CHAPTER

28

Graphs and Applications

Objectives

- To model real-world problems using graphs and explain the Seven Bridges of Königsberg problem (§28.1).
- To describe the graph terminologies: vertices, edges, simple graphs, weighted/unweighted graphs, and directed/undirected graphs (§28.2).
- To represent vertices and edges using lists, edge arrays, edge objects, adjacency matrices, and adjacency lists (§28.3).
- To model graphs using the `Graph` interface and the `UnweightedGraph` class (§28.4).
- To display graphs visually (§28.5).
- To represent the traversal of a graph using the `UnweightedGraph.SearchTree` class (§28.6).
- To design and implement depth-first search (§28.7).
- To solve the connected-circle problem using depth-first search (§28.8).
- To design and implement breadth-first search (§28.9).
- To solve the nine tails problem using breadth-first search (§28.10).

28.1 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 28.1. 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. At United Parcel Service, each driver makes an average 120 stops per day. There are many possible ways for ordering these stops. UPS spent hundreds of millions of dollars for 10 years to develop a system called Orion, which stands for On-Road Integrated Optimization and Navigation. The system uses graph algorithms to plan the most cost-efficient routes for each driver. This chapter studies the algorithms for unweighted graphs, and the next chapter studies those for weighted graphs.

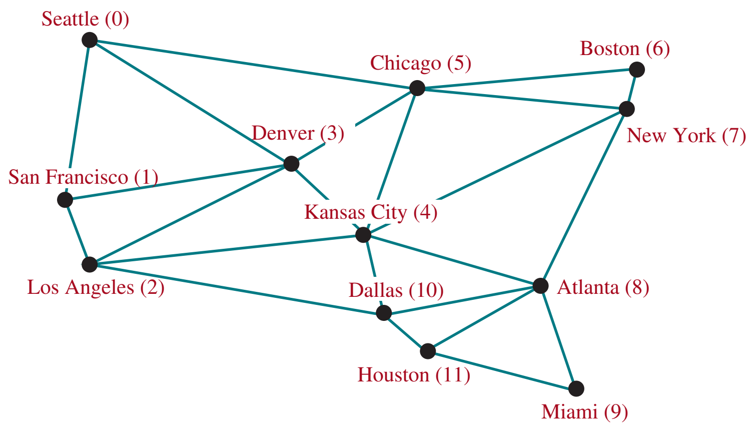


FIGURE 28.1 A graph can be used to model the flights between the cities.

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 28.2a. 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.

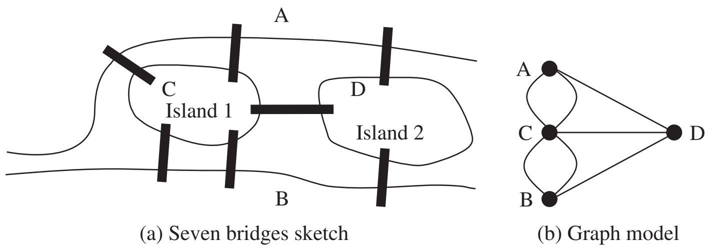


FIGURE 28.2 Seven bridges connected islands and land.

To establish a proof, Euler first abstracted the Königsberg city map by eliminating all streets, producing the sketch shown in Figure 28.2a. 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 28.2b. 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.

Graphs 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.

28.2 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 28.1 represents the flights among cities, and the graph in Figure 28.2b represents the bridges among land masses.

define a graph

A graph consists of a 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 28.1 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 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 28.3a shows a directed graph.

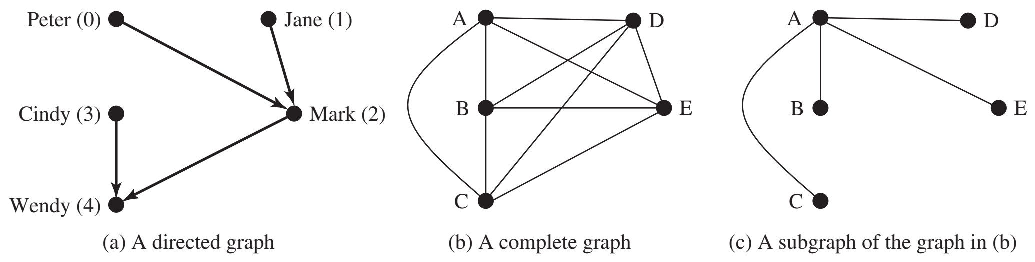


FIGURE 28.3 Graphs may appear in many forms.

In an *undirected graph*, you can move in both directions between vertices. The graph in Figure 28.1 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 28.1 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 *neighbors* if they are adjacent. Similarly, two edges are *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 doesn't have any loops or parallel edges. In a *complete graph*, every two vertices are adjacent, as shown in Figure 28.3b.

connected graph
subgraph

A graph is *connected* (also known as *strongly connected*) if there exists a path between every two vertices in the graph. A graph is *weakly connected* if it is connected when considering the graph is undirected. 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 28.3c is a subgraph of the graph in Figure 28.3b.

cycle
tree
spanning tree

Assume the graph is connected and undirected. A *cycle* is a closed path that starts from a vertex and ends at the same vertex. A connected graph is a *tree* if it does not have cycles. 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 liveexample.pearsoncmg.com/dsanimation /GraphLearning-TooleBook.html, as shown in Figure 28.4. 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.

Check Point

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

28.2.2 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.

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

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

28.3 Representing Graphs

Key Point

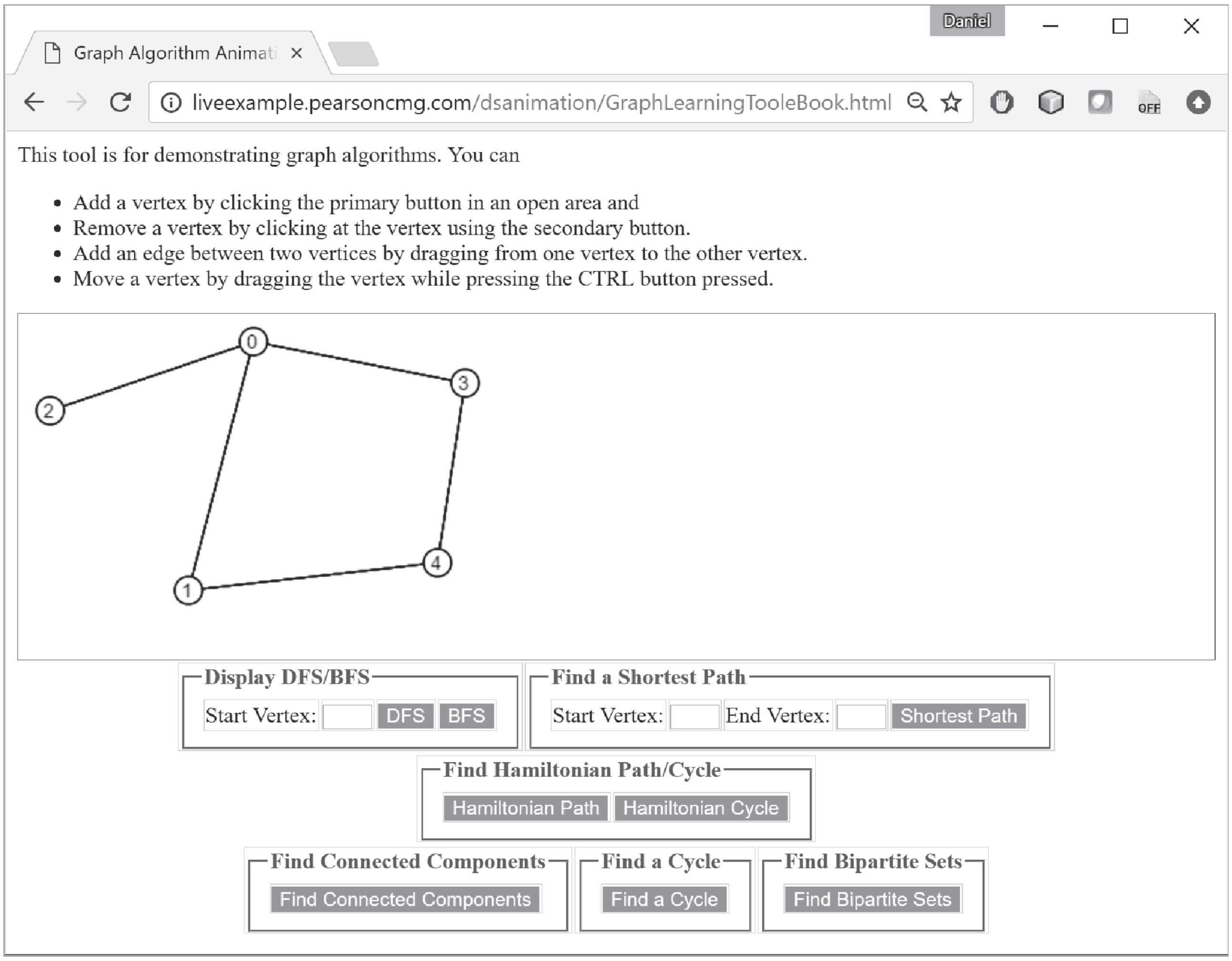
Representing a graph is to store its vertices and edges in a program. The data structure for storing a graph is arrays or lists.

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

28.3.1 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 28.1 using the following array:

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



This tool is for demonstrating graph algorithms. You can

- Add a vertex by clicking the primary button in an open area and
- Remove a vertex by clicking at the vertex using the secondary button.
- Add an edge between two vertices by dragging from one vertex to the other vertex.
- Move a vertex by dragging the vertex while pressing the CTRL button pressed.

FIGURE 28.4 You can use the tool to create a graph with mouse gestures and show DFS/BFS trees, shortest paths, and other operations.

Note

vertex type

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 of n vertices. Thus, `vertices[0]` represents `"Seattle"`, `vertices[1]` represents `"San Francisco"`, and so on, as shown in Figure 28.5.

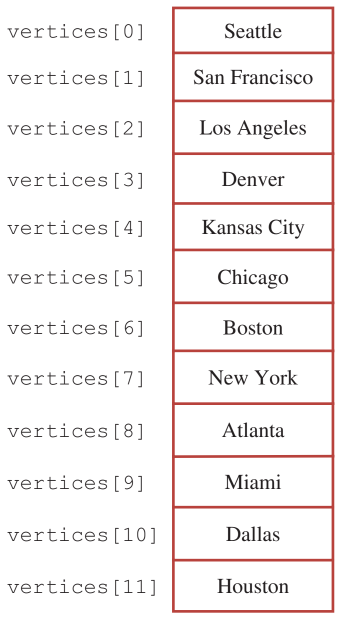


FIGURE 28.5 An array stores the vertex names.

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.

28.3.2 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 28.1 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 28.3a can be represented as follows:

edge array

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

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

28.3.3 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 in Listing 28.1.

LISTING 28.1 Edge.java

```
public class Edge {
  int u;
  int v;

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

  public boolean equals(Object o) {
    return u == ((Edge)o).u && v == ((Edge)o).v;
  }
}
```

For example, you can store all the edges in the graph in Figure 28.1 using the following list:

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

28.3.4 Representing Edges: Adjacency Matrices

adjacency matrix

Assume 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 example, the edges in the graph in Figure 28.1 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 28.3a 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
            };
```

28.3.5 Representing Edges: Adjacency Lists

adjacency vertex lists
adjacency edge lists

You can represent edges using *adjacency vertex lists* or *adjacency edge lists*. An adjacency vertex list for a vertex i contains the vertices that are adjacent to i and an adjacency edge list for a vertex i contains the edges that are adjacent to i. You may define an array of lists. The array has n entries, and each entry is a list. The adjacency vertex 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 28.1, you can create an array of lists as follows:

```
java.util.List<Integer>[] neighbors = new java.util.List[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 28.6.

To represent the adjacency edge lists for the graph in Figure 28.1, you can create an array of lists as follows:

```
java.util.List<Edge>[] neighbors = new java.util.List[12];
```

neighbors[0] contains all edges adjacent to vertex **0** (i.e., Seattle), **neighbors[1]** contains all edges adjacent to vertex **1** (i.e., San Francisco), and so on, as shown in Figure 28.7.

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

FIGURE 28.6 Edges in the graph in Figure 28.1 are represented using adjacency vertex lists.

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

FIGURE 28.7 Edges in the graph in Figure 28.1 are represented using adjacency edge lists.

Note

adjacency matrices vs. adjacency lists

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

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.

adjacency vertex lists vs. adjacency edge lists

Note

Adjacency vertex list is simpler for representing unweighted graphs. However, adjacency edge lists are more flexible for a wide range of graph applications. It is easy to add additional constraints on edges using adjacency edge lists. For this reason, this book will use adjacency edge lists to represent graphs.

using `ArrayList`

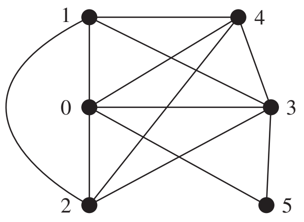
You can use arrays, array lists, or linked lists to store adjacency lists. We will use lists instead of arrays, because the lists are easily expandable to enable you to add new vertices. Further we will use array lists instead of linked lists, because our algorithms only require searching for adjacent vertices in the list. Using array lists is more efficient for our algorithms. Using array lists, the adjacency edge list in Figure 28.6 can be built as follows:

```
List<ArrayList<Edge>> neighbors = new ArrayList<>();
neighbors.add(new ArrayList<Edge>());
neighbors.get(0).add(new Edge(0, 1));
neighbors.get(0).add(new Edge(0, 3));
neighbors.get(0).add(new Edge(0, 5));
neighbors.add(new ArrayList<Edge>());
neighbors.get(1).add(new Edge(1, 0));
neighbors.get(1).add(new Edge(1, 2));
neighbors.get(1).add(new Edge(1, 3));
...
...
neighbors.get(11).add(new Edge(11, 8));
neighbors.get(11).add(new Edge(11, 9));
neighbors.get(11).add(new Edge(11, 10));
```

Check Point

28.3.1 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?

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



28.4 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 20.1. This design pattern is useful for modeling graphs. We will define an interface named `Graph` that contains all the common operations of graphs. 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 28.8.

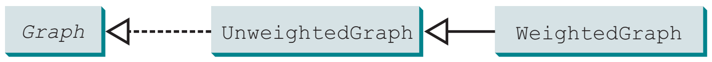
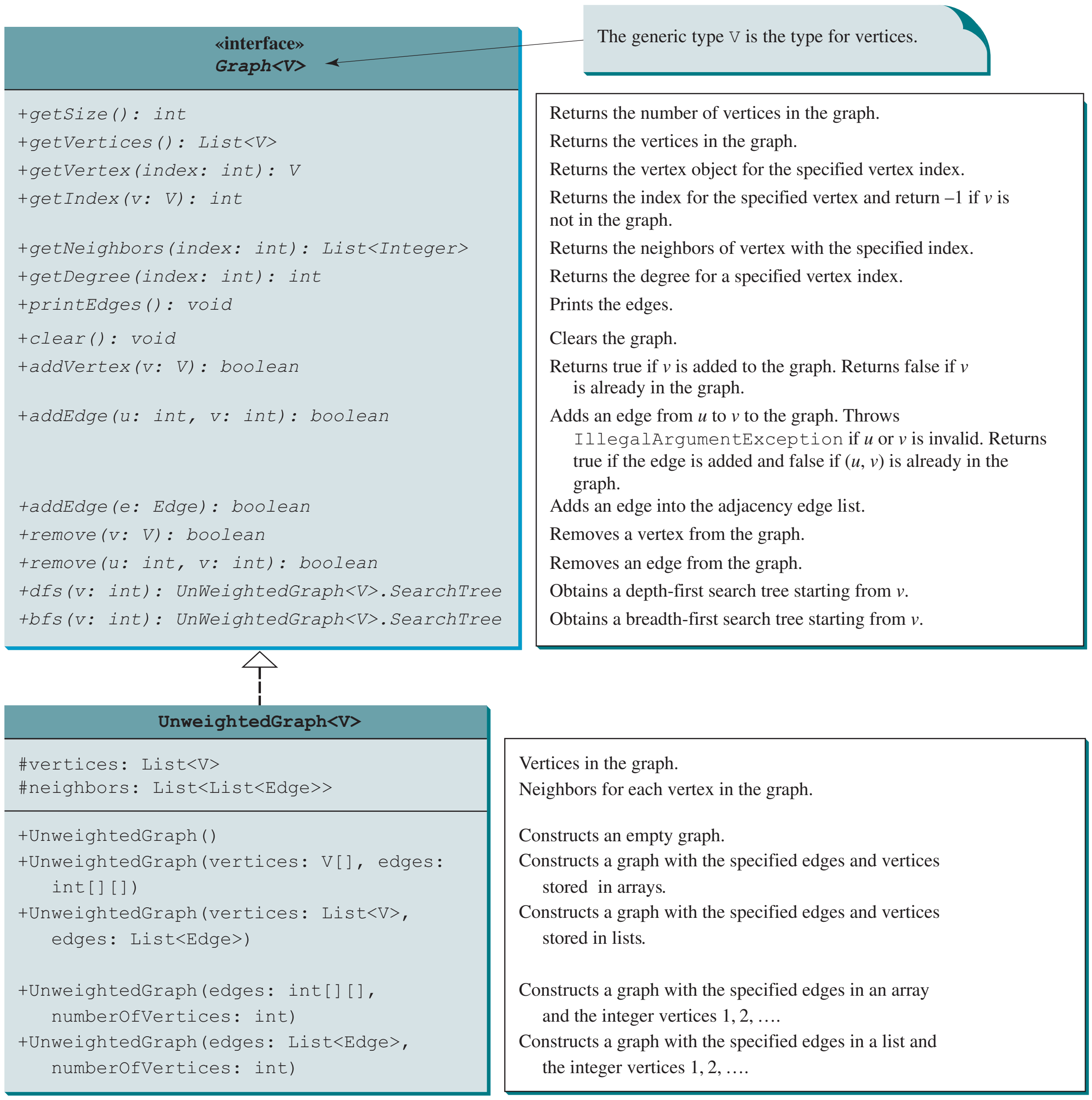


FIGURE 28.8 Common operations of graphs are defined in the interface and concrete classes define concrete graphs.

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 28.9 illustrates these methods in the UML diagram.

UnweightedGraph does not introduce any new methods. A list of vertices and an edge adjacency list are defined in the class. With these data fields, it is sufficient to implement all the methods defined in the **Graph** interface. For convenience, we assume the graph is a simple graph, that is, a vertex has no edge to itself and there are no parallel edges from vertex *u* to *v*.



«interface» *Graph<V>*	
+getSize(): int	Returns the number of vertices in the graph.
+getVertices(): List<V>	Returns the vertices in the graph.
+getVertex(index: int): V	Returns the vertex object for the specified vertex index.
+getIndex(v: V): int	Returns the index for the specified vertex and return –1 if *v* is not in the graph.
+getNeighbors(index: int): List<Integer>	Returns the neighbors of vertex with the specified index.
+getDegree(index: int): int	Returns the degree for a specified vertex index.
+printEdges(): void	Prints the edges.
+clear(): void	Clears the graph.
+addVertex(v: V): boolean	Returns true if *v* is added to the graph. Returns false if *v* is already in the graph.
+addEdge(u: int, v: int): boolean	Adds an edge from *u* to *v* to the graph. Throws `IllegalArgumentException` if *u* or *v* is invalid. Returns true if the edge is added and false if (*u*, *v*) is already in the graph.
+addEdge(e: Edge): boolean	Adds an edge into the adjacency edge list.
+remove(v: V): boolean	Removes a vertex from the graph.
+remove(u: int, v: int): boolean	Removes an edge from the graph.
+dfs(v: int): UnWeightedGraph<V>.SearchTree	Obtains a depth-first search tree starting from *v*.
+bfs(v: int): UnWeightedGraph<V>.SearchTree	Obtains a breadth-first search tree starting from *v*.

UnweightedGraph<V>	
#vertices: List<V>	Vertices in the graph.
#neighbors: List<List<Edge>>	Neighbors for each vertex in the graph.
+UnweightedGraph()	Constructs an empty graph.
+UnweightedGraph(vertices: V[], edges: int[][])	Constructs a graph with the specified edges and vertices stored in arrays.
+UnweightedGraph(vertices: List<V>, edges: List<Edge>)	Constructs a graph with the specified edges and vertices stored in lists.
+UnweightedGraph(edges: int[][], numberOfVertices: int)	Constructs a graph with the specified edges in an array and the integer vertices 1, 2, ...
+UnweightedGraph(edges: List<Edge>, numberOfVertices: int)	Constructs a graph with the specified edges in a list and the integer vertices 1, 2, ...

FIGURE 28.9 The **Graph** interface defines the common operations for all types of graphs.

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.

Assume the **Graph** interface and the **UnweightedGraph** class are available. Listing 28.2 gives a test program that creates the graph in Figure 28.1 and another graph for the one in Figure 28.3a.

Listing 28.2 TestGraph.java

vertices · edges · create a graph · number of vertices · get vertex · get index · print edges · list of Edge objects · create a graph · print edges

```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 28.1
    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<>(vertices, edges);
    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 28.3a
    String[] names = {"Peter", "Jane", "Mark", "Cindy", "Wendy"};
    java.util.ArrayList<Edge> edgeList
      = new java.util.ArrayList<>();
    edgeList.add(new Edge(0, 2));
    edgeList.add(new Edge(1, 2));
    edgeList.add(new Edge(2, 4));
    edgeList.add(new Edge(3, 4));
    // Create a graph with 5 vertices
    Graph<String> graph2 = new UnweightedGraph<>
      (java.util.Arrays.asList(names), edgeList);
    System.out.println("\nThe number of vertices in graph2: "
      + graph2.getSize());
    System.out.println("The edges for graph2:");
    graph2.printEdges();
  }
}
```

```
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 28.1 in lines 3–23. 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 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 23. Line 31 invokes the **printEdges()** method on **graph1** to display all edges in **graph1**.

The program creates **graph2** for the graph in Figure 28.3a in lines 34–43. The edges for **graph2** are defined in lines 37–40. **graph2** is created using a list of **Edge** objects in line 43. Line 47 invokes the **printEdges()** method on **graph2** to display all edges in **graph2**.

Note 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 28.3 and Listings 28.4 give the **Graph** interface and the **UnweightedGraph** class, respectively.

LISTING 28.3 Graph.java

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

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

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

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

getIndex
getNeighbors
getDegree
printEdges
clear
addVertex
addEdge
addEdge
remove vertex
remove edge
dfs
bfs

```
  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 the graph */
  public void clear();

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

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

  /** Add an edge to the graph */
  public boolean addEdge(Edge e);

  /** Remove a vertex v from the graph, return true if successful */
  public boolean remove(V v);

  /** Remove an edge (u, v) from the graph, return true if successful */
  public boolean remove(int u, int v);

  /** Obtain a depth-first search tree */
  public UnweightedGraph<V>.SearchTree dfs(int v);

  /** Obtain a breadth-first search tree */
  public UnweightedGraph<V>.SearchTree bfs(int v);
}
```

LISTING 28.4 UnweightedGraph.java

no-arg constructor
constructor
constructor

```
import java.util.*;

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

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

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

    createAdjacencyLists(edges, vertices.length);
  }

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

```
      addVertex(vertices.get(i));

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

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

    createAdjacencyLists(edges, numberOfVertices);
  }

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

    createAdjacencyLists(edges, numberOfVertices);
  }

  /** Create adjacency lists for each vertex */
  private void createAdjacencyLists(
      int[][] edges, int numberOfVertices) {
    for (int i = 0; i < edges.length; i++) {
      addEdge(edges[i][0], edges[i][1]);
    }
  }

  /** Create adjacency lists for each vertex */
  private void createAdjacencyLists(
      List<Edge> edges, int numberOfVertices) {
    for (Edge edge: edges) {
      addEdge(edge.u, 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 */
  public V getVertex(int index) {                                           getVertex
    return vertices.get(index);
  }

  @Override /** Return the index for the specified vertex object */
  public int getIndex(V v) {                                                getIndex
    return vertices.indexOf(v);
  }

  @Override /** Return the neighbors of the specified vertex */
  public List<Integer> getNeighbors(int index) {                            getNeighbors
    List<Integer> result = new ArrayList<>();
```

```java
    for (Edge e: neighbors.get(index))
      result.add(e.v);

    return result;
  }

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

getDegree

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

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

printEdges

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

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

clear

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

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

addVertex

```java
  public boolean addVertex(V vertex) {
    if (!vertices.contains(vertex)) {
      vertices.add(vertex);
      neighbors.add(new ArrayList<Edge>());
      return true;
    }
    else {
      return false;
    }
  }

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

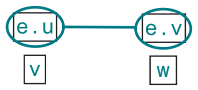
addEdge

```java
  public boolean addEdge(Edge e) {
    if (e.u < 0 || e.u > getSize() - 1)
      throw new IllegalArgumentException("No such index: " + e.u);

    if (e.v < 0 || e.v > getSize() -  1)
      throw new IllegalArgumentException("No such index: " + e.v);

    if (!neighbors.get(e.u).contains(e)) {
      neighbors.get(e.u).add(e);
      return true;
    }
    else {
      return false;
    }
  }

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

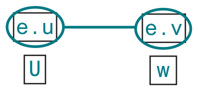
addEdge overloaded

```java
  public boolean addEdge(int u, int v) {
    return addEdge(new Edge(u, v));
  }

  @Override /** Obtain a DFS tree starting from vertex v */
  /** To be discussed in Section 28.7 */
  public SearchTree dfs(int v) {
    List<Integer> searchOrder = new ArrayList<>();
    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 SearchTree(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 (Edge e : neighbors.get(v)) {// e.u is v
      int w = e.v; // e.v is w in Listing 28.8
      if (!isVisited[w]) {
        parent[w] = v; // The parent of vertex w is v
        dfs(w, parent, searchOrder, isVisited); // Recursive search
      }
    }
  }

  @Override /** Starting bfs search from vertex v */
  /** To be discussed in Section 28.9 */
  public SearchTree bfs(int v) {
    List<Integer> searchOrder = new ArrayList<>();
    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<>(); // 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 (Edge e: neighbors.get(u)) {// Note that e.u is u
        int w = e.v; // e.v is w in Listing 28.8
        if (!isVisited[w]) {// e.v is w in Listing 28.11
          queue.offer(w); // Enqueue w
          parent[w] = u; // The parent of w is u
          isVisited[w] = true; // Mark it visited
        }
      }
```

dfs method



bfs method



SearchTree inner class

```
    }

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

  /** Tree inner class inside the UnweightedGraph class */
  /** To be discussed in Section 28.6 */
  public class SearchTree {
    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 SearchTree(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<>();

      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--)
        System.out.print(path.get(i) + " ");
    }
```

```
266
267       /** Print the whole tree */
268       public void printTree() {
269         System.out.println("Root is: " + vertices.get(root));
270         System.out.print("Edges: ");
271         for (int i = 0; i < parent.length; i++) {
272           if (parent[i] != -1) {
273             // Display an edge
274             System.out.print("(" + vertices.get(parent[i]) + ", " +
275               vertices.get(i) + ") ");
276           }
277         }
278         System.out.println();
279       }
280     }
281
282     @Override /** Remove vertex v and return true if successful */
283     public boolean remove(V v) {
284       return true; // Implementation left as an exercise
285     }
286
287     @Override /** Remove edge (u, v) and return true if successful */
288     public boolean remove(int u, int v) {
289       return true; // Implementation left as an exercise
290     }
291   }
```

The code in the `Graph` interface in Listing 28.3 is straightforward. Let us digest the code in the `UnweightedGraph` class in Listing 28.4.

The `UnweightedGraph` class defines the data field `vertices` (line 4) to store vertices and `neighbors` (line 5) to store edges in adjacency `edges` lists. `neighbors.get(i)` stores all edges 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 45–50). The `createAdjacencyLists (List<Edge> edges, int numberOfVertices)` method creates adjacency lists from edges in a list (lines 53–58).

The `getNeighbors(u)` method (lines 81–87) returns a list of vertices adjacent to vertex `u`. The `clear()` method (lines 106–110) removes all vertices and edges from the graph. The `addVertex(u)` method (lines 112–122) adds a new vertex to `vertices` and returns true. It returns false if the vertex is already in the graph (line 120).

The `addEdge(e)` method (lines 124–139) adds a new edge to the adjacency edge list and returns true. It returns false if the edge is already in the graph. This method may throw `IllegalArgumentExcepiton` if the edge is invalid (lines 126–130).

The `addEdge(u, v)` method (lines 141–144) adds an edge (`u`, `v`) to the graph. If a graph is undirected, you should invoke `addEdge(u, v)` and `addEdge(v, u)` to add an edge between `u` and `v`.

The `printEdges()` method (lines 95–104) displays all vertices and edges adjacent to each vertex.

The code in lines 148–208 gives the methods for finding a depth-first search tree and a breadth-first search tree, which will be introduced in Sections 28.7 and 28.9, respectively.

Check Point

28.4.1 Describe the methods in `Graph` and `UnweightedGraph`.

28.4.2 For the code in Listing 28.2, TestGraph.java, what is `graph1.getIndex("Seattle")`? What is `graph1.getDegree(5)`? What is `graph1.getVertex(4)`?

28.4.3 Show the output of the following code:

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

28.4.4 What will `getIndex(v)` return if v is not in the graph? What happens to `getVertex(index)` if index is not in the graph? What happens to `addVertex(v)` if v is already in the graph? What happens to `addEdge(u, v)` if u or v is not in the graph?

28.5 Graph Visualization

Key Point

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

The preceding section introduced the `Graph` interface and the `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 28.5.

Listing 28.5 Displayable.java

`Displayable` interface

```
public interface Displayable {
  public double getX(); // Get x-coordinate of the vertex
  public double 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 pane named `GraphView`, as shown in Listing 28.6.

Listing 28.6 GraphView.java

extends `BorderPane`
`Displayable` vertices

```
import javafx.scene.Group;
import javafx.scene.layout.BorderPane;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Line;
import javafx.scene.text.Text;

public class GraphView extends BorderPane {
  private Graph<? extends Displayable> graph;
  private Group group = new Group();

  public GraphView(Graph<? extends Displayable> graph) {
    this.graph = graph;
    this.setCenter(group); // Center the group
14       repaintGraph();
15     }
16
17     private void repaintGraph() {
18       group.getChildren().clear(); // Clear group for a new display
19
20       // Draw vertices and text for vertices
21       java.util.List<? extends Displayable> vertices
22         = graph.getVertices();
23       for (int i = 0; i < graph.getSize(); i++) {
24         double x = vertices.get(i).getX();
25         double y = vertices.get(i).getY();
26         String name = vertices.get(i).getName();
27
28         group.getChildren().add(new Circle(x, y, 16));
29         group.getChildren().add(new Text(x - 8, y - 18, name));
30       }
31
32       // Draw edges for pairs of vertices
33       for (int i = 0; i < graph.getSize(); i++) {
34         java.util.List<Integer> neighbors = graph.getNeighbors(i);
35         double x1 = graph.getVertex(i).getX();
36         double y1 = graph.getVertex(i).getY();
37         for (int v: neighbors) {
38           double x2 = graph.getVertex(v).getX();
39           double y2 = graph.getVertex(v).getY();
40
41           // Draw an edge for (i, v)
42           group.getChildren().add(new Line(x1, y1, x2, y2));
43         }
44       }
45     }
46   }
```

display a vertex

display a text

draw an edge

To display a graph on a pane, simply create an instance of `GraphView` by passing the graph as an argument in the constructor (line 11). The class for the graph's vertex must implement the `Displayable` interface in order to display the vertices (lines 21–44). For each vertex index `i`, invoking `graph.getNeighbors(i)` returns its adjacency list (line 34). 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 35–42).

Listing 28.7 gives an example of displaying the graph in Figure 28.1, as shown in Figure 28.10.

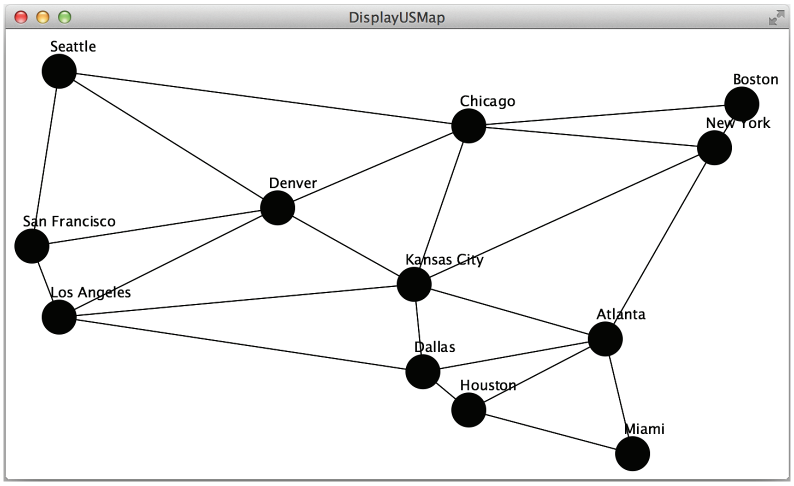


FIGURE 28.10 The graph is displayed in the pane.

LISTING 28.7 DisplayUSMap.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class DisplayUSMap extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    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 28.1
    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<City> graph = new UnweightedGraph<>(vertices, edges);

    // Create a scene and place it in the stage
    Scene scene = new Scene(new GraphView(graph), 750, 450);
    primaryStage.setTitle("DisplayUSMap"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

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

    City(String name, double x, double y) {
      this.name = name;
      this.x = x;
      this.y = y;
    }

    @Override
    public double getX() {
      return x;
    }

    @Override
    public double getY() {
      return y;
    }

```

create a graph

create a GraphView

City class

```
59      @Override
60      public String getName() {
61        return name;
62      }
63    }
64  }
```

The class `City` is defined to model the vertices with their coordinates and names (lines 39–63). The program creates a graph with the vertices of the `City` type (line 30). Since `City` implements `Displayable`, a `GraphView` object created for the graph displays the graph in the pane (line 33).

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

28.5.1 Will Listing 28.7, DisplayUSMap.java work, if the code in lines 38–42 in Listing 28.6, GraphView.java is replaced by the following code?

```
if (i < v) {
  double x2 = graph.getVertex(v).getX();
  double y2 = graph.getVertex(v).getY();

  // Draw an edge for (i, v)
  getChildren().add(new Line(x1, y1, x2, y2));
}
```

28.5.2 For the `graph1` object created in Listing 28.1, TestGraph.java, can you create a `GraphView` object as follows?

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

28.6 Graph Traversals

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

Key Point

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 28.11. Note `SearchTree` is an inner class defined in the `UnweightedGraph` class. `UnweightedGraph<V>.SearchTree` is different from the `Tree` interface defined in Section 25.2.5. `UnweightedGraph<V>.SearchTree` 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, `UnweightedGraph<V>.SearchTree` is not defined as a subtype of `Tree`.

The `SearchTree` class is defined as an inner class in the `UnweightedGraph` class in lines 210–278 in Listing 28.4. The constructor creates a tree with the root, edges, and a search order.

The `SearchTree` 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 28.7 and 28.9 will introduce depth-first search and breadth-first search, respectively. Both searches will result in an instance of the `SearchTree` class.

Check Point

28.6.1 Does `UnweightedGraph<V>.SearchTree` implement the `Tree` interface defined in Listing 25.3, Tree.java?

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

UnweightedGraph<V>.SearchTree	
-root: int	The root of the tree.
-parent: int[]	The parents of the vertices.
-searchOrder: List<Integer>	The orders for traversing the vertices.
+SearchTree(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 28.11 The SearchTree class describes the nodes with parent–child relationships.

28.7 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 25.2.4, 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 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 28.4 for finding connected components in a graph.

28.7.1 Depth-First Search Algorithm

The algorithm for the depth-first search is described in Listing 28.8.

LISTING 28.8 Depth-First Search Algorithm

```
Input: G = (V, E) and a starting vertex v
Output: a DFS tree rooted at v

SearchTree dfs(vertex v) {
  visit v;
  for each neighbor w of v
    if (w has not been visited) {
      set v as the parent for w in the tree;
      dfs(w);
    }
}
```

visit v

check a neighbor

set a parent in the tree

recursive search

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 28.12a. Suppose we 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 28.12b. 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 28.12c. 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 28.12d. 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 28.12e. 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.

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.

DFS time complexity

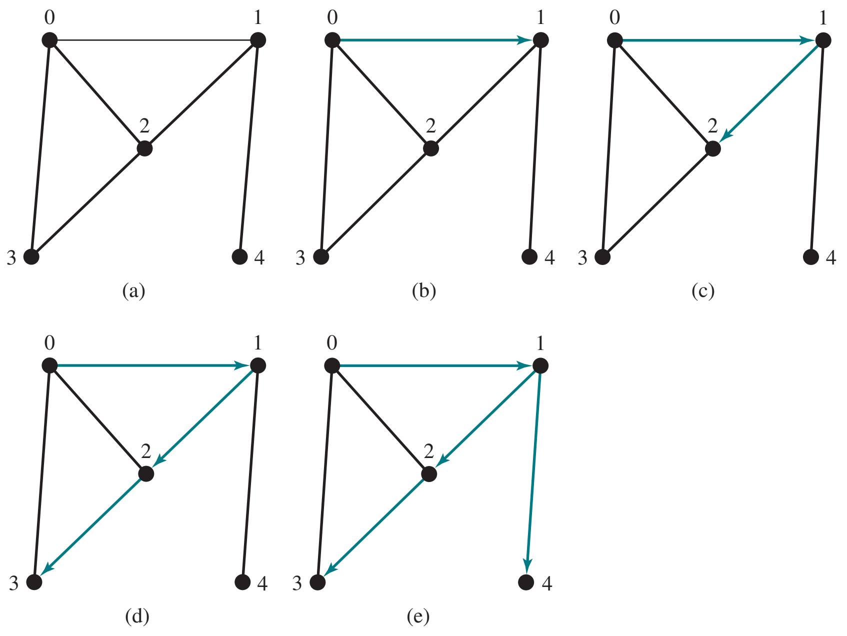


FIGURE 28.12 Depth-first search visits a node and its neighbors recursively.

28.7.2 Implementation of Depth-First Search

The algorithm for DFS in Listing 28.8 uses recursion. It is natural to use recursion to implement it. Alternatively, you can use a stack (see Programming Exercise 28.3).

The `dfs(int v)` method is implemented in lines 148–178 in Listing 28.4. It returns an instance of the `SearchTree` class with vertex `v` as the root. The method stores the vertices searched in the list `searchOrder` (line 150), the parent of each vertex in the array `parent` (line 150), and uses the `isVisited` array to indicate whether a vertex has been visited (line 155). It invokes the helper method `dfs(v, parent, searchOrder, isVisited)` to perform a depth-first search (line 159).

In the recursive helper method, the search starts from vertex `v`. `v` is added to `searchOrder` in line 169 and is marked as visited (line 170). For each unvisited neighbor of `v`, the method is recursively invoked to perform a depth-first search. When a vertex `w` (`w` is `e.v` in line 172 in

Listing 28.4) is visited, the parent of **w** is stored in **parent[w]** (line 174). The method returns when all vertices are visited for a connected graph, or in a connected component.

U.S. Map Search

Listing 28.9 gives a test program that displays a DFS for the graph in Figure 28.1 starting from Chicago. The graphical illustration of the DFS starting from Chicago is shown in Figure 28.13.

Listing 28.9 TestDFS.java

vertices

edges

create a graph

get DFS

get search order

```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},
      {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<>(vertices, edges);
    UnweightedGraph<String>.SearchTree dfs =
      graph.dfs(graph.getIndex("Chicago"));

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

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

```
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
```

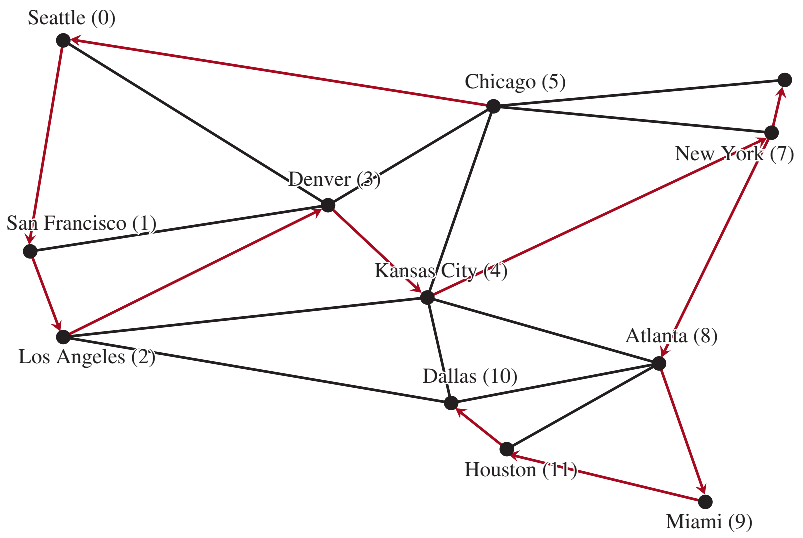


FIGURE 28.13 A DFS search starts from Chicago. *Source*: © Mozilla Firefox.

28.7.3 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 28.1.)
- Detecting whether there is a path between two vertices (see Programming Exercise 28.5).
- Finding a path between two vertices (see Programming Exercise 28.5).
- 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 28.4).
- Detecting whether there is a cycle in the graph (see Programming Exercise 28.6).
- Finding a cycle in the graph (see Programming Exercise 28.7).
- 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 28.17).

The first six problems can be easily solved using the `dfs` method in Listing 28.4. 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.E on the Companion Website.

Check Point

28.7.1 What is depth-first search?

28.7.2 Draw a DFS tree for the graph in Figure 28.3b starting from node `A`.

28.7.3 Draw a DFS tree for the graph in Figure 28.1 starting from vertex `Atlanta`.

28.7.4 What is the return type from invoking `dfs(v)`?

28.7.5 The depth-first search algorithm described in Listing 28.8 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
SearchTree 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;
  }
}
```

28.8 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 28.14a. Otherwise, they are not filled, as shown in Figure 28.14b.

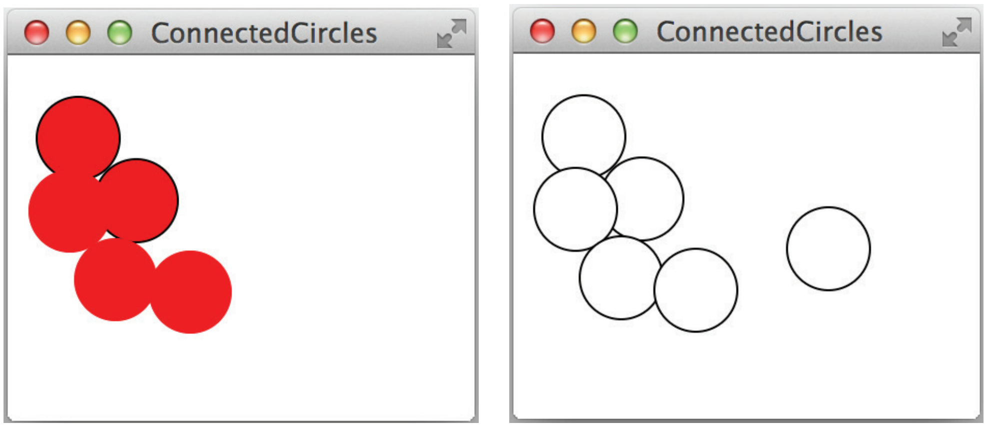


Figure 28.14 You can apply DFS to determine whether the circles are connected.
Source: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

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 28.10.

LISTING 28.10 ConnectedCircles.java

```
import javafx.application.Application;
import javafx.geometry.Point2D;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class ConnectedCircles extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a scene and place it in the stage
    Scene scene = new Scene(new CirclePane(), 450, 350);               create a circle pane
    primaryStage.setTitle("ConnectedCircles"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  /** Pane for displaying circles */
  class CirclePane extends Pane {                                      pane for showing circles
    public CirclePane() {
      this.setOnMouseClicked(e -> {                                    handle mouse clicked
        if (!isInsideACircle(new Point2D(e.getX(), e.getY()))) {       is it inside another circle?
          // Add a new circle
          getChildren().add(new Circle(e.getX(), e.getY(), 20));       add a new circle
          colorIfConnected();                                          color if all connected
        }
      });
    }

    /** Returns true if the point is inside an existing circle */
    private boolean isInsideACircle(Point2D p) {
      for (Node circle: this.getChildren())
        if (circle.contains(p))                                        contains the point?
          return true;

      return false;
    }

    /** Color all circles if they are connected */
    private void colorIfConnected() {
      if (getChildren().size() == 0)
        return; // No circles in the pane

      // Build the edges
      java.util.List<Edge> edges                                       create edges
        = new java.util.ArrayList<>();
      for (int i = 0; i < getChildren().size(); i++)
        for (int j = i + 1; j < getChildren().size(); j++)
          if (overlaps((Circle)(getChildren().get(i)),
              (Circle)(getChildren().get(j)))) {
            edges.add(new Edge(i, j));
            edges.add(new Edge(j, i));
          }

      // Create a graph with circles as vertices
```

create a graph

get a search tree

connected?

connected

not connected

two circles overlap?

```
58        Graph<Node> graph = new UnweightedGraph<>
59          ((java.util.List<Node>)getChildren(), edges);
60        UnweightedGraph<Node>.SearchTree tree = graph.dfs(0);
61        boolean isAllCirclesConnected = getChildren().size() == tree
62          .getNumberOfVerticesFound();
63
64        for (Node circle: getChildren()) {
65          if (isAllCirclesConnected) { // All circles are connected
66            ((Circle)circle).setFill(Color.RED);
67          }
68          else {
69            ((Circle)circle).setStroke(Color.BLACK);
70            ((Circle)circle).setFill(Color.WHITE);
71          }
72        }
73      }
74    }
75
76    public static boolean overlaps(Circle circle1, Circle circle2) {
77      return new Point2D(circle1.getCenterX(), circle1.getCenterY()).
78        distance(circle2.getCenterX(), circle2.getCenterY())
79        <= circle1.getRadius() + circle2.getRadius();
80    }
81  }
```

The JavaFX `Circle` class contains the data fields `x`, `y`, and `radius`, which specify the circle's center location and radius. It also defines the `contains` method for testing whether a point is inside the circle. The `overlaps` method (lines 76–80) checks 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 26).

To detect whether the circles are connected, the program constructs a graph (lines 46–59). The circles are the vertices of the graph. The edges are constructed in lines 47–55. Two circle vertices are connected if they overlap (line 51). The DFS of the graph results in a tree (line 60). The tree's `getNumberOfVerticesFound()` returns the number of vertices searched. If it is equal to the number of circles, all circles are connected (lines 61–62).

Check Point

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

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

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

28.9 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 25.2.4, 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 each vertex is visited only once, it skips a vertex if it has already been visited.

28.9.1 Breadth-First Search Algorithm

The algorithm for the breadth-first search starting from vertex *v* in a graph is described in Listing 28.11.

LISTING 28.11 Breadth-First Search Algorithm

```
Input: G = (V, E) and a starting vertex v
Output: a BFS tree rooted at v

SearchTree 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;
        set u as the parent for w in the tree;
        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 28.15a. 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 28.15b. 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 28.15c. 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.

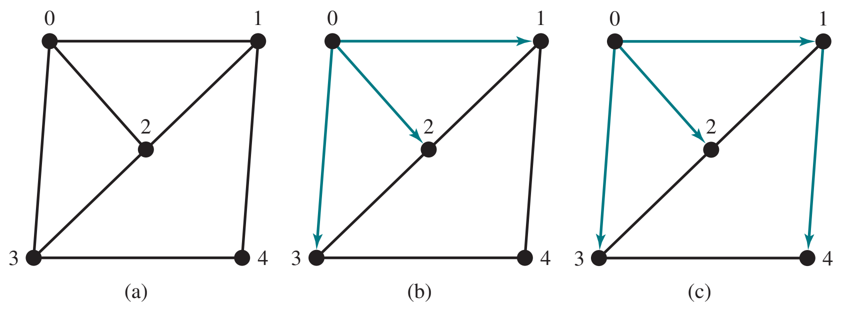


FIGURE 28.15 Breadth-first search visits a node, then its neighbors, then its neighbors's neighbors, and so on.

28.9.2 Implementation of Breadth-First Search

The `bfs(int v)` method is defined in the `Graph` interface and implemented in the `UnweightedGraph` class in Listing 28.4 (lines 182–208). It returns an instance of the `SearchTree` class with vertex `v` as the root. The method stores the vertices searched in the list `searchOrder` (line 183), the parent of each vertex in the array `parent` (line 184), uses a linked list for a queue (lines 188–189), and uses the `isVisited` array to indicate whether a vertex has been visited (line 190). The search starts from vertex `v`. `v` is added to the queue in line 191 and is marked as visited (line 192). The method now examines each vertex `u` in the queue (line 194) and adds it to `searchOrder` (line 196). The method adds each unvisited neighbor `w` (`w` is `e.v` in line 198 in Listing 28.4) of `u` to the queue (line 200), sets its parent to `u` (line 201), and marks it as visited (line 202).

Listing 28.12 gives a test program that displays a BFS for the graph in Figure 28.1 starting from Chicago. The graphical illustration of the BFS starting from Chicago is shown in Figure 28.16.

LISTING 28.12 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<>(vertices, edges);
    UnweightedGraph<String>.SearchTree bfs =
      graph.bfs(graph.getIndex("Chicago"));

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

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

```
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
```

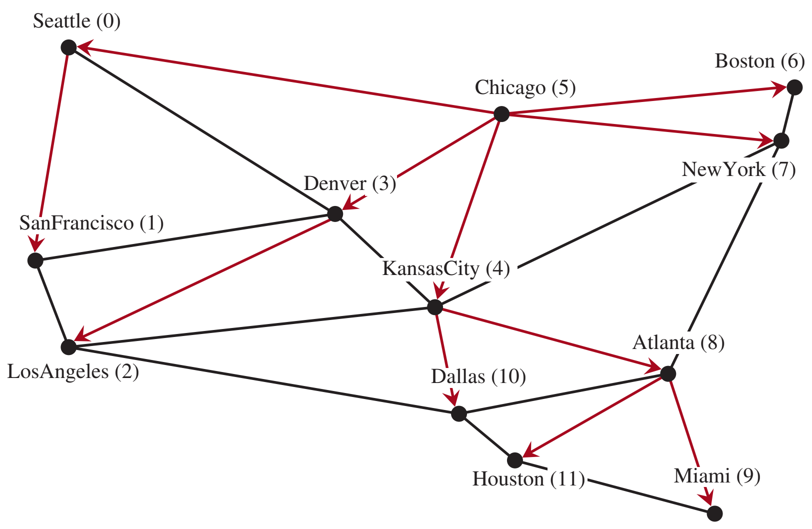


FIGURE 28.16 BFS search starts from Chicago. *Source*: © Mozilla Firefox.

28.9.3 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 CheckPoint Question 28.9.5.)
- 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 28.6).
- Finding a cycle in the graph (see Programming Exercise 28.7).
- 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 28.8.)

Check Point

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

28.9.2 What is breadth-first search?

28.9.3 Draw a BFS tree for the graph in Figure 28.3b starting from node `A`.

28.9.4 Draw a BFS tree for the graph in Figure 28.1 starting from vertex `Atlanta`.

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

28.10 Case Study: The Nine Tails Problem

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

Key Point

The nine tails problem is as follows. Nine coins are placed in a 3×3 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 28.17a. After you flip the second coin in the last row, the nine coins are now as shown in Figure 28.17b. After you flip the second coin in the first row, the nine coins are all face down, as shown in Figure 28.17c. See liveexample.pearsoncmg.com/dsanimation/NineCoin.html for an interactive demo.

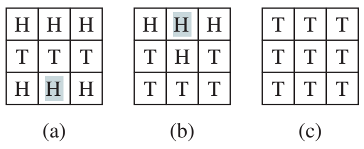


FIGURE 28.17 The problem is solved when all coins are face down.

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 28.17 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 28.18.

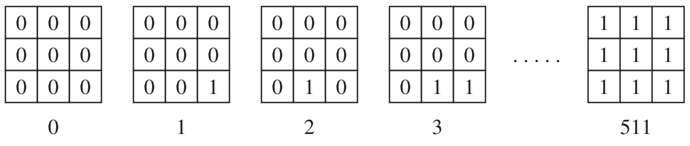


FIGURE 28.18 There are total of 512 nodes labeled in this order: **0**, **1**, **2**, . . . , **511**.

We assign an edge from node **v** to **u** if there is a legal move from **u** to **v**. Figure 28.19 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 28.18 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 directed 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 28.19.

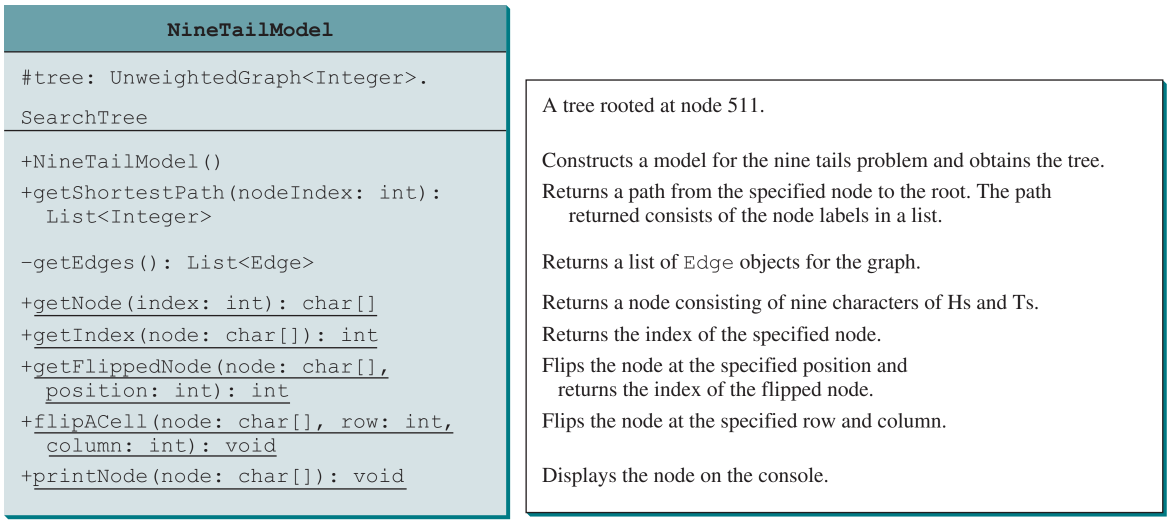

NineTailModel
#tree: UnweightedGraph<Integer>.SearchTree
+NineTailModel()
+getShortestPath(nodeIndex: int): List<Integer>
-getEdges(): List<Edge>
+getNode(index: int): char[]
+getIndex(node: char[]): int
+getFlippedNode(node: char[], position: int): int
+flipACell(node: char[], row: int, column: int): void
+printNode(node: char[]): void
A tree rooted at node 511.
Constructs a model for the nine tails problem and obtains the tree.
Returns a path from the specified node to the root. The path returned consists of the node labels in a list.
Returns a list of Edge objects for the graph.
Returns a node consisting of nine characters of Hs and Ts.
Returns the index of the specified node.
Flips the node at the specified position and returns the index of the flipped node.
Flips the node at the specified row and column.
Displays the node on the console.

FIGURE 28.19 The NineTailModel class models the nine tails problem using a graph.

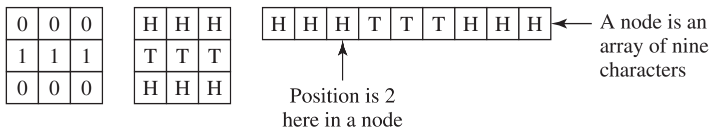
Visually, a node is represented in a 3 × 3 matrix with the letters H and T. In our program, we use a single-dimensional array of nine characters to represent a node. For example, the node for vertex 1 in Figure 28.18 is represented as {'H', 'H', 'H', 'H', 'H', 'H', 'H', 'H', 'T'} in the array.

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 Hs. getNode(511) returns the node that contains nine Ts. The getIndex(node) method returns the index of the node.

Note the data field tree is defined as protected so 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. The position is a value from 0 to 8, which points to a coin in the node, as shown in the following figure.


0 0 0
1 1 1
0 0 0
H H H
T T T
H H H
H H H T T T H H H
A node is an array of nine characters
Position is 2 here in a node

For example, for node 56 in Figure 28.20, 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.

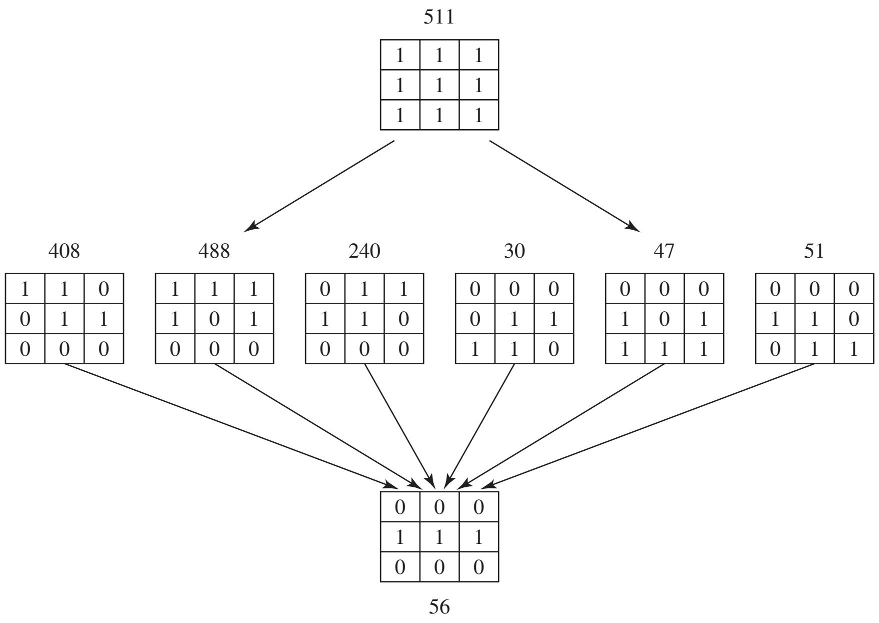


FIGURE 28.20 If node **u** becomes node **v** after cells are flipped, assign an edge from **v** to **u**.

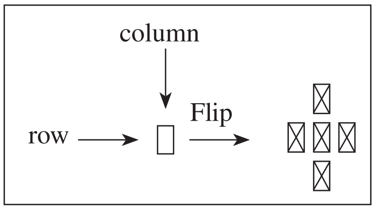
Listing 28.13 shows the source code for NineTailModel.java.

LISTING 28.13 `NineTailModel.java`

```
import java.util.*;

public class NineTailModel {
  public final static int NUMBER_OF_NODES = 512;
  protected UnweightedGraph<Integer>.SearchTree tree;     // declare a tree

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

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

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

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

  public static int getIndex(char[] node) {
    int result = 0;

    for (int i = 0; i < 9; i++)
      if (node[i] == 'T')
        result = result * 2 + 1;
      else
        result = result * 2 + 0;

    return result;
  }

  public static char[] getNode(int index) {
    char[] result = new char[9];

    for (int i = 0; i < 9; i++) {
      int digit = index % 2;
      if (digit == 0)
        result[8 - i] = 'H';
      else
        result[8 - i] = 'T';
      index = index / 2;
    }

    return result;
  }

  public List<Integer> getShortestPath(int nodeIndex) {
```

add an edge

flip cells



flip a cell

get index for a node

For example:
index: 3
node: HHHHHHHTT

```
H H H
H H H
H T T
```

get node for an index

For example:
node: THHHHHHTT
index: 259

```
T H H
H H H
H T T
```

shortest path

display a node

```
 90      return tree.getPath(nodeIndex);
 91    }
 92
 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  }
```

For example:
node: THHHHHHTT
Output:

```
T H H
H H H
H T T
```

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 a 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 28.14 gives a program that prompts the user to enter an initial node and displays the steps to reach the target node.

LISTING 28.14 NineTail.java

initial node

create model

get shortest path

```
 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  }
```

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

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

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

28.10.3 What is returned after invoking `getIndex("HTHTTTHHH".toCharArray())` in Listing 28.13? What is returned after invoking `getNode(46)` in Listing 28.13?

28.10.4 If lines 26 and 27 are swapped in Listing 28.13, NineTailModel.java, will the program work? Why not?

Key Terms

adjacency list	1075	parallel edge	1070
adjacency matrix	1074	Seven Bridges of Königsberg	1068
adjacent vertices	1070	simple graph	1070
breadth-first search	1089	spanning tree	1070
complete graph	1070	strongly connected graph	1070
cycle	1070	tree	1070
degree	1070	undirected graph	1069
depth-first search	1089	unweighted graph	1070
directed graph	1069	weakly connected graph	1070
graph	1068	weighted graph	1070
incident edges	1070		

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.

Quiz

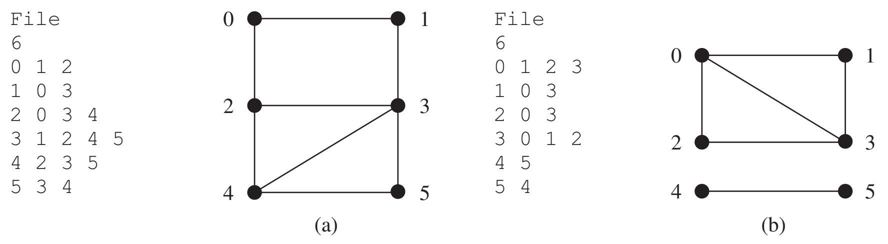
Answer the quiz for this chapter online at the book Companion Website.

Programming Exercises

MyProgrammingLab™

Sections 28.6–28.10

***28.1** (*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 28.21 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
```

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

FIGURE 28.21 The vertices and edges of a graph can be stored in a file.

Your program should prompt the user to enter a URL for 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 `UnweightedGraph<V>.SearchTree`. 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 URL:
https://liveexample.pearsoncmg.com/test/GraphSample1.txt
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 `Edge` objects. Use `new 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.)

***28.2** (*Create a file for a graph*) Modify Listing 28.2, TestGraph.java, to create a file representing `graph1`. The file format is described in Programming Exercise 28.1. Create the file from the array defined in lines 8–21 in Listing 28.2. 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
```

***28.3** (*Implement DFS using a stack*) The depth-first search algorithm described in Listing 28.8, Depth-First Search Algorithm, uses recursion. Design a new algorithm without using recursion. Describe it using pseudocode. Implement it by defining a new class named `UnweightedGraphWithNonrecursiveDFS` that extends `UnweightedGraph` and overriding the `dfs` method. Write a test program same as Listing 28.9, TestPFS.java, except that `UnweightedGraph` is replaced by `UnweightedGraphWithNonrecursiveDFS`.

***28.4** (*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 28.21b, `getConnectedComponents()` returns `[[0, 1, 2, 3], [4, 5]]`.

***28.5** (*Find paths*) Define a new class named `UnweightedGraphWithGetPath` that extends `UnweightedGraph` with a new method for finding 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`. Write a test program that creates a graph for Figure 28.1. The program prompts the user to enter two cities and displays their paths. Use https://liveexample.pearsoncmg.com/test/Exercise28_05.txt to test your code. Here is a sample run:

```
Enter a starting city: Seattle ↵Enter
Enter an ending city: Miami ↵Enter
The path is Seattle Denver Kansas City Atlanta Miami
```

***28.6** (*Detect cycles*) Define a new class named `UnweightedGraphDetectCycle` that extends `UnweightedGraph` with a new method for determining whether there is a cycle in the graph with the following header:

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

Describe the algorithm in pseudocode and implement it. Note the graph may be a directed graph.

***28.7** (*Find a cycle*) Define a new class named `UnweightedGraphFindCycle` that extends `UnweightedGraph` with a new method for finding a cycle starting at vertex `u` 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`. Describe the algorithm in pseudocode and implement it.

****28.8** (*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. Define a new class named `UnweightedGraphTestBipartite` with the following method to detect whether the graph is bipartite:

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

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

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

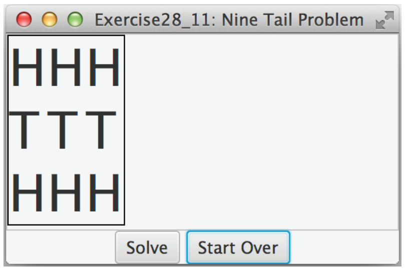
The method returns a `List` that contains two sublists, each of which contains a set of vertices. If the graph is not bipartite, the method returns `null`.

***28.10** (*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 Programming Exercise 28.1. 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 28.21a, the shortest path between `0` and `5` may be displayed as `0 1 3 5`.

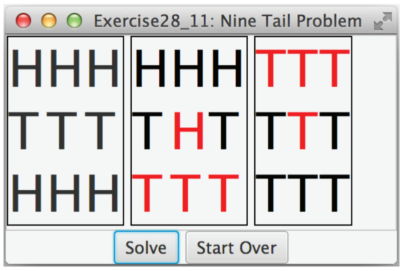
Here is a sample run of the program:

```
Enter a file name: c:\exercise\GraphSample1.txt ↵Enter
Enter two vertices (integer indexes): 0 5 ↵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 path is 0 1 3 5
```

****28.11** (*Revise Listing 28.14, NineTail.java*) The program in Listing 28.14 lets the user enter an input for the nine tails problem from the console and displays the result on the console. Write a program that lets the user set an initial state of the nine coins (see Figure 28.22a) and click the *Solve* button to display the solution, as shown in Figure 28.22b. Initially, the user can click the mouse button to flip a coin. Set a red color on the flipped cells.



(a)



(b)

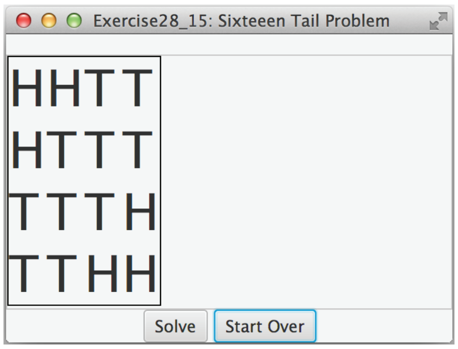
FIGURE 28.22 The program solves the nine tails problem. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

****28.12** (*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 all neighboring cells including the diagonal neighbors are also flipped.

****28.13** (*4 × 4 16 tails problem*) Listing 28.14, NineTail.java, presents a solution for the nine tails problem. Revise this program for the 4 × 4 16 tails problem. Note it is possible that a solution may not exist for a starting pattern. If so, report that no solution exists.

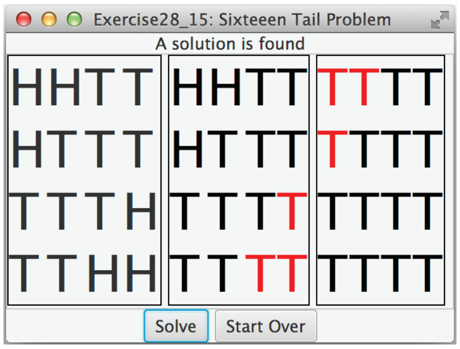
****28.14** (*4 × 4 16 tails analysis*) The nine tails problem in the text uses a 3 × 3 matrix. Assume you have 16 coins placed in a 4 × 4 matrix. Write a program to find out the number of the starting patterns that don't have a solution.

***28.15** (*4 × 4 16 tails GUI*) Rewrite Programming Exercise 28.14 to enable the user to set an initial pattern of the 4 × 4 16 tails problem (see Figure 28.23a). The user can click the *Solve* button to display the solution, as shown in Figure 28.23b. Initially, the user can click the mouse button to flip a coin. If a solution does not exist, display a message dialog to report it.



```
HHTT
HTTT
TTTH
TTHH
```

(a)



A solution is found

```
HHTT  HHTT  TTTT
HTTT  HTTT  TTTT
TTTH  TTTT  TTTT
TTHH  TTTT  TTTT
```

(b)

FIGURE 28.23 The problem solves the 16 tails problem. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

****28.16** (*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 <V> Graph<V> maxInducedSubgraph(Graph<V> g, int k)
```

The method returns an empty graph 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.)

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

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

*****28.18** (*Knight's Tour cycle*) Rewrite KnightTourApp.java in the case study in Supplement VI.E 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.

****28.19** (*Display a DFS/BFS tree in a graph*) Modify `GraphView` in Listing 28.6 to add a new data field `tree` with a setter method. The edges in the tree are displayed in red. Write a program that displays the graph in Figure 28.1 and the DFS/BFS tree starting from a specified city, as shown in Figures 28.13 and 28.16. If a city not in the map is entered, the program displays an error message in the label.

***28.20** (*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 28.24a 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 pane using `GraphView`, as shown in Figure 28.24b.

```
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
```

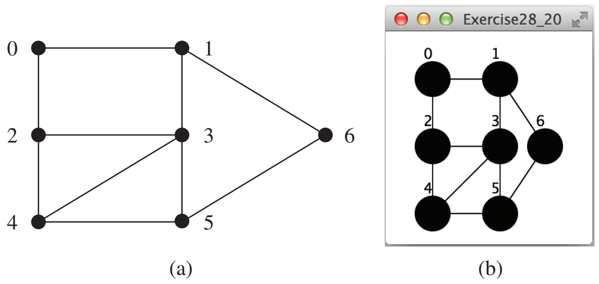


Figure 28.24 The program reads the information about the graph and displays it visually. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

****28.21** (*Display sets of connected circles*) Modify Listing 28.10, 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 28.25. (*Hint*: See Programming Exercise 28.4.)

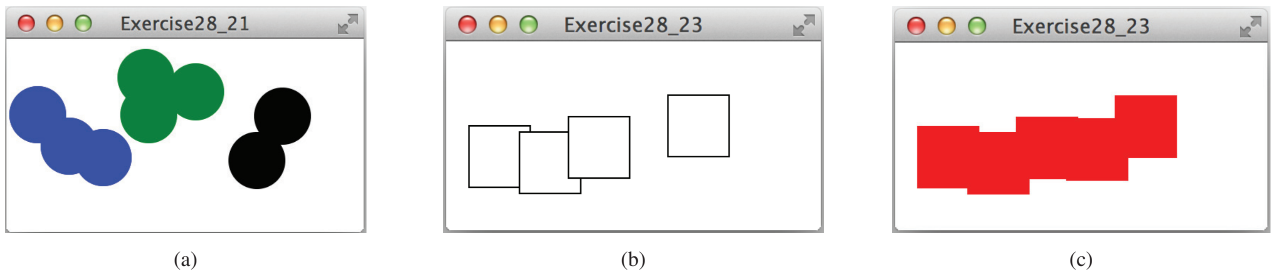


Figure 28.25 (a) Connected circles are displayed in the same color. (b) Rectangles are not filled with a color if they are not connected. (c) Rectangles are filled with a color if they are connected. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

***28.22** (*Move a circle*) Modify Listing 28.10, ConnectedCircles.java, to enable the user to drag and move a circle.

****28.23** (*Connected rectangles*) Listing 28.10, 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 28.25b–c.

***28.24** (*Remove a circle*) Modify Listing 28.10, ConnectedCircles.java, to enable the user to remove a circle when the mouse is clicked inside the circle.

***28.25** (*Implement remove(V v)*) Modify Listing 28.4, UnweightedGraph.java, to override the `remove(V v)` method defined in the `Graph` interface.

***28.26** (*Implement remove(int u, int v)*) Modify Listing 28.4, UnweightedGraph.java, to override the `remove(int u, int v)` method defined in the `Graph` interface.