CHAPTER

30

GRAPHS AND APPLICATIONS

Objectives

- To model real-world problems using graphs and explain the Seven Bridges of Königsberg problem (§30.1).
- To describe the graph terminologies: vertices, edges, simple graphs, weighted/unweighted graphs, and directed/undirected graphs (§30.2).
- To represent vertices and edges using lists, edge arrays, edge objects, adjacency matrices, and adjacency lists (§30.3).
- To model graphs using the `Graph` interface, the `AbstractGraph` class, and the `UnweightedGraph` class (§30.4).
- To display graphs visually (§30.5).
- To represent the traversal of a graph using the `AbstractGraph.Tree` class (§30.6).
- To design and implement depth-first search (§30.7).
- To solve the connected-circle problem using depth-first search (§30.8).
- To design and implement breadth-first search (§30.9).
- To solve the nine-tail problem using breadth-first search (§30.10).

30.1 Introduction

Key Point

Many real-world problems can be solved using graph algorithms.

problem

Graphs are useful in modeling and solving real-world problems. For example, the problem to find the least number of flights between two cities can be modeled using a graph, where the vertices represent cities and the edges represent the flights between two adjacent cities, as shown in Figure 30.1. The problem of finding the minimal number of connecting flights between two cities is reduced to finding the shortest path between two vertices in a graph.

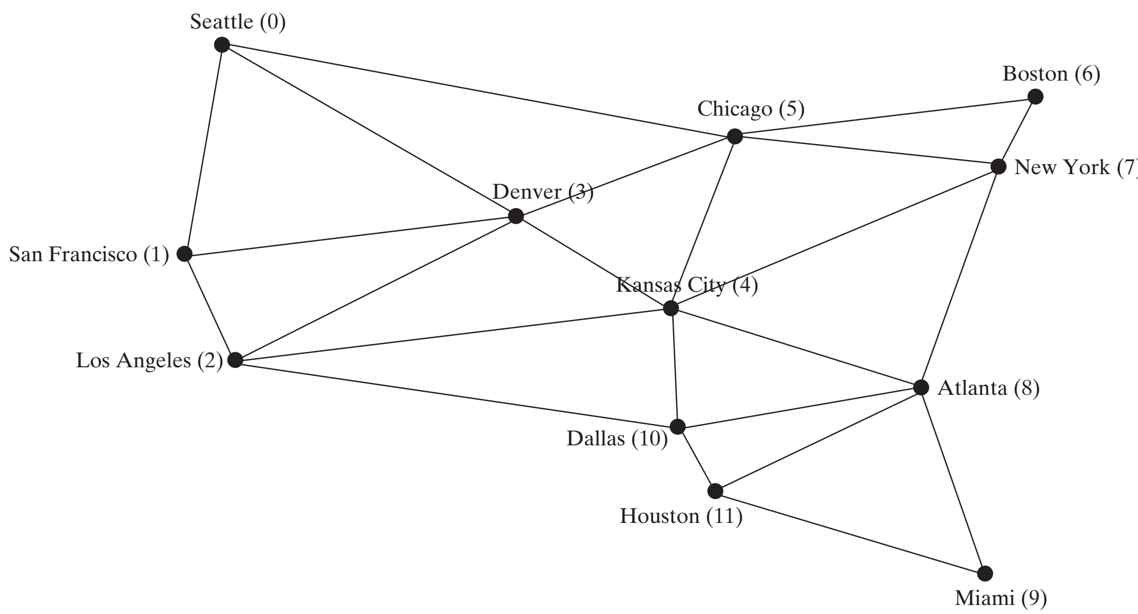


FIGURE 30.1 A graph can be used to model the flights between the cities.

graph theory
Seven Bridges of Königsberg

The study of graph problems is known as *graph theory*. Graph theory was founded by Leonhard Euler in 1736, when he introduced graph terminology to solve the famous *Seven Bridges of Königsberg* problem. 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 30.2a. The question is, can one take a walk, cross each bridge exactly once, and return to the starting point? Euler proved that it is not possible.

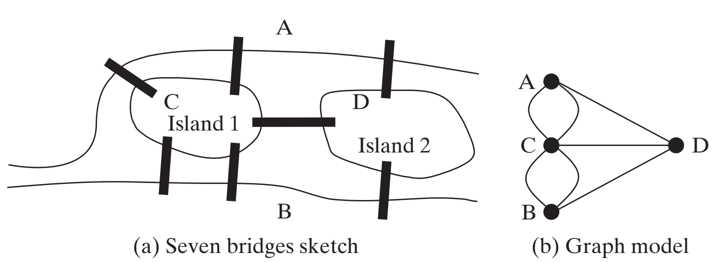


FIGURE 30.2 Seven bridges connected islands and land.

To establish a proof, Euler first abstracted the Königsberg city map by eliminating all streets, producing the sketch shown in Figure 30.2a. Next, 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 30.2b. This structure with vertices and edges is called a *graph*.

Looking at the graph, we ask 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 a path to exist, each vertex must have an even number of edges. Therefore, the Seven Bridges of Königsberg problem has no solution.

Graph problems are often solved using algorithms. Graph algorithms have many applications in various areas, such as in 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 a minimum spanning tree and shortest paths in weighted graphs, and their applications.

30.2 Basic Graph Terminologies

A graph consists of vertices, and edges that connect the vertices.

Key Point

This chapter does not assume that you have any prior knowledge of graph theory or discrete mathematics. We use plain and simple terms to define graphs.

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 30.1 represents the flights among cities, and the graph in Figure 30.2b represents the bridges among land masses.

define a graph

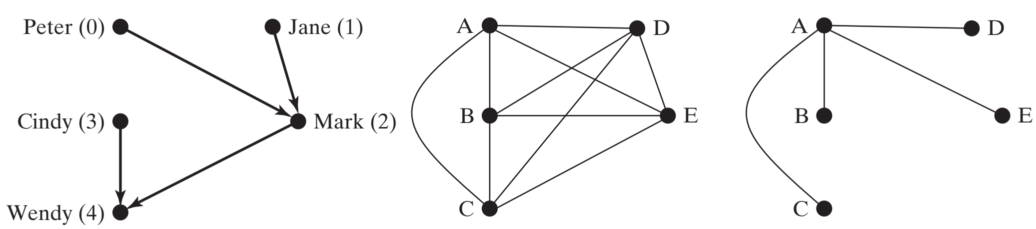
A graph consists of a nonempty set of vertices (also known as *nodes* or *points*), and a set of edges that connect the vertices. For convenience, we define a graph as G = (V, E), where V represents a set of vertices and E represents a set of edges. For example, V and E for the graph in Figure 30.1 are 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"},
       ...
    };
```

directed vs. undirected graphs

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 can 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 30.3a shows a directed graph.



(a) A directed graph (b) A complete graph (c) A subgraph of the graph in (b)

FIGURE 30.3 Graphs may appear in many forms.

In an *undirected graph*, you can move in both directions between vertices. The graph in Figure 30.1 is undirected.

weighted vs. unweighted graphs

Edges may be weighted or unweighted. For example, you can assign a weight for each edge in the graph in Figure 30.1 to indicate the flight time between the two cities.

adjacent vertices
incident edges
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 are connected 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.

neighbor

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

loop
parallel edge
simple graph
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 doesn't have any loops or parallel edges. In a *complete graph*, every two pairs of vertices are connected, as shown in Figure 30.3b.

connected graph
subgraph

A graph is *connected* if there exists a path between any two vertices in the graph. A *subgraph* of a graph *G* is a graph whose vertex set is a subset of that of *G* and whose edge set is a subset of that of *G*. For example, the graph in Figure 30.3c is a subgraph of the graph in Figure 30.3b.

tree
cycle
spanning tree

Assume that the graph is connected and undirected. A connected graph is a *tree* if it does not have cycles. A *cycle* is a closed path that starts from a vertex and ends at the same vertex. A *spanning tree* of a graph *G* is a connected subgraph of *G* and the subgraph is a tree that contains all vertices in G.

graph learning tool on Companion Website

Pedagogical Note

Before we introduce graph algorithms and applications, it is helpful to get acquainted with graphs using the interactive tool at www.cs.armstrong.edu/liang/animation/GraphLearningTool.html, as shown in Figure 30.4. The tool allows you to add/remove/move vertices and draw edges using mouse gestures. You can also find depth-first search (DFS) trees and breadth-first search (BFS) trees, and the shortest path between two vertices.

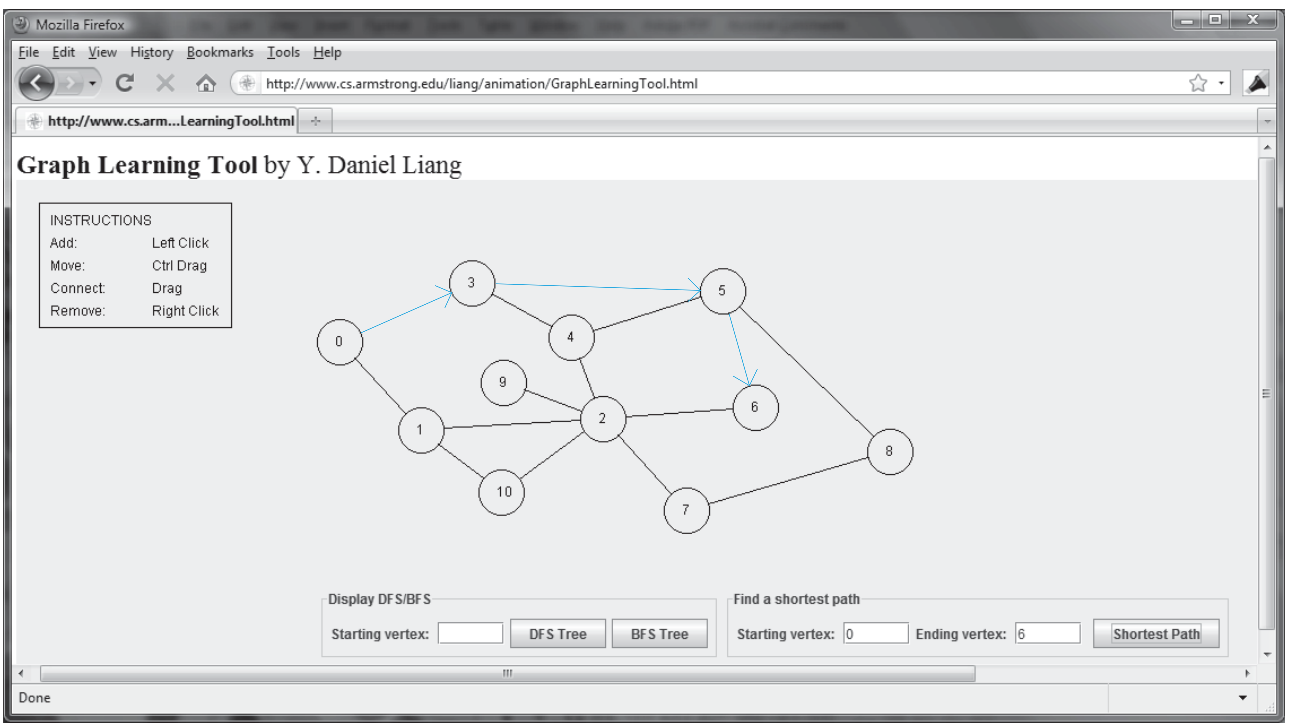


FIGURE 30.4 You can use the tool to create a graph with mouse gestures and show DFS/BFS trees and shortest paths.

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

30.2 What is a graph? Explain the following terms: undirected graph, directed graph, weighted graph, degree of a vertex, parallel edge, simple graph, complete graph, connected graph, cycle, subgraph, tree, and spanning tree.

30.3 How many edges are in a complete graph with 5 vertices? How many edges are in a tree of 5 vertices?

30.4 How many edges are in a complete graph with *n* vertices? How many edges are in a tree of *n* vertices?

Check Point

MyProgrammingLab™

30.3 Representing Graphs

Key Point

Representing a graph is to store its vertices and edges in a program. The data structure for storing a graph is arrays or lists.

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

30.3.1 Representing Vertices

The vertices can be stored in an array or a list. For example, you can store all the city names in the graph in Figure 30.1 using the following array:

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

vertex type

Note

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

```
City city0 = new City("Seattle", 608660, "Mike McGinn");
...
City city11 = new City("Houston", 2099451, "Annise Parker");
City[] 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 getMayor() {
      return mayor;
    }

    public void setMayor(String mayor) {
      this.mayor = mayor;
    }

    public void setPopulation(int population) {
      this.population = population;
    }
  }
```

The vertices can be conveniently labeled using natural numbers 0, 1, 2, . . . , $n - 1$, for a graph for n vertices. Thus, `vertices[0]` represents `"Seattle"`, `vertices[1]` represents `"San Francisco"`, and so on, as shown in Figure 30.5.

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 30.5 An array stores the vertex names.

reference vertex

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

30.3.2 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 30.1 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, 1}, {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}
  };
```

This representation is known as the *edge array*. The vertices and edges in Figure 30.3a can be represented as follows:

edge array

```
String[] names = {"Peter", "Jane", "Mark", "Cindy", "Wendy"};

int[][] edges = {{0, 2}, {1, 2}, {2, 4}, {3, 4}};
```

30.3.3 Representing Edges: **Edge** Objects

Another way to represent the edges is to define edges as objects and store 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 30.1 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.

While representing edges using an edge array or **Edge** objects in Section 30.3.2 and earlier in this section may be intuitive for input, it's not efficient for internal processing. The next two sections introduce the representation of graphs using *adjacency matrices* and *adjacency lists*. These two data structures are efficient for processing graphs.

30.3.4 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 the 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 adjacency matrix

example, the edges in the graph in Figure 30.1 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, 0}, // San Francisco
  {0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0}, // Los Angeles
  {1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0}, // Denver
  {0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0}, // Kansas City
  {1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0}, // Chicago
  {0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0}, // Boston
  {0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0}, // New York
  {0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1}, // Atlanta
  {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1}, // Miami
  {0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1}, // Dallas
  {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0}  // Houston
};
```

ragged array

Note

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

The adjacency matrix for the directed graph in Figure 30.3a 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
            };
```

30.3.5 Representing Edges: Adjacency Lists

adjacency lists

To represent edges using *adjacency lists*, define an array of lists. The array has *n* entries, and each entry is a linked list. 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 30.1, you can create an array of linked lists as follows:

```
java.util.LinkedList[] neighbors = new java.util.LinkedList[12];
```

`neighbors[0]` contains all vertices adjacent to vertex `0` (i.e., Seattle), `neighbors[1]` contains all vertices adjacent to vertex `1` (i.e., San Francisco), and so on, as shown in Figure 30.6.

To represent the edges in the graph in Figure 30.3a, you can create an array of linked lists as follows:

```
java.util.LinkedList[] neighbors = new java.util.LinkedList[5];
```

`neighbors[0]` contains all vertices pointed from vertex `0` via directed edges, `neighbors[1]` contains all vertices pointed from vertex `1` via directed edges, and so on, as shown in Figure 30.7. Wendy does not point to any vertex, so `neighbors[4]` is `null`.

adjacency matrices vs. adjacency lists

Note

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 using an adjacency matrix would waste a lot of space.

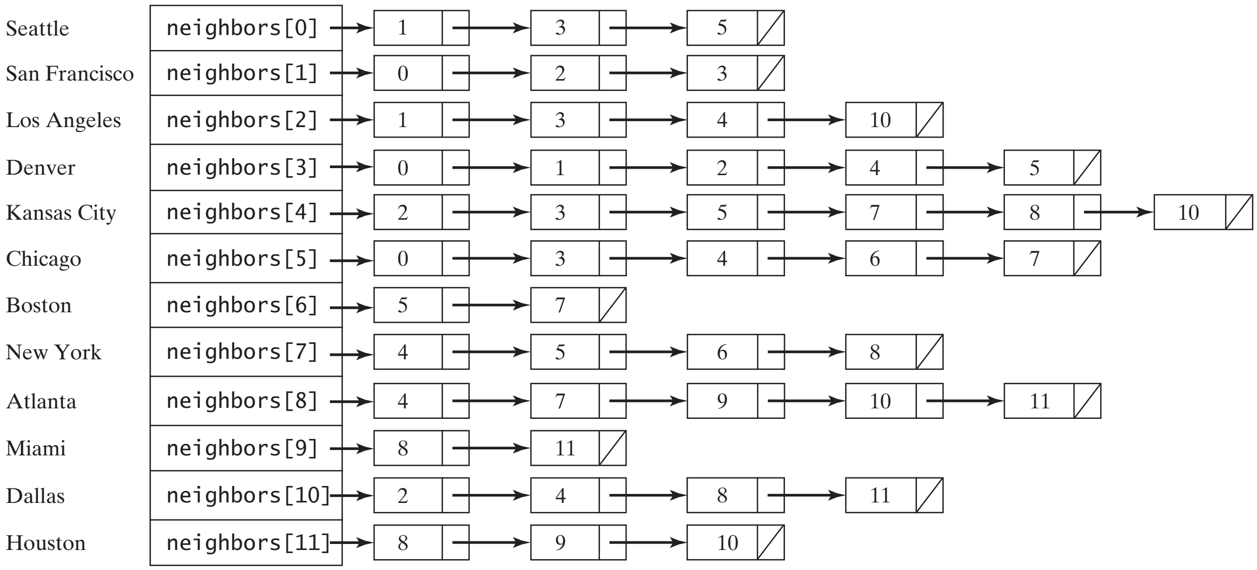


FIGURE 30.6 Edges in the graph in Figure 30.1 are represented using linked lists.

FIGURE 30.7 Edges in the graph in Figure 30.3a are represented using linked lists.

Both adjacency matrices and adjacency lists can be used in a program to make algorithms more efficient. For example, it takes $O(1)$ constant time to check whether 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.

Note

other representations

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

using `ArrayList`

For flexibility and simplicity, we will use array lists to represent arrays. Also, we will use array lists instead of linked lists, because our algorithms only require searching for adjacent vertices in the list. Using array lists is more efficient for our algorithms. Using array lists, the adjacency list in Figure 30.6 can be built as follows:

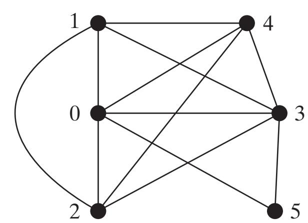
```
List<ArrayList<Integer>> neighbors
  = new ArrayList<List<Integer>>();
neighbors.add(new ArrayList<Integer>());
```

```
neighbors.get(0).add(1); neighbors.get(0).add(3);
  neighbors.get(0).add(5);
neighbors.add(new ArrayList<Integer>());
neighbors.get(1).add(0); neighbors.get(1).add(2);
  neighbors.get(1).add(3);
...
...
```

Check Point
MyProgrammingLab™

30.5 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 an adjacency matrix? How do you represent edges using adjacency lists?

30.6 Represent the following graph using an edge array, a list of edge objects, an adjacency matrix, and an adjacency list, respectively.



30.4 Modeling Graphs

Key Point

The **`Graph`** *interface defines the common operations for a graph.*

The Java Collections Framework serves as a good example for designing complex data structures. The common features of data structures are defined in the interfaces (e.g., **`Collection`**, **`Set`**, **`List`**, **`Queue`**), as shown in Figure 22.1. Abstract classes (e.g., **`AbstractCollection`**, **`AbstractSet`**, **`AbstractList`**) partially implement the interfaces. Concrete classes (e.g., **`HashSet`**, **`LinkedHashSet`**, **`TreeSet`**, **`ArrayList`**, **`LinkedList`**, **`PriorityQueue`**) provide concrete implementations. This design pattern is useful for modeling graphs. We will define an interface named **`Graph`** that contains all the common operations of graphs and an abstract class named **`AbstractGraph`** that partially implements the **`Graph`** interface. Many concrete graphs can 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 30.8.

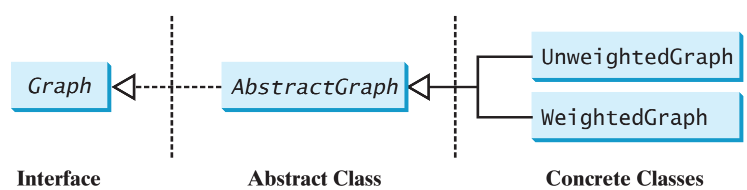


FIGURE 30.8 Graphs can be modeled using interfaces, abstract classes, and concrete classes.

What are the 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 degree for a vertex, clear the graph, add a new vertex, add a new edge, perform a depth-first search, and perform a breadth-first search. Depth-first search and breadth-first search will be introduced in the next section. Figure 30.9 illustrates these methods in the UML diagram.

`AbstractGraph` does not introduce any new methods. A list of vertices and a list 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.

FIGURE 30.9 The `Graph` interface defines the common operations for all types of graphs.

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

vertices and their indices

Note

You can create a graph with any type of vertices. Each vertex is associated with an index, which is the same as the index of the vertex in the vertices list. If you create a graph without specifying the vertices, the vertices are the same as their indices.

why AbstractGraph?

Note

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

Assume all these interfaces and classes are available. Listing 30.1 gives a test program that creates the graph in Figure 30.1 and another graph for the one in Figure 30.3a.

LISTING 30.1 TestGraph.java

vertices

edges

create a graph

number of vertices

get vertex

get index

print edges

list of Edge objects

```java
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"};

    // Edge array for graph in Figure 30.1
    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, 1}, {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<String> graph1 =
      new UnweightedGraph<String>(edges, vertices);
    System.out.println("The number of vertices in graph1: "
      + graph1.getSize());
    System.out.println("The vertex with index 1 is "
      + graph1.getVertex(1));
    System.out.println("The index for Miami is " +
      graph1.getIndex("Miami"));
    System.out.println("The edges for graph1:");
    graph1.printEdges();

    // List of Edge objects for graph in Figure 30.3a
    String[] names = {"Peter", "Jane", "Mark", "Cindy", "Wendy"};
    java.util.ArrayList<AbstractGraph.Edge> edgeList
      = new java.util.ArrayList<AbstractGraph.Edge>();
    edgeList.add(new AbstractGraph.Edge(0, 2));
```

```
39      edgeList.add(new AbstractGraph.Edge(1, 2));
40      edgeList.add(new AbstractGraph.Edge(2, 4));
41      edgeList.add(new AbstractGraph.Edge(3, 4));
42      // Create a graph with 5 vertices
43      Graph<String> graph2 = new UnweightedGraph<String>
44        (edgeList, java.util.Arrays.asList(names));
45      System.out.println("\nThe number of vertices in graph2: "
46        + graph2.getSize());
47      System.out.println("The edges for graph2:");
48      graph2.printEdges();
49    }
50  }
```

create a graph

print edges

```
The number of vertices in graph1: 12
The vertex with index 1 is San Francisco
The index for Miami is 9
The edges for graph1:
Seattle (0): (0, 1) (0, 3) (0, 5)
San Francisco (1): (1, 0) (1, 2) (1, 3)
Los Angeles (2): (2, 1) (2, 3) (2, 4) (2, 10)
Denver (3): (3, 0) (3, 1) (3, 2) (3, 4) (3, 5)
Kansas City (4): (4, 2) (4, 3) (4, 5) (4, 7) (4, 8) (4, 10)
Chicago (5): (5, 0) (5, 3) (5, 4) (5, 6) (5, 7)
Boston (6): (6, 5) (6, 7)
New York (7): (7, 4) (7, 5) (7, 6) (7, 8)
Atlanta (8): (8, 4) (8, 7) (8, 9) (8, 10) (8, 11)
Miami (9): (9, 8) (9, 11)
Dallas (10): (10, 2) (10, 4) (10, 8) (10, 11)
Houston (11): (11, 8) (11, 9) (11, 10)

The number of vertices in graph2: 5
The edges for graph2:
Peter (0): (0, 2)
Jane (1): (1, 2)
Mark (2): (2, 4)
Cindy (3): (3, 4)
Wendy (4):
```

The program creates **graph1** for the graph in Figure 30.1 in lines 3–24. The vertices for **graph1** are defined in lines 3–5. The edges for **graph1** are defined in 8–21. 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[2][0]**) to vertex **5** (**edges[2][1]**). The graph is created in line 24. Line 32 invokes the **printEdges()** method on **graph1** to display all edges in **graph1**.

The program creates **graph2** for the graph in Figure 30.3a in lines 35–44. The edges for **graph2** are defined in lines 38–41. **graph2** is created using a list of **Edge** objects in line 44. Line 48 invokes the **printEdges()** method on **graph2** to display all edges in **graph2**.

Note that both **graph1** and **graph2** contain the vertices of strings. The vertices are associated with indices **0**, **1**, . . . , **n-1**. The index is the location of the vertex in **vertices**. For example, the index of vertex **Miami** is **9**.

Now we turn our attention to implementing the interface and classes. Listings 30.2, 30.3, and 30.4 give the **Graph** interface, the **AbstractGraph** class, and the **UnweightedGraph** class, respectively.

Listing 30.2 Graph.java

getSize
getVertices
getVertex
getIndex
getNeighbors
getDegree
printEdges
clear
addVertex
addEdge
dfs
bfs

```java
public interface Graph<V> {
  /** Return the number of vertices in the graph */
  public int getSize();

  /** Return the vertices in the graph */
  public java.util.List<V> getVertices();

  /** Return the object for the specified vertex index */
  public V getVertex(int index);

  /** Return the index for the specified vertex object */
  public int getIndex(V v);

  /** Return the neighbors of vertex with the specified index */
  public java.util.List<Integer> getNeighbors(int index);

  /** Return the degree for a specified vertex */
  public int getDegree(int v);

  /** Print the edges */
  public void printEdges();

  /** Clear graph */
  public void clear();

  /** Add a vertex to the graph */
  public void addVertex(V vertex);

  /** Add an edge to the graph */
  public void addEdge(int u, int v);

  /** Obtain a depth-first search tree starting from v */
  public AbstractGraph<V>.Tree dfs(int v);

  /** Obtain a breadth-first search tree starting from v */
  public AbstractGraph<V>.Tree bfs(int v);
}
```

Listing 30.3 AbstractGraph.java

no-arg constructor
constructor

```java
import java.util.*;

public abstract class AbstractGraph<V> implements Graph<V> {
  protected List<V> vertices = new ArrayList<V>(); // Store vertices
  protected List<List<Integer>> neighbors
    = new ArrayList<List<Integer>>(); // Adjacency lists

  /** Construct an empty graph */
  protected AbstractGraph() {
  }

  /** Construct a graph from edges and vertices stored in arrays */
  protected AbstractGraph(int[][] edges, V[] vertices) {
    for (int i = 0; i < vertices.length; i++)
      this.vertices.add(vertices[i]);

    createAdjacencyLists(edges, vertices.length);
  }
```

```

  /** Construct a graph from edges and vertices stored in List */
  protected AbstractGraph(List<Edge> edges, List<V> vertices) {      constructor
    for (int i = 0; i < vertices.size(); i++)
      this.vertices.add(vertices.get(i));

    createAdjacencyLists(edges, vertices.size());
  }

  /** Construct a graph for integer vertices 0, 1, 2 and edge list */
  protected AbstractGraph(List<Edge> edges, int numberOfVertices) {  constructor
    for (int i = 0; i < numberOfVertices; i++)
      vertices.add((V)(new Integer(i))); // vertices is {0, 1, ...}

    createAdjacencyLists(edges, numberOfVertices);
  }

  /** Construct a graph from integer vertices 0, 1, and edge array */
  protected AbstractGraph(int[][] edges, int numberOfVertices) {     constructor
    for (int i = 0; i < numberOfVertices; i++)
      vertices.add((V)(new Integer(i))); // vertices is {0, 1, ...}

    createAdjacencyLists(edges, numberOfVertices);
  }

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

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

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

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

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

  @Override /** Return the vertices in the graph */
  public List<V> getVertices() {                                     getVertices
```

```
    return vertices;
  }

  @Override /** Return the object for the specified vertex */
```

getVertex

```
  public V getVertex(int index) {
    return vertices.get(index);
  }

  @Override /** Return the index for the specified vertex object */
```

getIndex

```
  public int getIndex(V v) {
    return vertices.indexOf(v);
  }

  @Override /** Return the neighbors of the specified vertex */
```

getNeighbors

```
  public List<Integer> getNeighbors(int index) {
    return neighbors.get(index);
  }

  @Override /** Return the degree for a specified vertex */
```

getDegree

```
  public int getDegree(int v) {
    return neighbors.get(v).size();
  }

  @Override /** Print the edges */
```

printEdges

```
  public void printEdges() {
    for (int u = 0; u < neighbors.size(); u++) {
      System.out.print(getVertex(u) + " (" + u + "): ");
      for (int j = 0; j < neighbors.get(u).size(); j++) {
        System.out.print("(" + u + ", " +
          neighbors.get(u).get(j) + ") ");
      }
      System.out.println();
    }
  }

  @Override /** Clear graph */
```

clear

```
  public void clear() {
    vertices.clear();
    neighbors.clear();
  }

  @Override /** Add a vertex to the graph */
```

addVertex

```
  public void addVertex(V vertex) {
    vertices.add(vertex);
    neighbors.add(new ArrayList<Integer>());
  }

  @Override /** Add an edge to the graph */
```

addEdge

```
  public void addEdge(int u, int v) {
    neighbors.get(u).add(v);
    neighbors.get(v).add(u);
  }

  /** Edge inner class inside the AbstractGraph class */
```

Edge inner 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) {
```

```java
      this.u = u;
      this.v = v;
    }
  }

  @Override /** Obtain a DFS tree starting from vertex v */
  /** To be discussed in Section 30.7 */
```

dfs method

```java
  public Tree dfs(int v) {
    List<Integer> searchOrder = new ArrayList<Integer>();
    int[] parent = new int[vertices.size()];
    for (int i = 0; i < parent.length; i++)
      parent[i] = -1; // Initialize parent[i] to -1

    // Mark visited vertices
    boolean[] isVisited = new boolean[vertices.size()];

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

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

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

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

  @Override /** Starting BFS search from vertex v */
  /** To be discussed in Section 30.9 */
```

bfs method

```java
  public Tree bfs(int v) {
    List<Integer> searchOrder = new ArrayList<Integer>();
    int[] parent = new int[vertices.size()];
    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.size()];
    queue.offer(v); // Enqueue v
    isVisited[v] = true; // Mark it visited

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

Tree inner class

```
        }
      }
    }

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

  /** Tree inner class inside the AbstractGraph class */
  /** To be discussed in Section 30.5 */
  public class Tree {
    private int root; // The root of the tree
    private int[] parent; // Store the parent of each vertex
    private List<Integer> searchOrder; // Store the search order

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

    /** 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> getSearchOrder() {
      return searchOrder;
    }

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

    /** Return the path of vertices from a vertex to the root */
    public List<V> getPath(int index) {
      ArrayList<V> path = new ArrayList<V>();

      do {
        path.add(vertices.get(index));
        index = parent[index];
      }
      while (index != -1);

      return path;
    }

    /** Print a path from the root to vertex v */
    public void printPath(int index) {
      List<V> path = getPath(index);
      System.out.print("A path from " + vertices.get(root) + " to " +
        vertices.get(index) + ": ");
      for (int i = path.size() - 1; i >= 0; i--)
```

```
259         System.out.print(path.get(i) + " ");
260       }
261 
262       /** Print the whole tree */
263       public void printTree() {
264         System.out.println("Root is: " + vertices.get(root));
265         System.out.print("Edges: ");
266         for (int i = 0; i < parent.length; i++) {
267           if (parent[i] != -1) {
268             // Display an edge
269             System.out.print("(" + vertices.get(parent[i]) + ", " +
270               vertices.get(i) + ") ");
271           }
272         }
273         System.out.println();
274       }
275     }
276   }
```

LISTING 30.4 UnweightedGraph.java

```
import java.util.*;

public class UnweightedGraph<V> extends AbstractGraph<V> {
  /** Construct an empty graph */
  public UnweightedGraph() {
  }

  /** Construct a graph from edges and vertices stored in arrays */
  public UnweightedGraph(int[][] edges, V[] vertices) {
    super(edges, vertices);
  }

  /** Construct a graph from edges and vertices stored in List */
  public UnweightedGraph(List<Edge> edges, List<V> vertices) {
    super(edges, vertices);
  }

  /** Construct a graph for integer vertices 0, 1, 2 and edge list */
  public UnweightedGraph(List<Edge> edges, int numberOfVertices) {
    super(edges, numberOfVertices);
  }

  /** Construct a graph from integer vertices 0, 1, and edge array */
  public UnweightedGraph(int[][] edges, int numberOfVertices) {
    super(edges, numberOfVertices);
  }
}
```

no-arg constructor

constructor

constructor

constructor

constructor

The code in the `Graph` interface in Listing 30.2 and the `UnweightedGraph` class in Listing 30.4 are straightforward. Let us digest the code in the `AbstractGraph` class in Listing 30.3.

The `AbstractGraph` class defines the data field `vertices` (line 4) to store vertices and `neighbors` (line 5) to store edges in adjacency lists. `neighbors.get(i)` stores all vertices adjacent to vertex `i`. Four overloaded constructors are defined in lines 9–42 to create a default graph, or a graph from arrays or lists of edges and vertices. The `createAdjacencyLists(int[][] edges, int numberOfVertices)` method creates adjacency lists from edges in an array (lines 45–57). The `createAdjacencyLists(List<Edge> edges, int numberOfVertices)` method creates adjacency lists from edges in a list (lines 60–70).

The **printEdges()** method (lines 103–112) displays all vertices and edges adjacent to each vertex.

The code in lines 146–275 gives the methods for finding a depth-first search tree and a breadth-first search tree, which will be introduced in Sections 30.7 and 30.9, respectively.

Check Point

MyProgrammingLab

30.7 Describe the relationships among **Graph**, **AbstractGraph**, and **UnweightedGraph**.

30.8 For the code in Listing 30.1, TestGraph.java, what is **graph1.getIndex("Seattle")**? What is **graph1.getDegree(5)**? What is **graph1.getVertex(4)**?

30.5 Graph Visualization

Key Point

To display a graph visually, each vertex must be assigned a location.

The preceding section introduced how to model a graph using the **Graph** interface, **AbstractGraph** class, and **UnweightedGraph** class. This section discusses how to display graphs graphically. In order to display a graph, you need to know where each vertex is displayed and the name of each vertex. To ensure a graph can be displayed, we define an interface named **Displayable** that has the methods for obtaining the **x**-and **y**-coordinates and their names, and make vertices instances of **Displayable**, in Listing 30.5.

LISTING 30.5 Displayable.java

Displayable interface

```
public interface Displayable {
  public int getX(); // Get x-coordinate of the vertex
  public int getY(); // Get y-coordinate of the vertex
  public String getName(); // Get display name of the vertex
}
```

A graph with **Displayable** vertices can now be displayed on a panel named **GraphView**, as shown in Listing 30.6.

LISTING 30.6 GraphView.java

extends JPanel

```
public class GraphView extends javax.swing.JPanel {
  private Graph<? extends Displayable> graph;

  public GraphView(Graph<? extends Displayable> graph) {
    this.graph = graph;
  }

  @Override
  protected void paintComponent(java.awt.Graphics g) {
    super.paintComponent(g);

    // Draw vertices
    java.util.List<? extends Displayable> vertices
      = graph.getVertices();
    for (int i = 0; i < graph.getSize(); i++) {
      int x = vertices.get(i).getX();
      int y = vertices.get(i).getY();
      String name = vertices.get(i).getName();

      g.fillOval(x - 8, y - 8, 16, 16); // Display a vertex
      g.drawString(name, x - 12, y - 12); // Display the name
    }

    // Draw edges for pair of vertices
    for (int i = 0; i < graph.getSize(); i++) {
```

```
26        java.util.List<Integer> neighbors = graph.getNeighbors(i);
27        int x1 = graph.getVertex(i).getX();
28        int y1 = graph.getVertex(i).getY();
29        for (int v: neighbors) {
30          int x2 = graph.getVertex(v).getX();
31          int y2 = graph.getVertex(v).getY();
32
33          g.drawLine(x1, y1, x2, y2); // Draw an edge for (i, v)
34        }
35      }
36    }
37  }
```

To display a graph on a panel, simply create an instance of **GraphView** by passing the graph as an argument in the constructor (line 4). The class for the graph's vertex must implement the **Displayable** interface in order to display the vertices (lines 13–22). For each vertex index **i**, invoking **graph.getNeighbors(i)** returns its adjacency list (line 26). From this list, you can find all vertices that are adjacent to **i** and draw a line to connect **i** with its adjacent vertex (lines 27–34).

Listing 30.7 gives an example of displaying the graph in Figure 30.1, as shown in Figure 30.10.

LISTING 30.7 DisplayUSMap.java

```
import javax.swing.*;

public class DisplayUSMap extends JApplet {
  private City[] vertices = {new City("Seattle", 75, 50),
    new City("San Francisco", 50, 210),
    new City("Los Angeles", 75, 275), new City("Denver", 275, 175),
    new City("Kansas City", 400, 245),
    new City("Chicago", 450, 100), new City("Boston", 700, 80),
    new City("New York", 675, 120), new City("Atlanta", 575, 295),
    new City("Miami", 600, 400), new City("Dallas", 408, 325),
    new City("Houston", 450, 360) };

  // Edge array for graph in Figure 30.1
  private 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, 1}, {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}
  };

  private Graph<City> graph
    = new UnweightedGraph<City>(edges, vertices);

  public DisplayUSMap() {
    add(new GraphView(graph));
  }

  static class City implements Displayable {
    private int x, y;
    private String name;
```

create a graph

create a GraphView

City class

```
36
37       City(String name, int x, int y) {
38         this.name = name;
39         this.x = x;
40         this.y = y;
41       }
42
43       @Override
44       public int getX() {
45         return x;
46       }
47
48       @Override
49       public int getY() {
50         return y;
51       }
52
53       @Override
54       public String getName() {
55         return name;
56       }
57     }
58   }
```

main method omitted

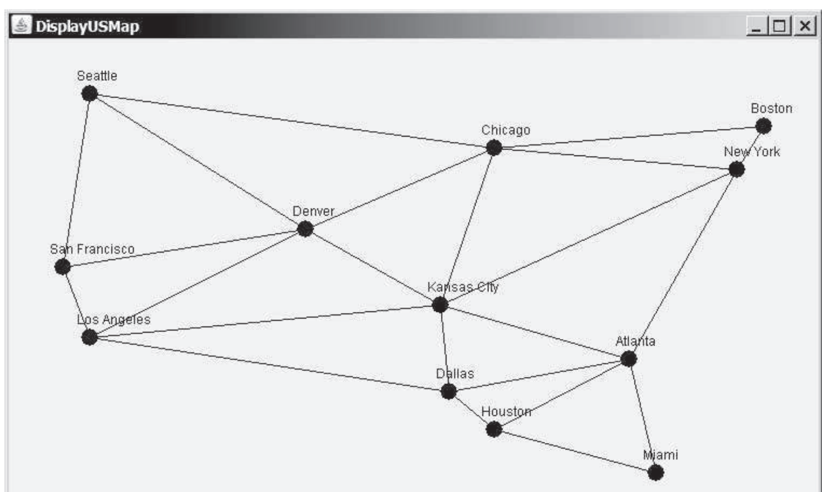


FIGURE 30.10 The graph is displayed in the panel.

The class **City** is defined to model the vertices with their coordinates and names (lines 33–57). The program creates a graph with the vertices of the **City** type (line 27). Since **City** implements **Displayable**, a **GraphView** object created for the graph displays the graph in the panel (line 30).

As an exercise to get acquainted with the graph classes and interfaces, add a city (e.g., Savannah) with appropriate edges into the graph.

Check Point

MyProgrammingLab

30.9 For the **graph1** object created in Listing 30.1, TestGraph.java, can you create a **GraphView** object as follows?

```
GraphView view = new GraphView(graph1);
```

30.6 Graph Traversals

Key Point

Depth-first and breadth-first are two common ways to traverse a graph.

depth-first search
breadth-first search

Graph traversal is the process of visiting each vertex in the graph exactly once. There are two popular ways to traverse a graph: *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 30.11. Note that **Tree** is an inner class defined in the **AbstractGraph** class. **AbstractGraph<V>.Tree** is different from the **Tree** interface defined in Section 27.2.5. **AbstractGraph.Tree** is a specialized class designed for describing the parent-child relationship of the nodes, whereas the **Tree** interface defines common operations such as searching, inserting, and deleting in a tree. Since there is no need to perform these operations for a spanning tree, **AbstractGraph<V>.Tree** is not defined as a subtype of **Tree**.

AbstractGraph<V>.Tree	
-root: int	The root of the tree.
-parent: int[]	The parents of the vertices.
-searchOrder: List<Integer>	The orders for traversing the vertices.
+Tree(root: int, parent: int[], searchOrder: List<Integer>)	Constructs a tree with the specified root, parent, and searchOrder.
+getRoot(): int	Returns the root of the tree.
+getSearchOrder(): List<Integer>	Returns the order of vertices searched.
+getParent(index: int): int	Returns the parent for the specified vertex index.
+getNumberOfVerticesFound(): int	Returns the number of vertices searched.
+getPath(index: int): List<V>	Returns a list of vertices from the specified vertex index to the root.
+printPath(index: int): void	Displays a path from the root to the specified vertex.
+printTree(): void	Displays tree with the root and all edges.

FIGURE 30.11 The **Tree** class describes the nodes with parent-child relationships.

The **Tree** class is defined as an inner class in the **AbstractGraph** class in lines 208–275 in Listing 30.3. The constructor creates a tree with the root, edges, and a search order.

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 **getSearchOrder()** method. You can invoke **getParent(v)** to find the parent of vertex **v** in the search. Invoking **getNumberOfVerticesFound()** returns the number of vertices searched. The method **getPath(index)** returns a list of vertices from the specified vertex index to the root. Invoking **printPath(v)** displays a path from the root to **v**. You can display all edges in the tree using the **printTree()** method.

Sections 30.7 and 30.9 will introduce depth-first search and breadth-first search, respectively. Both searches will result in an instance of the **Tree** class.

Check Point
MyProgrammingLab™

30.10 Does **AbstractGraph<V>.Tree** implement the **Tree** interface defined in Listing 27.3 Tree.java?

30.11 What method do you use to find the parent of a vertex in the tree?

30.7 Depth-First Search (DFS)

Key Point

The depth-first search of a graph starts from a vertex in the graph and visits all vertices in the graph as far as possible before backtracking.

The depth-first search of a graph is like the depth-first search of a tree discussed in Section 27.2.4, 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 it recursively visits all the vertices adjacent to that vertex. The difference is that the graph may contain cycles, which could 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*, it next visits an unvisited neighbor of *v*. If *v* has no unvisited neighbor, the search backtracks to the vertex from which it reached *v*. We assume that the graph is connected and the search starting from any vertex can reach all the vertices. If this is not the case, see Programming Exercise 30.4 for finding connected components in a graph.

30.7.1 Depth-First Search Algorithm

The algorithm for the depth-first search is described in Listing 30.8.

Listing 30.8 Depth-First Search Algorithm

visit v

check a neighbor

recursive search

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

You can 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 30.12a. 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 30.12b. 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 30.12c. Vertex 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 30.12d. At this point, the vertices have been visited in this order:

```
0, 1, 2, 3
```

Since all the neighbors of 3 have been visited, backtrack to 2. Since all the vertices of 2 have been visited, backtrack to 1. 4 is adjacent to 1, but 4 has not been visited. Therefore, visit `4`, as shown in Figure 30.12e. Since all the neighbors of 4 have been visited, backtrack to 1. Since all the neighbors of 1 have been visited, backtrack to 0. Since all the neighbors of 0 have been visited, the search ends.

DFS time complexity

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

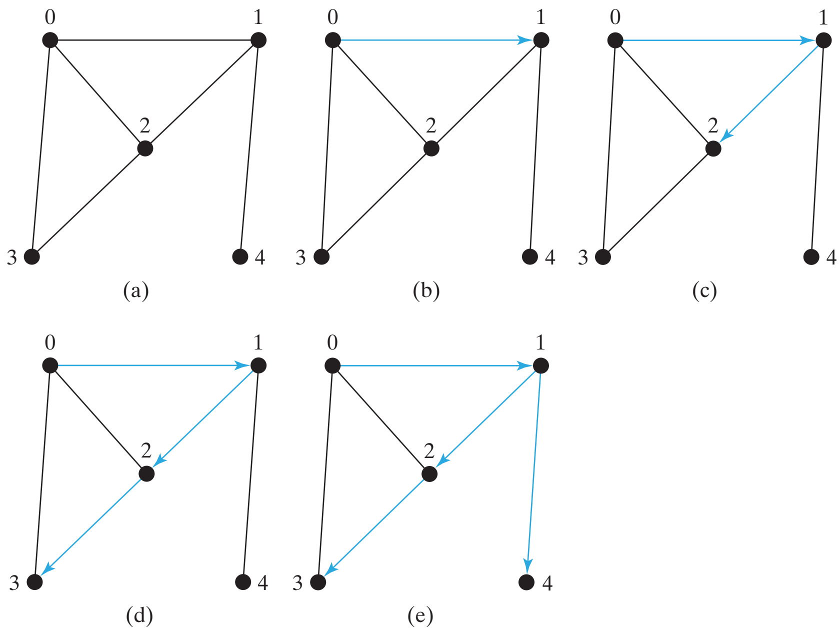


FIGURE 30.12 Depth-first search visits a node and its neighbors recursively.

30.7.2 Implementation of Depth-First Search

The algorithm for DFS in Listing 30.8 uses recursion. It is natural to use recursion to implement it. Alternatively, you can use a stack (see Programming Exercise 30.3).

The `dfs(int v)` method is implemented in lines 146–175 in Listing 30.3. It returns an instance of the `Tree` class with vertex `v` as the root. The method stores the vertices searched in the list `searchOrder` (line 147), the parent of each vertex in the array `parent` (line 148), and uses the `isVisited` array to indicate whether a vertex has been visited (line 153). It invokes the helper method `dfs(v, parent, searchOrder, isVisited)` to perform a depth-first search (line 156).

In the recursive helper method, the search starts from vertex `v`. `v` is added to `searchOrder` in line 166 and is marked as visited (line 167). 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 171). The method returns when all vertices are visited for a connected graph, or in a connected component.

Listing 30.9 gives a test program that displays a DFS for the graph in Figure 30.1 starting from Chicago. The graphical illustration of the DFS starting from Chicago is shown in Figure 30.13. For an interactive GUI demo of DFS, go to www.cs.armstrong.edu/liang/animation/USMapSearch.html.

U.S. Map Search

LISTING 30.9 `TestDFS.java`

```java
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},
```

vertices

edges

```
      {1, 0}, {1, 2}, {1, 3},
      {2, 1}, {2, 3}, {2, 4}, {2, 10},
      {3, 0}, {3, 1}, {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<String> graph =
      new UnweightedGraph<String>(edges, vertices);
    AbstractGraph<String>.Tree dfs =
      graph.dfs(graph.getIndex("Chicago"));

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

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

create a graph

get DFS

get search order

```
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 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
```

30.7.3 Applications of the DFS

The depth-first search can be used to solve many problems, such as the following:

- Detecting whether a graph is connected. Search the graph starting from any vertex. If the number of vertices searched is the same as the number of vertices in the graph, the graph is connected. Otherwise, the graph is not connected. (See Programming Exercise 30.1.)

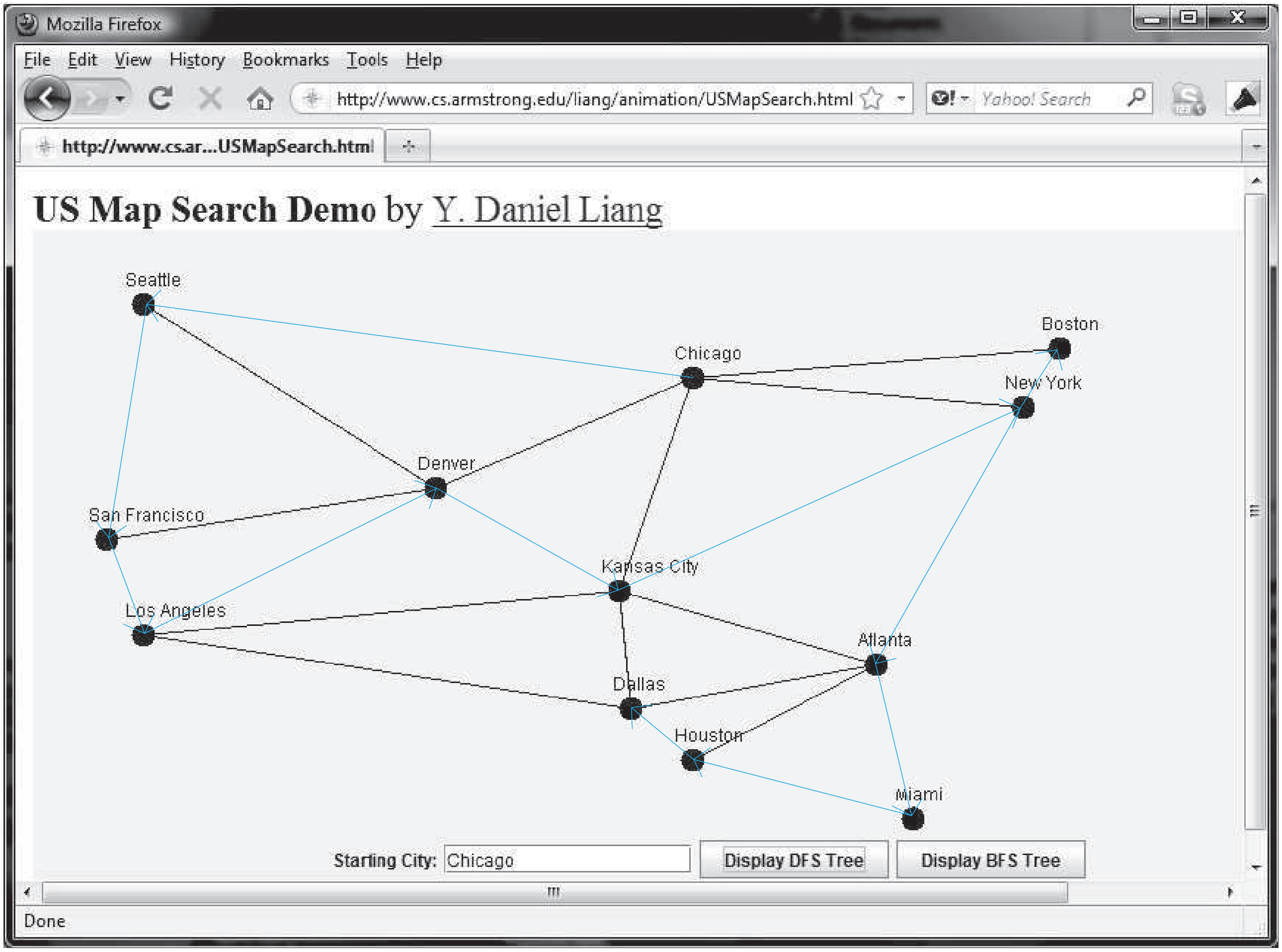


FIGURE 30.13 A DFS search starts from Chicago.

- Detecting whether there is a path between two vertices (see Programming Exercise 30.5).
- Finding a path between two vertices (see Programming Exercise 30.5).
- Finding all connected components. A connected component is a maximal connected subgraph in which every pair of vertices are connected by a path. (See Programming Exercise 30.4.)
- Detecting whether there is a cycle in the graph (see Programming Exercise 30.6).
- Finding a cycle in the graph (see Programming Exercise 30.7).
- Finding a Hamiltonian path/cycle. A *Hamiltonian path* in a graph is a path that visits each vertex in the graph exactly once. A *Hamiltonian cycle* visits each vertex in the graph exactly once and returns to the starting vertex. (See Programming Exercise 30.17.)

The first six problems can be easily solved using the `dfs` method in Listing 30.3. To find a Hamiltonian path/cycle, you have to explore all possible DFSs to find the one that leads to the longest path. The Hamiltonian path/cycle has many applications, including for solving the well-known Knight's Tour problem, which is presented in Supplement VI.C on the Companion Website.

Check Point

MyProgrammingLab™

30.12 What is depth-first search?

30.13 Draw a DFS tree for the graph in Figure 30.3b starting from node **A**.

30.14 Draw a DFS tree for the graph in Figure 30.1 starting from vertex **Atlanta**.

30.15 What is the return type from invoking **dfs(v)**?

30.16 The depth-first search algorithm described in Listing 30.8 uses recursion. Alternatively, you can use a stack to implement it, as shown below. Point out 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;
  }
}
```

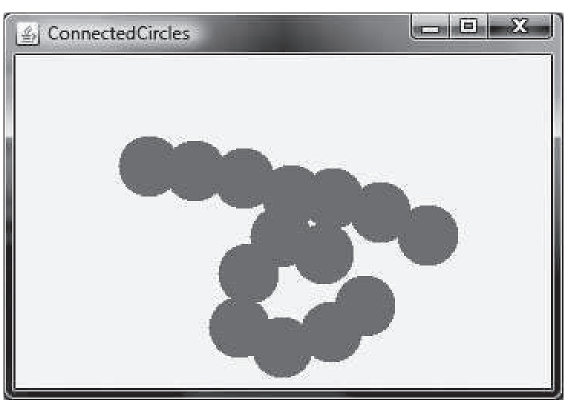
30.8 Case Study: The Connected Circles Problem

Key Point

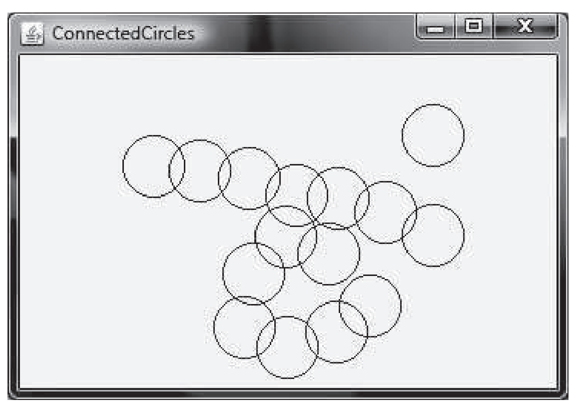
The connected circles problem is to determine whether all circles in a two-dimensional plane are connected. This problem can be solved using a depth-first traversal.

The DFS algorithm has many applications. This section applies the DFS algorithm to solve the connected circles problem.

In the connected circles problem, you determine whether all the circles in a two-dimensional plane are connected. If all the circles are connected, they are painted as filled circles, as shown in Figure 30.14a. Otherwise, they are not filled, as shown in Figure 30.14b.



(a) Circles are connected



(b) Circles are not connected

FIGURE 30.14 You can apply DFS to determine whether the circles are connected.

We will write a program that lets the user create a circle by clicking a mouse in a blank area that is not currently covered by a circle. As the circles are added, the circles are repainted filled if they are connected or unfilled otherwise.

We will create a graph to model the problem. Each circle is a vertex in the graph. Two circles are connected if they overlap. We apply the DFS in the graph, and if all vertices are found in the depth-first search, the graph is connected.

The program is given in Listing 30.10.

LISTING 30.10 ConnectedCircles.java

```
import java.util.List;
import java.util.ArrayList;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ConnectedCircles extends JApplet {
  // Circles are stored in a list
  private List<Circle> circles = new ArrayList<Circle>();

  public ConnectedCircles() {
    add(new CirclePanel()); // Add to circle panel to applet
  }

  /** Panel for displaying circles */
  class CirclePanel extends JPanel {
    public CirclePanel() {
      addMouseListener(new MouseAdapter() {
        @Override
        public void mouseClicked(MouseEvent e) {
          if (!isInsideACircle(e.getPoint())) { // Add a new circle
            circles.add(new Circle(e.getX(), e.getY()));
            repaint();
          }
        }
      });
    }

    /** Returns true if the point is inside an existing circle */
    private boolean isInsideACircle(Point p) {
      for (Circle circle: circles)
        if (circle.contains(p))
          return true;

      return false;
    }

    @Override
    protected void paintComponent(Graphics g) {
      if (circles.size() == 0)
        return; // Nothing to paint

      super.paintComponent(g);

      // Build the edges
      List<AbstractGraph.Edge> edges
        = new ArrayList<AbstractGraph.Edge>();
      for (int i = 0; i < circles.size(); i++)
        for (int j = i + 1; j < circles.size(); j++)
          if (circles.get(i).overlaps(circles.get(j))) {
            edges.add(new AbstractGraph.Edge(i, j));
            edges.add(new AbstractGraph.Edge(j, i));
          }
```

circles in a list

panel for showing circles

mouse clicked

add a new circle

inside circle check

no circles

create edges

create a graph
get a search tree
connected?
connected
not connected
the Circle class
contains a point?
two circles overlap
main method omitted

```
54
55         // Create a graph with circles as vertices
56         Graph<Circle> graph
57           = new UnweightedGraph<Circle>(edges, circles);
58         AbstractGraph<Circle>.Tree tree = graph.dfs(0); // a DFS tree
59         boolean isAllCirclesConnected = circles.size() == tree
60             .getNumberOfVerticesFound();
61
62         for (Circle circle : circles) {
63           int radius = circle.radius;
64           if (isAllCirclesConnected) { // All circles are connected
65             g.setColor(Color.RED);
66             g.fillOval(circle.x - radius, circle.y - radius,
67               2 * radius, 2 * radius);
68           } else
69             // circles are not all connected
70             g.drawOval(circle.x - radius, circle.y - radius,
71               2 * radius, 2 * radius);
72         }
73       }
74     }
75
76     private static class Circle {
77       int radius = 20;
78       int x, y;
79
80       Circle(int x, int y) {
81         this.x = x;
82         this.y = y;
83       }
84
85       public boolean contains(Point p) {
86         double d = distance(x, y, p.x, p.y);
87         return d <= radius;
88       }
89
90       public boolean overlaps(Circle circle) {
91         return distance(this.x, this.y, circle.x, circle.y) <= radius
92           + circle.radius;
93       }
94
95       private static double distance(int x1, int y1, int x2, int y2) {
96         return Math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
97       }
98     }
99   }
```

The `Circle` class is defined in lines 76–98. It contains the data fields `x`, `y`, and `radius`, which specify the circle's center location and radius. It also defines the `contains` and `overlaps` methods (lines 85–93) for checking whether a point is inside the circle and whether two circles overlap.

When the user clicks the mouse outside of any existing circle, a new circle is created centered at the mouse point and the circle is added to the list `circles` (line 22).

To detect whether the circles are connected, the program constructs a graph (lines 56–57). The circles are the vertices of the graph. The edges are constructed in lines 46–53. Two circle vertices are connected if they overlap (line 50). The DFS of the graph results in a tree (line 58). The tree's `getNumberOfVerticesFound()` returns the number of vertices searched. If it is equal to the number of circles, all circles are connected (lines 59–60).

Check Point

MyProgrammingLab™

30.17 How is a graph created for the connected circles problem?

30.18 When you click the mouse inside a circle, does the program create a new circle?

30.19 How does the program know if all circles are connected?

30.9 Breadth-First Search (BFS)

Key Point

The breadth-first search of a graph visits the vertices level by level. The first level consists of the starting vertex. Each next level consists of the vertices adjacent to the vertices in the preceding level.

The breadth-first traversal of a graph is like the breadth-first traversal of a tree discussed in Section 27.2.4, 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, 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, it skips a vertex if it has already been visited.

30.9.1 Breadth-First Search Algorithm

The algorithm for the breadth-first search starting from vertex *v* in a graph is described in Listing 30.11.

LISTING 30.11 Breadth-First Search Algorithm

```
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;
    add u into a list of traversed vertices;
    for each neighbor w of u
      if w has not been visited {
        add w into the queue;
        mark w visited;
      }
  }
}
```

create a queue
enqueue v
dequeue into u
u traversed
check a neighbor w
is w visited?
enqueue w

Consider the graph in Figure 30.15a. Suppose you start the breadth-first search from vertex 0. First visit `0`, then visit all its neighbors, `1`, `2`, and `3`, as shown in Figure 30.15b. Vertex 1 has three neighbors: 0, 2, and 4. Since 0 and 2 have already been visited, you will now visit just `4`, as shown in Figure 30.15c. Vertex 2 has three neighbors, 0, 1, and 3, which have all been visited. Vertex 3 has three neighbors, 0, 2, and 4, which have all been visited. Vertex 4 has two neighbors, 1 and 3, which have all been visited. Hence, the search ends.

BFS time complexity

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

30.9.2 Implementation of Breadth-First Search

The **`bfs(int v)`** method is defined in the **`Graph`** interface and implemented in the **`AbstractGraph`** class in Listing 30.3 (lines 179–204). It returns an instance of the **`Tree`** class with vertex **`v`** as the root. The method stores the vertices searched in the list **`searchOrder`** (line 180), the parent of each vertex in the array **`parent`** (line 181), uses a linked list for a

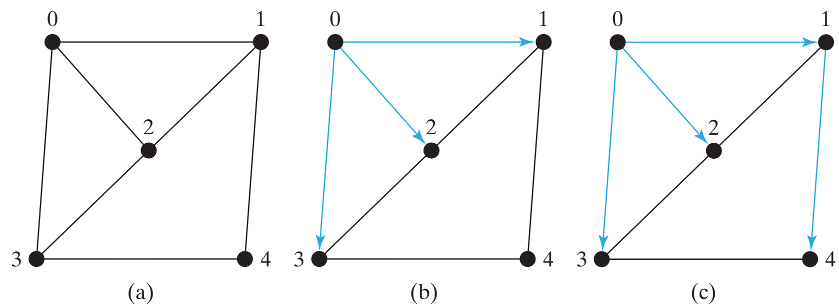


Figure 30.15 Breadth-first search visits a node, then its neighbors, then its neighbors's neighbors, and so on.

queue (lines 185–186), and uses the **isVisited** array to indicate whether a vertex has been visited (line 187). The search starts from vertex **v**. **v** is added to the queue in line 188 and is marked as visited (line 189). The method now examines each vertex **u** in the queue (line 192) and adds it to **searchOrder** (line 193). The method adds each unvisited neighbor **w** of **u** to the queue (line 196), sets its parent to **u** (line 197), and marks it as visited (line 198).

Listing 30.12 gives a test program that displays a BFS for the graph in Figure 30.1 starting from Chicago. The graphical illustration of the BFS starting from Chicago is shown in Figure 30.16. For an interactive GUI demo of BFS, go to www.cs.armstrong.edu/liang/animation/USMapSearch.html.

Listing 30.12 TestBFS.java

vertices

edges

create a graph

create a BFS tree

get search order

```java
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, 1}, {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<String> graph =
      new UnweightedGraph<String>(edges, vertices);
    AbstractGraph<String>.Tree bfs =
      graph.bfs(graph.getIndex("Chicago"));

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

```java

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

```
12 vertices are searched in this 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 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
```

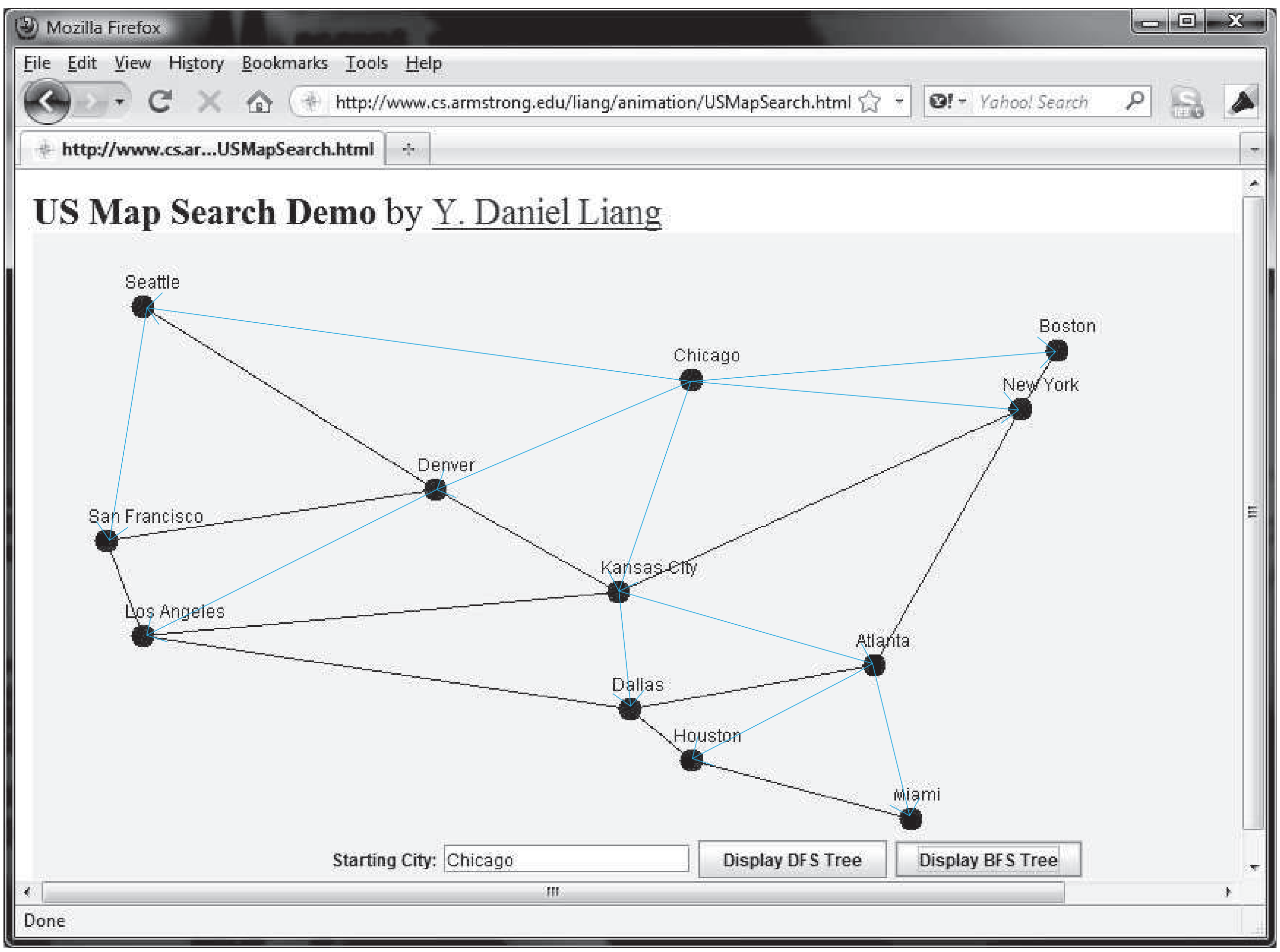


FIGURE 30.16 BFS search starts from Chicago.

30.9.3 Applications of the BFS

Many of the problems solved by the DFS can also be solved using the BFS. 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 the 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 Check Point Question 30.24.)
- 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 Programming Exercise 30.6).
- Finding a cycle in the graph (see Programming Exercise 30.7).
- 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 Programming Exercise 30.8.)

Check Point
MyProgrammingLab™

30.20 What is the return type from invoking `bfs(v)`?

30.21 What is breadth-first search?

30.22 Draw a BFS tree for the graph in Figure 30.3b starting from node `A`.

30.23 Draw a BFS tree for the graph in Figure 30.1 starting from vertex `Atlanta`.

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

30.10 Case Study: The Nine Tails Problem

Key Point

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

The nine tails problem is 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 moves that lead to all coins being face down. For example, start with the nine coins as shown in Figure 30.17a. After you flip the second coin in the last row, the nine coins are now as shown in Figure 30.17b. After you flip the second coin in the first row, the nine coins are all face down, as shown in Figure 30.17c.

H	H	H
T	T	T
H	H	H

(a)

H	H	H
T	H	T
T	T	T

(b)

T	T	T
T	T	T
T	T	T

(c)

FIGURE 30.17 The problem is solved when all coins are face down.

We will write a program that prompts the user to enter an initial state of the nine coins and displays the solution, as shown in the following sample run.

```
Enter the initial nine coins Hs and Ts: HHHTTTHHH ↵Enter

The steps to flip the coins are
HHH
TTT
HHH

HHH
THT
TTT

TTT
TTT
TTT
```

Each state of the nine coins represents a node in the graph. For example, the three states in Figure 30.17 correspond to three nodes in the graph. For convenience, we use a 3 × 3 matrix to represent all nodes and use **0** for heads and **1** for tails. Since there are nine cells and each cell is either **0** or **1**, there are a total of 2^9 (512) nodes, labeled **0**, **1**, . . . , and **511**, as shown in Figure 30.18.

$$\underset{0}{\begin{array}{|c|c|c|}\hline 0&0&0\\\hline 0&0&0\\\hline 0&0&0\\\hline\end{array}}\quad \underset{1}{\begin{array}{|c|c|c|}\hline 0&0&0\\\hline 0&0&0\\\hline 0&0&1\\\hline\end{array}}\quad \underset{2}{\begin{array}{|c|c|c|}\hline 0&0&0\\\hline 0&0&0\\\hline 0&1&0\\\hline\end{array}}\quad \underset{3}{\begin{array}{|c|c|c|}\hline 0&0&0\\\hline 0&0&0\\\hline 0&1&1\\\hline\end{array}}\quad \cdots\cdots\quad \underset{511}{\begin{array}{|c|c|c|}\hline 1&1&1\\\hline 1&1&1\\\hline 1&1&1\\\hline\end{array}}$$

FIGURE 30.18 There are total of 512 nodes labeled in this order: **0**, **1**, **2**, . . . , **511**.

We assign an edge from node **v** to **u** if there is a legal move from **u** to **v**. Figure 30.19 shows a partial graph. Note there is an edge from **511** to **47**, since you can flip a cell in node **47** to become node **511**.

The last node in Figure 30.18 represents the state of nine face-down coins. For convenience, we call this last node the *target node*. Thus, the target node is labeled **511**. Suppose the initial state of the nine tails problem corresponds to the node **s**. The problem is reduced to finding the shortest path from node **s** to the target node, which is equivalent to finding the path from node **s** to the target node in a BFS tree rooted at the target node.

Now the task is to build a graph that consists of 512 nodes labeled **0**, **1**, **2**, . . . , **511**, and edges among the nodes. Once the graph is created, obtain a BFS tree rooted at node **511**. From the BFS tree, you can find the shortest path from the root to any vertex. We will create a class named **NineTailModel**, which contains the method to get the shortest path from the target node to any other node. The class UML diagram is shown in Figure 30.20.

Visually, a node is represented in a 3 × 3 matrix with the letters **H** and **T**. In our program, we use a single-dimensional array of nine characters to represent a node. For example, the node for vertex **1** in Figure 30.18 is represented as {**'H'**, **'H'**, **'H'**, **'H'**, **'H'**, **'H'**, **'H'**, **'H'**, **'T'**} in the array.

The **getEdges()** method returns a list of **Edge** objects.

The **getNode(index)** method returns the node for the specified index. For example, **getNode(0)** returns the node that contains nine **H**s. **getNode(511)** returns the node that contains nine **T**s. The **getIndex(node)** method returns the index of the node.

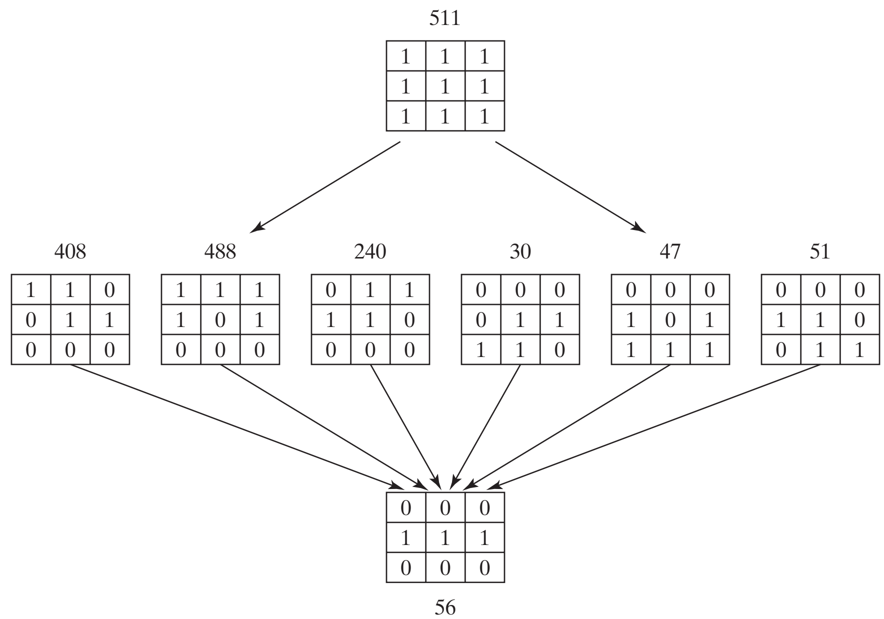


FIGURE 30.19 If node **u** becomes node **v** after cells are flipped, assign an edge from **v** to **u**.

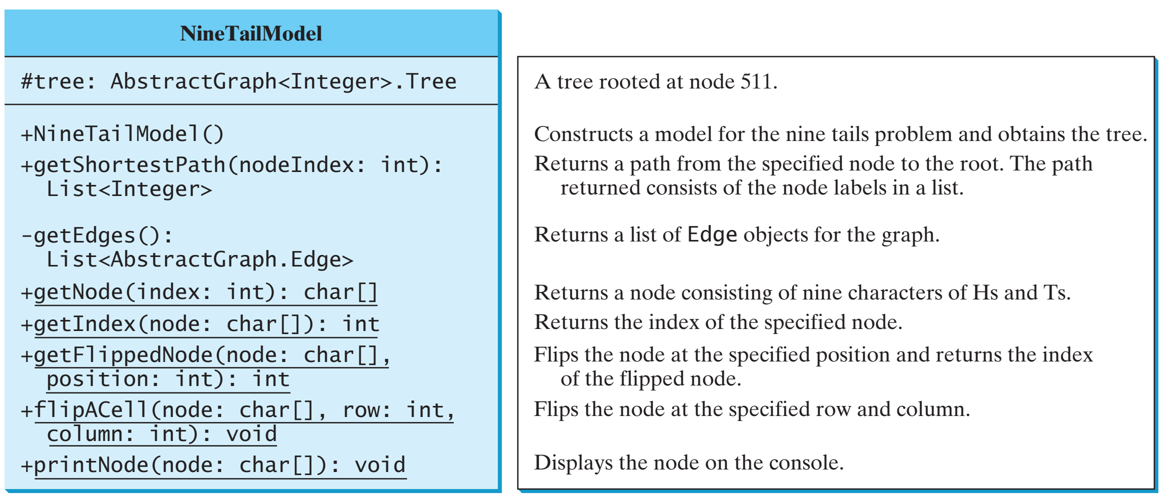


FIGURE 30.20 The `NineTailModel` class models the nine tails problem using a graph.

Note that the data field `tree` is defined as protected so that it can be accessed from the `WeightedNineTail` subclass in the next chapter.

The `getFlippedNode(char[] node, int position)` method flips the node at the specified position and its adjacent positions. This method returns the index of the new node. For example, for node `56` in Figure 30.19, flip it at position `0`, and you will get node `51`. If you flip node `56` at position `1`, you will get node `47`.

The `flipACell(char[] node, int row, int column)` method flips a node at the specified row and column. For example, if you flip node `56` at row `0` and column `0`, the new node is `408`. If you flip node `56` at row `2` and column `0`, the new node is `30`.

Listing 30.13 shows the source code for NineTailModel.java.

LISTING 30.13 NineTailModel.java

```java
import java.util.*;

public class NineTailModel {
  public final static int NUMBER_OF_NODES = 512;
  protected AbstractGraph<Integer>.Tree tree; // Define a tree

  /** Construct a model */
  public NineTailModel() {
    // Create edges
    List<AbstractGraph.Edge> edges = getEdges();

    // Create a graph
    UnweightedGraph<Integer> graph = new UnweightedGraph<Integer>(
      edges, NUMBER_OF_NODES);

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

  /** Create all edges for the graph */
  private List<AbstractGraph.Edge> getEdges() {
    List<AbstractGraph.Edge> edges =
      new ArrayList<AbstractGraph.Edge>(); // Store edges

    for (int u = 0; u < NUMBER_OF_NODES; u++) {
      for (int k = 0; k < 9; k++) {
        char[] node = getNode(u); // Get the node for vertex u
        if (node[k] == 'H') {
          int v = getFlippedNode(node, k);
          // Add edge (v, u) for a legal move from node u to node v
          edges.add(new AbstractGraph.Edge(v, u));
        }
      }
    }

    return edges;
  }

  public static int getFlippedNode(char[] node, int position) {
    int row = position / 3;
    int column = position % 3;

    flipACell(node, row, column);
    flipACell(node, row - 1, column);
    flipACell(node, row + 1, column);
    flipACell(node, row, column - 1);
    flipACell(node, row, column + 1);

    return getIndex(node);
  }

  public static void flipACell(char[] node, int row, int column) {
    if (row >= 0 && row <= 2 && column >= 0 && column <= 2) {
      // Within the boundary
      if (node[row * 3 + column] == 'H')
        node[row * 3 + column] = 'T'; // Flip from H to T
      else
        node[row * 3 + column] = 'H'; // Flip from T to H
```

declare a tree

create edges

create graph

create tree

get edges

add an edge

flip cells

flip a cell

```
59        }
60      }
61
```

get index for a node

```
62      public static int getIndex(char[] node) {
63        int result = 0;
64
65        for (int i = 0; i < 9; i++)
66          if (node[i] == 'T')
67            result = result * 2 + 1;
68          else
69            result = result * 2 + 0;
70
71        return result;
72      }
73
```

For example:
index: 3
node: HHHHHHHTT

H	H	H
H	H	H
H	T	T

get node for an index

```
74      public static char[] getNode(int index) {
75        char[] result = new char[9];
76
77        for (int i = 0; i < 9; i++) {
78          int digit = index % 2;
79          if (digit == 0)
80            result[8 - i] = 'H';
81          else
82            result[8 - i] = 'T';
83          index = index / 2;
84        }
85
86        return result;
87      }
88
```

For example:
node: THHHHHHTT
index: 259

T	H	H
H	H	H
H	T	T

shortest path

```
89      public List<Integer> getShortestPath(int nodeIndex) {
90        return tree.getPath(nodeIndex);
91      }
92
```

display a node

```
93      public static void printNode(char[] node) {
94        for (int i = 0; i < 9; i++)
95          if (i % 3 != 2)
96            System.out.print(node[i]);
97          else
98            System.out.println(node[i]);
99
100       System.out.println();
101     }
102   }
```

The constructor (lines 8–18) creates a graph with 512 nodes, and each edge corresponds to the move from one node to the other (line 10). From the graph, a BFS tree rooted at the target node **511** is obtained (line 17).

To create edges, the **getEdges** method (lines 21–37) checks each node **u** to see if it can be flipped to another node **v**. If so, add (**v**, **u**) to the **Edge** list (line 31). The **getFlippedNode(node, position)** method finds a flipped node by flipping an **H** cell and its neighbors in a node (lines 43–47). The **flipACell(node, row, column)** method actually flips an **H** cell and its neighbors in a node (lines 52–60).

The **getIndex(node)** method is implemented in the same way as converting a binary number to a decimal number (lines 62–72). The **getNode(index)** method returns a node consisting of the letters **H** and **T** (lines 74–87).

The `getShortestpath(nodeIndex)` method invokes the `getPath(nodeIndex)` method to get the vertices in the shortest path from the specified node to the target node (lines 89–91).

The `printNode(node)` method displays a node on the console (lines 93–101).

Listing 30.14 gives a program that prompts the user to enter an initial node and displays the steps to reach the target node.

Listing 30.14 NineTail.java

```
1  import java.util.Scanner;
2
3  public class NineTail {
4    public static void main(String[] args) {
5      // Prompt the user to enter nine coins' Hs and Ts
6      System.out.print("Enter the initial nine coins Hs and Ts: ");
7      Scanner input = new Scanner(System.in);
8      String s = input.nextLine();
9      char[] initialNode = s.toCharArray();
10
11     NineTailModel model = new NineTailModel();
12     java.util.List<Integer> path =
13       model.getShortestPath(NineTailModel.getIndex(initialNode));
14
15     System.out.println("The steps to flip the coins are ");
16     for (int i = 0; i < path.size(); i++)
17       NineTailModel.printNode(
18         NineTailModel.getNode(path.get(i).intValue()));
19   }
20 }
```

initial node

create model

get shortest path

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

Check Point

MyProgrammingLab™

30.25 How are the nodes created for the graph in `NineTailModel`?

30.26 How are the edges created for the graph in `NineTailModel`?

30.27 What is returned after invoking `getIndex("HTHTTTHHH".toCharArray())` in Listing 30.13? What is returned after invoking `getNode(46)` in Listing 30.13?

30.28 If lines 26 and 27 are swapped in Listing 30.13, NineTailModel.java, will the program work? Why not?

Key Terms

adjacency list 1054
adjacency matrix 1054
adjacent vertices 1050
breadth-first search 1069
complete graph 1050
cycle 1050
degree 1050
depth-first search 1069
directed graph 1049
graph 1049
incident edges 1050
parallel edge 1050
Seven Bridges of Königsberg 1048
simple graph 1050
spanning tree 1050
tree 1050
undirected graph 1049
unweighted graph 1049
weighted graph 1049

Chapter Summary

1. 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 graphs.

2. Graph traversal is the process of visiting each vertex in the graph exactly once. You learned two popular ways for traversing a graph: the *depth-first search* (DFS) and *breadth-first search* (BFS).

3. DFS and BFS can be used to solve many problems such as detecting whether a graph is connected, detecting whether there is a cycle in the graph, and finding the shortest path between two vertices.

Test Questions

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

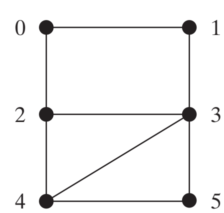
MyProgrammingLab™

Programming Exercises

Sections 30.6–30.10

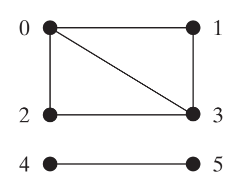
***30.1** (*Test whether a graph is connected*) Write a program that reads a graph from a file and determines whether the graph is connected. The first line in the file contains a number that indicates the number of vertices (`n`). The vertices are labeled as `0`, `1`, . . . , `n-1`. Each subsequent line, with the format `u v1 v2 ...`, describes edges (`u`, `v1`), (`u`, `v2`), and so on. Figure 30.21 gives the examples of two files for their corresponding graphs.

```
File
6
0 1 2
1 0 3
2 0 3 4
3 1 2 4 5
4 2 3 5
5 3 4
```



(a)

```
File
6
0 1 2 3
1 0 3
2 0 3
3 0 1 2
4 5
5 4
```



(b)

Figure 30.21 The vertices and edges of a graph can be stored in a file.

Your program should prompt the user to enter the name of the file, then it should read data from the file, create an instance `g` of `UnweightedGraph`, invoke `g.printEdges()` to display all edges, and invoke `dfs()` to obtain an instance `tree` of `AbstractGraph.Tree`. If `tree.getNumberOfVerticesFound()` is the same as the number of vertices in the graph, the graph is connected. Here is a sample run of the program:

```
Enter a file name: c:\exercise\GraphSample1.txt  Enter
The number of vertices is 6
Vertex 0: (0, 1) (0, 2)
Vertex 1: (1, 0) (1, 3)
Vertex 2: (2, 0) (2, 3) (2, 4)
Vertex 3: (3, 1) (3, 2) (3, 4) (3, 5)
Vertex 4: (4, 2) (4, 3) (4, 5)
Vertex 5: (5, 3) (5, 4)
The graph is connected
```

(*Hint*: Use `new UnweightedGraph(list, numberOfVertices)` to create a graph, where `list` contains a list of `AbstractGraph.Edge` objects. Use `new AbstractGraph.Edge(u, v)` to create an edge. Read the first line to get the number of vertices. Read each subsequent line into a string `s` and use `s.split("[\\s+]")` to extract the vertices from the string and create edges from the vertices.)

***30.2** (*Create a file for a graph*) Modify Listing 30.1, TestGraph.java, to create a file representing `graph1`. The file format is described in Programming Exercise 30.1. Create the file from the array defined in lines 8–21 in Listing 30.1. The number of vertices for the graph is `12`, which will be stored in the first line of the file. The contents of the file should be as follows:

12
0 1 3 5
1 0 2 3
2 1 3 4 10
3 0 1 2 4 5
4 2 3 5 7 8 10
5 0 3 4 6 7
6 5 7
7 4 5 6 8
8 4 7 9 10 11
9 8 11
10 2 4 8 11
11 8 9 10

***30.3** (*Implement DFS using a stack*) The depth-first search algorithm described in Listing 30.8 uses recursion. Implement it without using recursion.

***30.4** (*Find connected components*) Create a new class named `MyGraph` as a subclass of `UnweightedGraph` that contains a method for finding all connected components in a graph with the following header:

```
public List<List<Integer>> getConnectedComponents();
```

The method returns a `List<List<Integer>>`. Each element in the list is another list that contains all the vertices in a connected component. For example, for the graph in Figure 30.21b, `getConnectedComponents()` returns `[[0, 1, 2, 3], [4, 5]]`.

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

```
public List<Integer> getPath(int u, int v);
```

The method returns a **List<Integer>** that contains all the vertices in a path from **u** to **v** in this order. Using the BFS approach, you can obtain the shortest path from **u** to **v**. If there isn't a path from **u** to **v**, the method returns **null**.

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

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

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

```
public List<Integer> getACycle(int u);
```

The method returns a **List** that contains all the vertices in a cycle starting from **u**. If the graph doesn't have any cycles, the method returns **null**.

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

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

****30.9** (*Get bipartite sets*) Add a new method in **AbstractGraph** with the following header to return two bipartite sets if the graph is bipartite:

```
public List<List<Integer>> getBipartite();
```

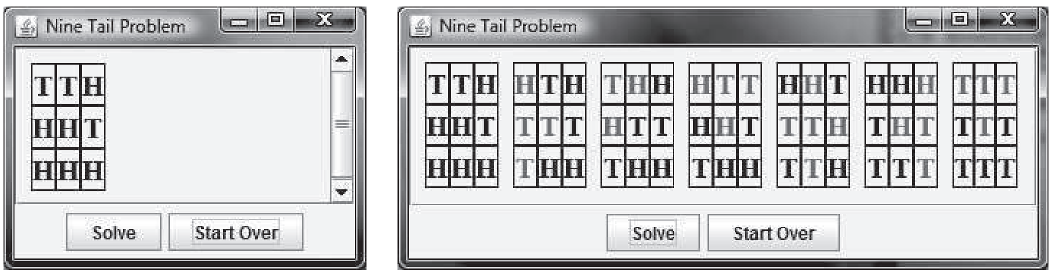
The method returns a **List** that contains two sublists, each of which contains a set of vertices. If the graph is not bipartite, the method returns **null**.

***30.10** (*Find the shortest path*) 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 30.1. Your program should prompt the user to enter the name of the file, then two vertices, and should display the shortest path between the two vertices. For example, for the graph in Figure 30.21a, the shortest path between **0** and **5** may be displayed as **0 1 3 5**.

Here is a sample run of the program:

```
Enter a file name: c:\exercise\GraphSample1.txt
Enter two vertices (integer indexes): 0 5
The number of vertices is 6
Vertex 0: (0, 1) (0, 2)
Vertex 1: (1, 0) (1, 3)
Vertex 2: (2, 0) (2, 3) (2, 4)
Vertex 3: (3, 1) (3, 2) (3, 4) (3, 5)
Vertex 4: (4, 2) (4, 3) (4, 5)
Vertex 5: (5, 3) (5, 4)
The path is 0 1 3 5
```

****30.11** (*Revise Listing 30.14, NineTail.java*) The program in Listing 30.14 lets the user enter an input for the nine tails problem from the console and displays the result on the console. Write an applet that lets the user set an initial state of the nine coins (see Figure 30.22a) and click the *Solve* button to display the solution, as shown in Figure 30.22b. Initially, the user can click the mouse button to flip a coin. Set a red color on the flipped cells.



(a) (b)

FIGURE 30.22 The applet solves the nine tails problem.

****30.12** (*Variation of the nine tails problem*) In the nine tails problem, when you flip a coin, the horizontal and vertical neighboring cells are also flipped. Rewrite the program, assuming that all neighboring cells including the diagonal neighbors are also flipped.

****30.13** (*4 × 4 16 tails model*) The nine tails problem in the text uses a 3 × 3 matrix. Assume that you have 16 coins placed in a 4 × 4 matrix. Create a new model class named `TailModel16`. Create an instance of the model and save the object into a file named **TailModel16.dat**.

****30.14** (*4 × 4 16 tails view*) Listing 30.14, NineTail.java, presents a solution for the nine tails problem. Revise this program for the 4 × 4 16 tails problem. Your program should read the model object created from the preceding exercise.

****30.15** (*Dynamic graphs*) Write a program that lets the user create a graph dynamically. The user can create a vertex by entering its name and location, as shown in Figure 30.23. The user can also create an edge to connect two vertices. To simplify the program, assume that the vertex names are the same as the vertex indices. You have to add the vertex indices `0`, `1`, . . . , `n`, in this order. The user can specify two vertices and let the program display their shortest path in red.

****30.16** (*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 a degree >= *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 a 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.)

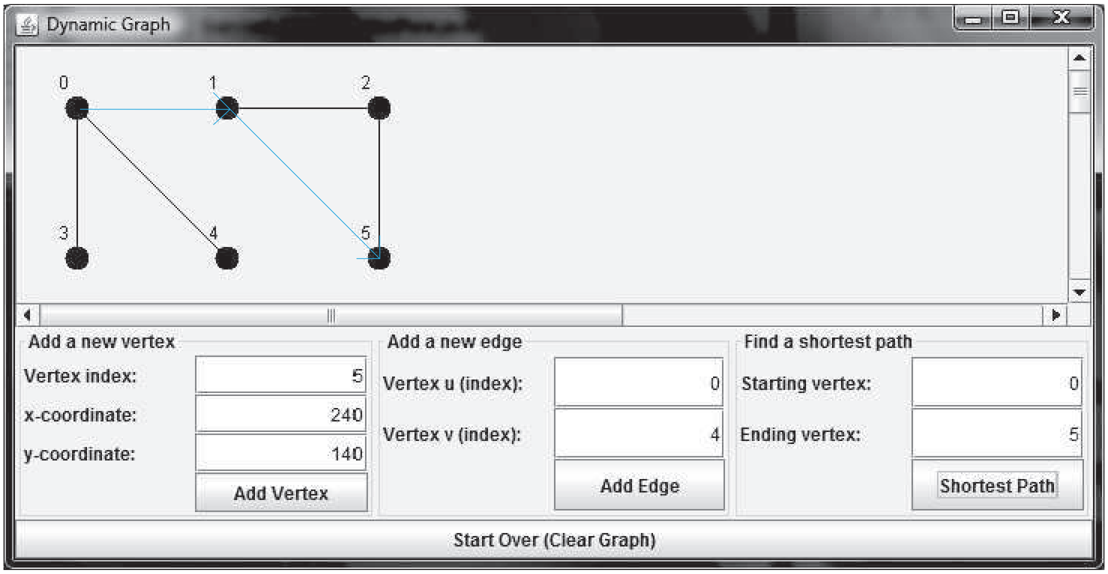


Figure 30.23 The program can add vertices and edges and display the shortest path between two specified vertices.

*****30.17** (*Hamiltonian cycle*) The Hamiltonian path algorithm is implemented in Supplement VI.C. Add the following `getHamiltonianCycle` method in the `Graph` interface and implement it in the `AbstractGraph` class:

```
/** Return a Hamiltonian cycle
  * Return null if the graph doesn't contain a Hamiltonian cycle */
public List<Integer> getHamiltonianCycle()
```

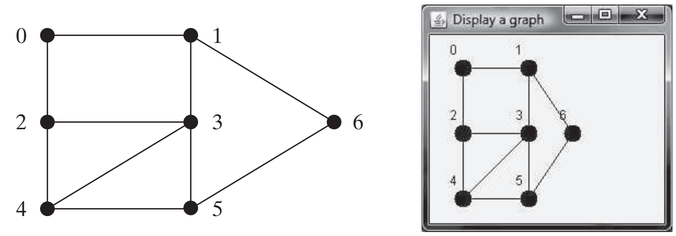
*****30.18** (*Knight's Tour cycle*) Rewrite KnightTourApp.java in the case study in Supplement VI.C to find a knight's tour that visits each square in a chessboard and returns to the starting square. Reduce the Knight's Tour cycle problem to the problem of finding a Hamiltonian cycle.

****30.19** (*Display a DFS/BFS tree in a graph*) Modify `GraphView` in Listing 30.6 to add a new data field `tree` with a set method. The edges in the tree are displayed in red. Write a program that displays the graph in Figure 30.1 and the DFS/BFS tree starting from a specified city, as shown in Figures 30.13 and 30.16. If a city not in the map is entered, the program displays a dialog box to alert the user.

***30.20** (*Display a graph*) Write a program that reads a graph from a file and displays it. The first line in the file contains a number that indicates the number of vertices (`n`). The vertices are labeled `0`, `1`, . . . , `n-1`. Each subsequent line, with the format `u x y v1 v2 ...`, describes the position of `u` at (`x`, `y`) and edges (`u`, `v1`), (`u`, `v2`), and so on. Figure 30.24a gives an example of the file for their corresponding graph. Your program prompts the user to enter the name of the file, reads data from the file, and displays the graph on a panel using `GraphView`, as shown in Figure 30.24b.

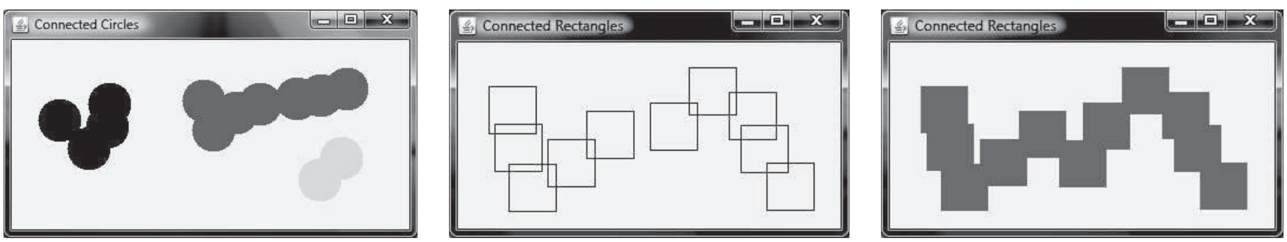
****30.21** (*Display sets of connected circles*) Modify Listing 30.10, ConnectedCircles.java, to display sets of connected circles in different colors. That is, if two circles are connected, they are displayed using the same color; otherwise, they are not in same color, as shown in Figure 30.25. (*Hint*: See Programming Exercise 30.4.)

```
File
7
0  30   30 1 2
1  90   30 0 3 6
2  30   90 0 3 4
3  90   90 1 2 4 5
4  30  150 2 3 5
5  90  150 3 4 6
6 130   90 1 5
```



(a) (b)

FIGURE 30.24 The program reads the information about the graph and displays it visually.



(a) (b) (c)

FIGURE 30.25 (a) Connected circles are displayed in the same color. (b) Rectangles are not filled with a color if they are not connected. (c) Rectangles are filled with a color if they are connected.

***30.22** (*Move a circle*) Modify Listing 30.10, ConnectedCircles.java, to enable the user to drag and move a circle.

****30.23** (*Connected rectangles*) Listing 30.10, ConnectedCircles.java, allows the user to create circles and determine whether they are connected. Rewrite the program for rectangles. The program lets the user create a rectangle by clicking a mouse in a blank area that is not currently covered by a rectangle. As the rectangles are added, the rectangles are repainted as filled if they are connected or are unfilled otherwise, as shown in Figure 30.25b–c.

***30.24** (*Remove a circle*) Modify Listing 30.10, ConnectedCircles.java, to enable the user to remove a circle when the mouse is clicked inside the circle.

*****30.25** (*Graph visualization tool*) Develop an applet as shown in Figure 30.4, with the following requirements: (1) The radius of each vertex is 20 pixels. (2) The user clicks the left-mouse button to place a vertex centered at the mouse point, provided that the mouse point is not inside or too close to an existing vertex. (3) The user clicks the right-mouse button inside an existing vertex to remove the vertex. (4) The user presses a mouse button inside a vertex, drags to another vertex, and then releases the button to create an edge. (5) The user drags a vertex while pressing the *CTRL* key to move a vertex. (6) The vertices are numbers starting from **0**. When a vertex is removed, the vertices are renumbered. (7) You can click the *DFS* or *BFS* button to display a DFS or BFS tree from a starting vertex. (8) You can click the *Shortest Path* button to display the shortest path between the two specified vertices.