 = 0; k1 <= 1; k1++) {
            for (int k2 = 0; k2 <= 1; k2++) {
                for (int k3 = 0; k3 <= 1; k3++) {
                    for (int k4 = 0; k4 <= 1; k4++) {
                        for (int k5 = 0; k5 <= 1; k5++) {
                            for (int k6 = 0; k6 <= 1; k6++) {
                                for (int k7 = 0; k7 <= 1; k7++) {
                                    for (int k8 = 0; k8 <= 1; k8++) {
                                        for (int k9 = 0; k9 <= 1; k9++) {
                                            nodes.add(new Node(k1, k2, k3, k4,
                                                                    k5, k6, k7, k8, k9));
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }

    /** Node inner class */
    public static class Node {
        int[][] matrix = new int[3][3];

        /** Construct a Node from nine numbers */
        Node(int ...numbers) { // Variable-length argument
            int k = 0;
            for (int i = 0; i < 3; i++) {
                for (int j = 0; j < 3; j++) {
                    matrix[i][j] = numbers[k++];
                }
            }
        }

        /** Construct a Node from nine numbers in a 3 by 3 array */
        Node(int[][] numbers) {
            this.matrix = numbers;
        }

        /** Override the equals method to compare two nodes */

```

```

    public boolean equals(Object o) {
        int[][] anotherMatrix = ((Node)o).matrix;
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (matrix[i][j] != anotherMatrix[i][j]) {
                    return false;
                }
            }
        }

        return true; // Nodes with the same matrix values
    }

    /** Return a string representation for the node */
    public String toString() {
        StringBuilder result = new StringBuilder();

        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                result.append(matrix[i][j] + " ");
            }
            result.append("\n");
        }

        return result.toString();
    }

    /** Create all edges for the graph */
    private void createEdges() {
        for (Node node : nodes) {
            int u = nodes.indexOf(node); // node index
            int[][] matrix = node.matrix; // matrix for the node

            for (int i = 0; i < 3; i++) {
                for (int j = 0; j < 3; j++) {
                    if (matrix[i][j] == 0) {
                        Node adjacentNode = getAdjacentNode(matrix, i, j);
                        int v = nodes.indexOf(adjacentNode);
                        // Add edge (v, u) for a legal move from node u to node v
                        edges.add(new AbstractGraph.Edge(v, u));
                    }
                }
            }
        }
    }

    /** Get the adjacent node after flipping the cell at i and j */
    private Node getAdjacentNode(int[][] matrix, int i, int j) {
        int[][] matrixOfNextNode = new int[3][3];
        for (int i1 = 0; i1 < 3; i1++) {
            for (int j1 = 0; j1 < 3; j1++) {
                matrixOfNextNode[i1][j1] = matrix[i1][j1];
            }
        }
    }

```

```

        flipACell(matrixOfNextNode, i - 1, j); // Top neighbor
        flipACell(matrixOfNextNode, i + 1, j); // Bottom neighbor
        flipACell(matrixOfNextNode, i, j - 1); // Left neighbor
        flipACell(matrixOfNextNode, i, j + 1); // Right neighbor
        flipACell(matrixOfNextNode, i, j); // Flip self

    }

    return new Node(matrixOfNextNode);
}

/** Change a valid cell from 0 to 1 and 1 to 0 */
private void flipACell(int[][] matrix, int i, int j) {
    if (i >= 0 && i <= 2 && j >= 0 && j <= 2) { // Within boundary
        if (matrix[i][j] == 0) {
            matrix[i][j] = 1; // Flip from 0 to 1
        }
        else {
            matrix[i][j] = 0; // Flip from 1 to 0
        }
    }
}

/** Return the shortest path from the specified node to the root */
public LinkedList<Node> getShortestPath(Node node) {
    Iterator iterator = tree.pathIterator(nodes.indexOf(node));
    LinkedList list = new LinkedList();

    // Insert the vertices on the path starting from the root to list
    while (iterator.hasNext())
        list.addFirst(iterator.next());

    return list;
}
}

```

The NineTailApp class is given in Listing 13.12 to create the GUI. A node in the model is displayed in a NodePanel (lines 58-72) of nine cells. Each cell is a label, defined in the Cell class (lines 75-82). Each cell represents a coin. The InitialNodePanel class (lines 85-115) displays the starting node. The cells in InitialNodePanel are clickable. You can use the mouse to click on a cell in a starting node to flip a coin. The ClickableCell class (lines 118-132) extends Cell to enable the user to click on a cell.

Listing 13.12 NineTailApp.java

*****PD: Please add line numbers in the following code*****
*****Layout: Please layout exactly. Don't skip the space.**
This is true for all source code in the book. Thanks, AU.

<Side Remark line 8: initial node panel>
 <Side Remark line 12: solution panel>
 <Side Remark line 14: model>
 <Side Remark line 19: add initialNodePanel>
 <Side Remark line 20: enable scrolling>
 <Side Remark line 29: button listener>
 <Side Remark line 31: remove all components>
 <Side Remark line 34: get solution>

<Side Remark line 39: display solution>
 <Side Remark line 43: redisplay solutionPanel>
 <Side Remark line 48: button listener>
 <Side Remark line 49: remove all components>
 <Side Remark line 51: add initialNodePanel>
 <Side Remark line 52: redisplay solutionPanel>
 <Side Remark line 58: inner class NodePanel>
 <Side Remark line 61: add cells to panel>
 <Side Remark line 75: inner class Cell>
 <Side Remark line 77: set cell properties>
 <Side Remark line 85: inner class InitialNodePanel>
 <Side Remark line 87: clickable cells>
 <Side Remark line 92: add clickable cells>
 <Side Remark line 100: get cell values>
 <Side Remark line 118: inner class ClickableCell>
 <Side Remark line 121: mouse listener>

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.LineBorder;

public class NineTailApp extends JApplet {
    // Create the initial board
    private InitialNodePanel initialNodePanel = new InitialNodePanel();
    private JButton jbtSolve = new JButton("Solve");
    private JButton jbtStartOver = new JButton("Start Over");
    // solutionPanel holds a sequeunce of panels for displaying nodes
    private JPanel solutionPanel =
        new JPanel(new FlowLayout(FlowLayout.LEFT, 10, 10));
    private NineTailModel model = new NineTailModel();

    /** Initialize UI */
    public NineTailApp() {
        // Place solutionPanel in a scroll pane
        solutionPanel.add(initialNodePanel);
        add(new JScrollPane(solutionPanel), BorderLayout.CENTER);

        // buttonPanel holds two buttons
        JPanel buttonPanel = new JPanel();
        buttonPanel.add(jbtSolve);
        buttonPanel.add(jbtStartOver);
        add(buttonPanel, BorderLayout.SOUTH);

        // Listener for the Solve button
        jbtSolve.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                solutionPanel.removeAll();

                // Get a shortest path
                java.util.List<NineTailModel.Node> list =
                    model.getShortestPath(new NineTailModel.Node(
                        initialNodePanel.getMatrix()));
            }
        });
    }
}

```

```

        // Display nodes in the shortest path
        for (NineTailModel.Node node: list) {
            solutionPanel.add(new NodePanel(node.matrix));
        }

        solutionPanel.revalidate();
    }
});

// Listener for the Start Over button
jbtStartOver.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        solutionPanel.removeAll();
        solutionPanel.add(initialNodePanel); // Display initial node
        solutionPanel.repaint();
    }
});
}

/** An inner class for displaying a node on a panel */
static class NodePanel extends JPanel {
    public NodePanel(int[][] matrix) {
        this.setLayout(new GridLayout(3, 3));
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (matrix[i][j] == 0) {
                    add(new Cell("H")); // Display H for cell 0
                }
                else {
                    add(new Cell("T")); // Display T for cell 1
                }
            }
        }
    }
}

/** An inner class for displaying a cell */
static class Cell extends JLabel {
    public Cell(String s) {
        this.setBorder(new LineBorder(Color.black, 1)); // Cell border
        this.setHorizontalAlignment(JLabel.CENTER);
        this.setFont(new Font("TimesRoman", Font.BOLD, 20));
        setText(s);
    }
}

/** An inner class for displaying the initial node */
static class InitialNodePanel extends JPanel {
    // Each cell represents a coin, which can be flipped
    ClickableCell[][] clickableCells = new ClickableCell[3][3];

    public InitialNodePanel() {
        this.setLayout(new GridLayout(3, 3));

        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {

```

```

        add(clickableCells[i][j] = new ClickableCell("H"));
    }
}

/** Get a 3 by 3 matrix from GUI cells */
public int[][] getMatrix() {
    int[][] matrix = new int[3][3];
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (clickableCells[i][j].getText().equals("H")) {
                matrix[i][j] = 0; // 0 is for head
            }
            else {
                matrix[i][j] = 1; // 0 is for tail
            }
        }
    }

    return matrix;
}

/** An inner class for displaying a clickable cell */
static class ClickableCell extends Cell {
    public ClickableCell(String s) {
        super(s);
        addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                if (getText().equals("H")) {
                    setText("T"); // Flip from H to T
                }
                else {
                    setText("H"); // Flip from T to H
                }
            }
        });
    }
}
}

```

Key Terms

*****PD:** Please place terms in two columns same as in *intro6e*.

- adjacency list
- adjacent vertices
- adjacency matrix
- breadth-first search
- complete graph
- degree
- depth-first search

- directed graph
- graph
- incident edges
- parallel edge
- Seven Bridges of Königsberg
- simple graph
- spanning tree
- weighted graph
- undirected graph
- unweighted graph

Chapter Summary

- A graph is a useful mathematical structure that represents relationships among entities in the real world. You learned how to model graphs using classes and interfaces, how to represent vertices and edges using arrays and linked lists, and how to implement operations for vertices and edges.
- Graph traversal is the process of visiting each vertex in the graph exactly once. You learned two popular ways for traversing a graph: depth-first search and breadth-first search.

Review Questions

Sections 13.1-13.3

13.1

What is the famous *Seven Bridges of Königsberg* problem?

13.2

What is a graph? Explain terms: undirected graph, directed graph, weighted graph, degree of a vertex, parallel edge, simple graph, and complete graph.

13.3

How do you represent vertices in a graph? How do you represent edges using an edge array? How do you represent an edge using an edge object? How do you represent edges using adjacency matrix? How do you represent edges using adjacency lists?

Sections 13.4-13.7

13.4

Describe the relationship among Graph, AbstractGraph, and UnweightedGraph.

13.5

Describe the relationship among Graph, AbstractGraph, and UnweightedGraph.

13.6

What is the return type from invoking dfs(v) and bfs(v)?

13.7

What is depth-first search and breadth-first search?

13.8

Show the DFS and BFS for the graph in Figure 13.1 starting from vertex Atlanta.

13.9

The depth-first search algorithm described in Listing 13.7 uses recursion. Alternatively you may use a stack to implement it, as shown below. Point the error in this algorithm and give a correct algorithm.

```
// Wrong version
dfs(vertex v) {
    push v into the stack;
    mark v visited;

    while (the stack is not empty) {
        pop a vertex, say u, from the stack
        visit u;
        for each neighbor w of u
            if (w has not been visited)
                push w into the stack;
    }
}
```

13.10

Prove that the path between the root and any node in the BFS tree is the shortest path between the root and the node.

Section 13.8

13.11

Lines 117-122 in Listing 13.11 copies array `matrix` to `matrixOfNextNode`. Will it work if you replace these lines using the following code?

```
int[][] matrixOfNextNode = (int[][])matrix.clone();
```

Programming Exercises

Sections 13.6-13.7

13.1*

(Implementing DFS using a stack) The depth-first search algorithm described in Listing 13.7 uses recursion. Implement it without using recursion.

13.2*

(Finding connected components) Add a new method in `AbstractGraph` to find all connected components in a graph with the following header:

```
public List getConnectedComponents();
```

The method returns a `List`. Each element in the list is another list that contains all the vertices in a connected component. For example, if the graph has three connected components, the method returns a list with three elements, each of which contains the vertices in a connected component.

13.3*

(Finding paths) Add a new method in `AbstractGraph` to find a path between two vertices with the following header:

```
public List getPath(int u, int v);
```

The method returns a `List` that contains all the vertices in a path from `u` to `v` in this order. Using the BFS approach, you can obtain a shortest path from `u` to `v`. If there is no path from `u` to `v`, the method returns `null`.

13.4*

(*Detecting cycles*) Add a new method in AbstractGraph to determine whether there is a cycle in the graph with the following header:

```
public boolean isCyclic();
```

13.5*

(*Finding a cycle*) Add a new method in AbstractGraph to find a cycle in the graph with the following header:

```
public List getACycle();
```

The method returns a List that contains all the vertices in a cycle from u to v in this order. If the graph has no cycles, the method returns null.

13.6**

(*Testing bipartite*) Recall that a graph is bipartite if the vertices of the graph can be divided into two disjoint sets such that no edges exist between vertices in the same set. Add a new method in AbstractGraph to detect whether the graph is bipartite:

```
public boolean isBipartite();
```

13.7**

(*Getting bipartite sets*) Add a new method in AbstractGraph to return two bipartite sets if the graph is bipartite:

```
public List isBipartite();
```

The method returns a List that contains two sublists, each of which contains a set of vertices. If the graph is not bipartite, the method returns null.

13.8**

(*Revising Listing 13.11, NineTailModel.java*) In Listing 13.11, a 3x3 array is used to store the cell values for a node. Revise the model to store cell values using integers 0, 1, 2, ..., and 511. Value 0 represents the node with all heads and 511 for the node with all tails, as shown below:

0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0		1	1	1
0	0	0	0	0	1	0	1	0	0	1	1		1	1	1
0			1			2			3				511		

This leads to a simple implementation for the model class. For example, line 50 in Listing 13.11

```
int[][] matrix = new int[3][3];
```

can be replaced by

```
int value;
```

The createNodes() (line 25 in Listing 13.11) method can be implemented using just one loop as follows:

```
for (int i = 0; i < 512; i++)
```

13.9**

(*Variation of the nine tail Problem*) In the nine tail problem, when you flip a head, the horizontal and vertical neighboring cells are also flipped. Rewrite the program assume that all neighboring cells including the diagonal neighbors are also flipped.

13.10**
(*4x4 sixteen tail model*) The nine tail problem in the text uses a 3x3 matrix. Assume that you have sixteen coins placed in a 4x4 matrix. Create a new model class named TailModel16. Create an instance of the model and save the object into a file named Exercise27_10.dat.

13.11**
(*4x4 sixteen tail view*) Listing 13.12, NineTailApp.java, presents a view for the nine tail problem. Revise this program for the 4x4 sixteen tail problem. Your program should read the model object created from the preceding exercise.

13.12**
(*Dynamic graphs*) Add the following methods in the Graph interface to dynamically add and remove vertices and edges:

```
// Return true if the vertex is added to the graph successfully  
public boolean add(Object vertex)  
  
// Return true if the vertex is removed from the graph successfully  
public boolean remove(Object vertex)  
  
// Return true if the edge is added to the graph successfully  
public boolean add(Edge edge)  
  
// Return true if the edge is removed from the graph successfully  
public boolean remove(Edge edge)
```

For simplicity, assume the vertices are labeled with integers 1, 2, ..., and so on.

13.13**
(*Induced subgraph*) Given an undirected graph $G = (V, E)$ and an integer k , find an induced subgraph H of G of maximum size such that all vertices of H have degree $\geq k$, or conclude that no such induced subgraph exists. Implement the method with the following header:

```
public static Graph maxInducedSubgraph(Graph edge, int k)
```

The method returns null if such subgraph does not exist.

Hint: An intuitive approach is to remove vertices whose degree is less than k . As vertices are removed with their adjacent edges, the degrees of other vertices may be reduced. Continue the process until no vertices can be removed, or all the vertices are removed.