Engineering Software Development in Java

**Lecture Notes for ENCE 688R,
Civil Information Systems**

Spring Semester, 2019

Mark Austin,
Department of Civil and Enviromental Engineering,
University of Maryland,
College Park,
Maryland 20742, U.S.A.

Contents

Chapter 13

Modeling Real-World Networks

13.1 Introduction

Networks occur in almost every area of engineering. In Civil and Environmental Engineering, for example, graphs can be used to represent infrastructure networks, transportation networks, and biological networks. Buildings can be modeled as networks of intertwined networks for the plumbing, HVAC, and electrical networks. Graphs can be used to model the connectivity relationship among spaces in a building [1, 5, 7].

Example 1. Modeling a House and Surrounding Grounds

Figure 13.1 shows a schematic for a house and its surrounding grounds.

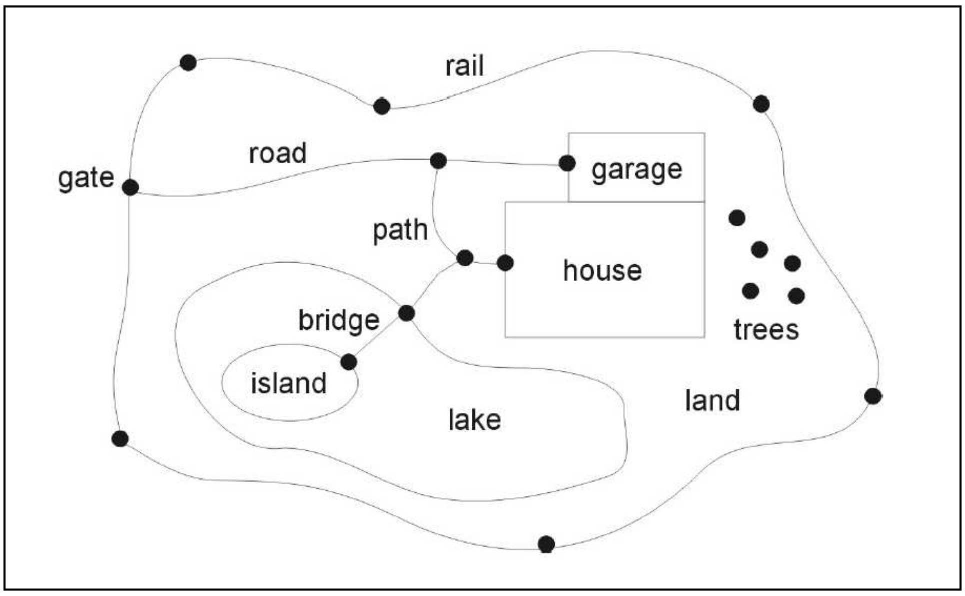


Figure 13.1. Modeling for a house and its surrounding grounds.

The exterior rail and pathways connecting the buildings and the lake can be viewed as simple networks. With thie network model in hand, an engineer might ask:

- What path should I take to get from the gate to the house?

- How is the path connected to the road and bridge?
- What is the shortest distance between the gate and the island?
- Is it possible to get to the trees without walking on the grass?
- Are there any loops in the road/path structure?

Example 2. Modeling Points, Lines and Regions

Points, lines and regions are fundamental abstractions for modeling single, self-contained objects:

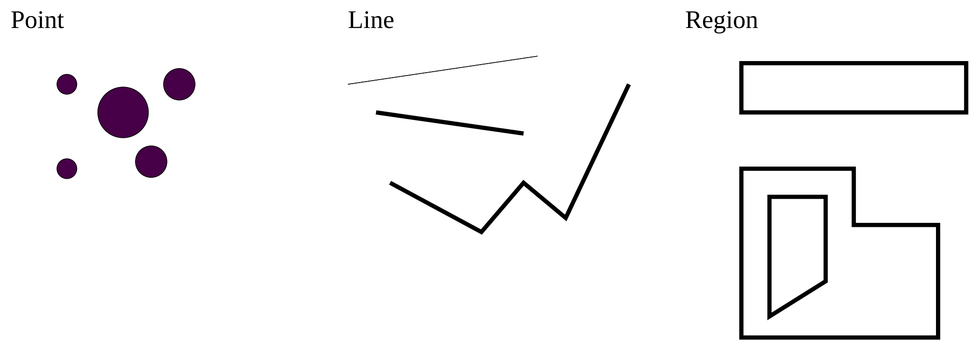


Figure 13.2. Points, lines and regions are fundamental spatial data types.

Partitions and networks are fundamental abstractions for modeling spatially related collections of objects.

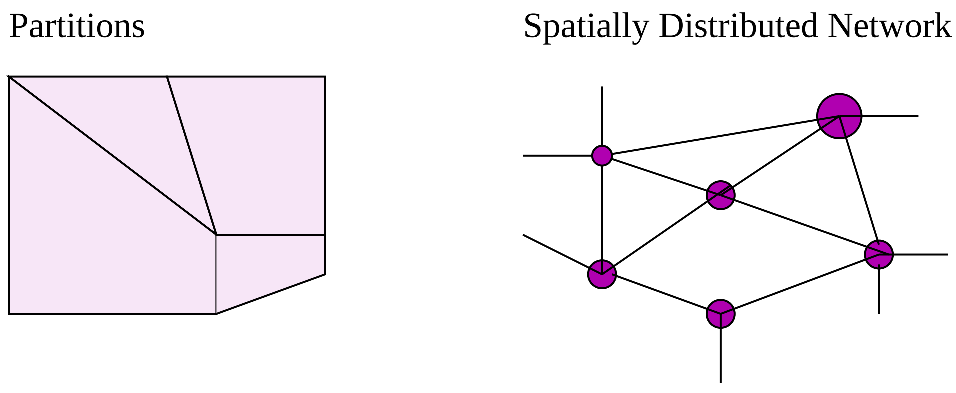


Figure 13.3. Partitions and networks are two abstractions for modeling collections of spatial objects.

Examples of partitions include: rooms in a building, districts in a state, countries in a continent, and so forth.

Conceptual Modeling of Partitions and Networks

Conceptual Model for Partitions. Figure 13.4 says:

1. A Partition can be decomposed into 1 or more Partitions (sub-Partitions).

2. Each Partition has one boundary (here we ignore the possibility of partitions containing holes).

3. Boundaries are composed of edges (..at least 3 edges).

4. Each Edge segment has a Node and Link.

5. Nodes and Link are paired in a one-to-one correspondence.

6. A Node has a coordinate.

7. Edges also have Neighboring Partitions.

8. Neighboring Partitions can be classified as to whether they are on the Left and Right of the Edge.

Conceptual Model for Networks. Networks can be conveniently partitioned into two types:

- Topological networks allow for the study of problems associated with connectivity (e.g., How are the nodes connected? Can I get from node A to node E?). Topological networks abstract (remove) geometric concerns from consideration.

- Geometric networks look at relationships among spatial entities such as lines, polygons and solids.

With this division of types in place, Figure 13.5 states:

1. A Network is composed of Features.

2. Each Feature has Geometry and Topology.

3. Geometry is a generalization for Chains and Points...

4. A Chain corresponds to one or more Line segments.

5. A Point has a coordinate.

6. Topology is a generalization for Nodes and Links.

7. Nodes also have coordinates.

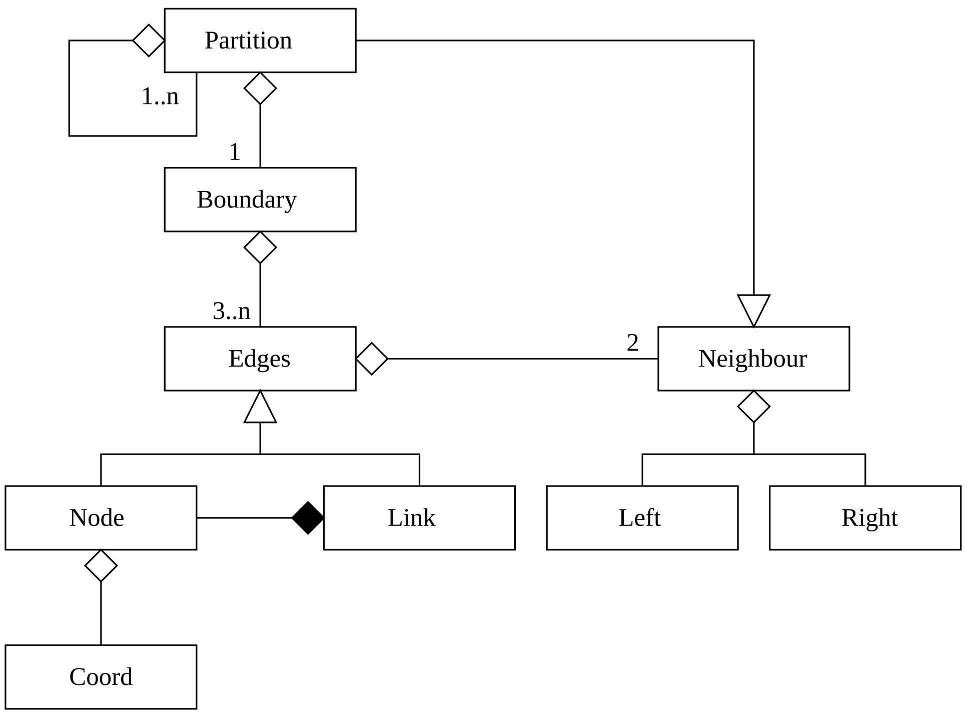


Figure 13.4. Conceptual model for partition hierarchies (Adapted from Chunithipaisan S. et al. [2]).

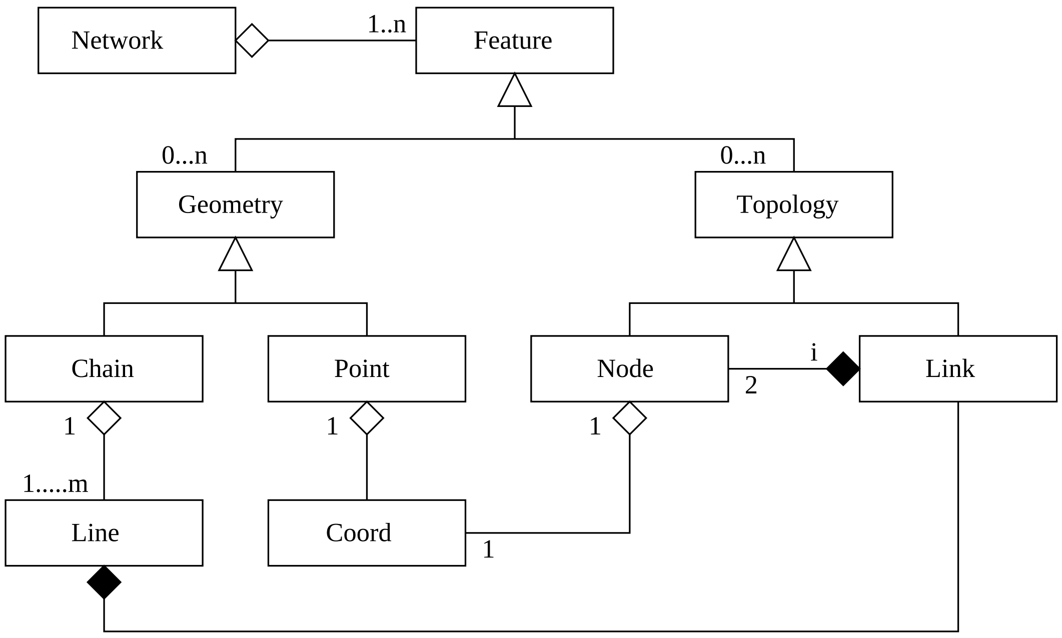


Figure 13.5. Conceptual model for networks (Adapted from Chunithipaisan S. et al. [2]).

Example 3. Modeling Dependencies in Urban Infrastructure

Problems in behavior modeling and management of urban operations can be cast in terms of evalation of dependency and reachability relationships on graphs.

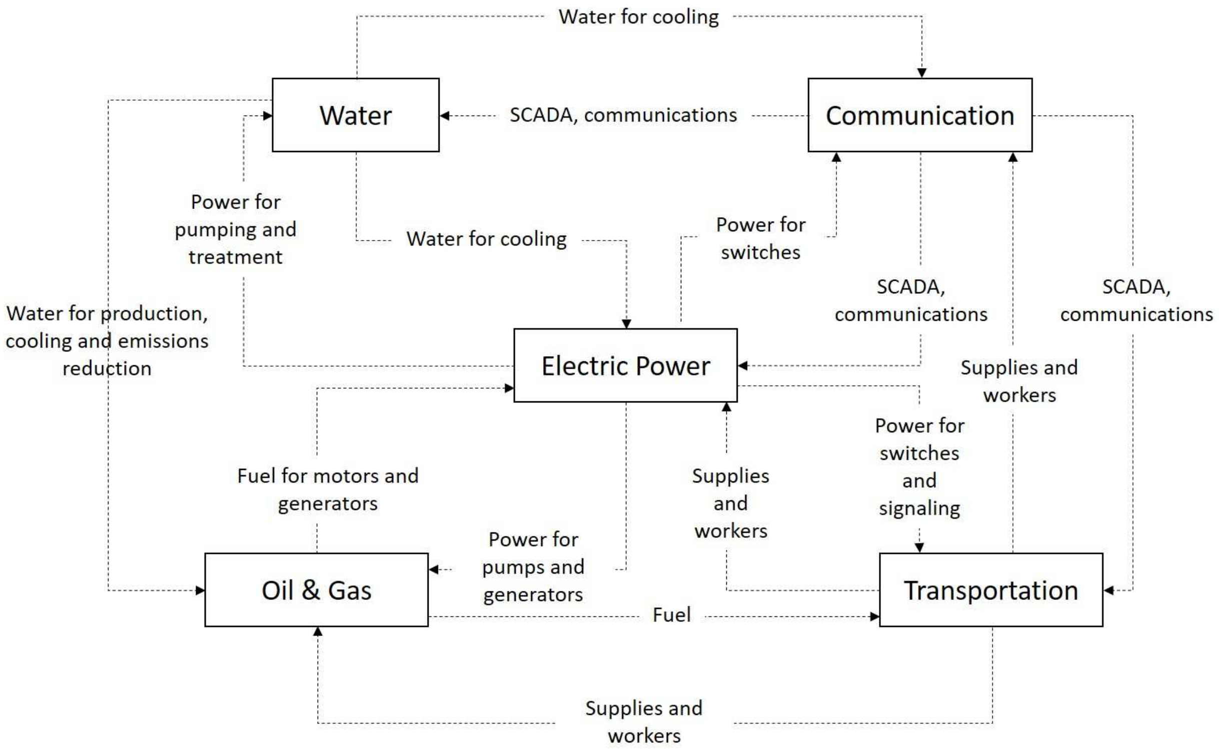


Figure 13.6. Illustration of the interdependent relationship among different infrastructures [3, 4, 6].

Impact of Faults on System Operations

Failure of the energy system (oil and gas) can impact other types of systems:

- Electricity. No fuel to operate power plant motors and generators.
- Transportation. No fuel to operate transport vehicles.
- Water. No fuel to operate pumps and treatment. A pas pipeline failure located beneath roads may contaminate water pipeline also located beneath roads.
- Communication. No fuel to maintain temperatures for equipment; no fuel to backup power.

Failure of the energy system (electricity) can impact other types of systems:

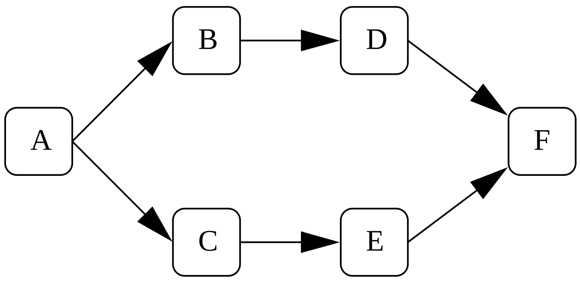
- Transportation. No power for traffic lights, rail systems, street lights. Passengers may be trapped inside trains. Air transport may become compromised due to to the loss of communications and unlit runways.

- Water. No electric power to operate pumps and treatment leading to potential water quality issues and pumping issues in buildings. No power to operate flood protection systems.
- Communication. No energy to run cell towers and other transmission equipment.

Failure of the water system can impact other types of systems:

- Energy. No water available for production, cooling, and emissions reduction.
- Transportation. Water pipeline failure located beneath roads may interrupt traffic.
- Communication. Water pipeline failure located beneath roads may damage power lines located beneath and above roads.

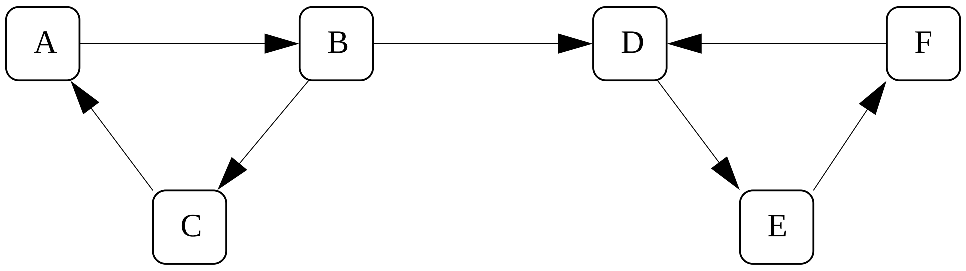
Definition of Graph Dependency. A dependency graph is a directed graph representing dependencies among objects.



The adjacent figure expresses the following list of dependency relationships: Node A depends on nodes B and C. Node B depends on D. Node C depends on E. And nodes D and E both depend on F.

From this list we can state: Node A depends on nodes B, C, D, E and F. Also, if node F fails, then it will impact nodes B, C, D, E and A.

Definition of Graph Reachability Reachability refers to the ability to get from one vertex to another within a graph.



In Figure 13.1, nodes A, B and C can reach nodes D, E and F. The opposite is not true.

13.2 Graph Theory

Graph theory is the study mathematical structures used to model pairwise relations between objects. In this context, a graph consists of vertices (also called nodes or points) which are connected by edges (also called links or lines).

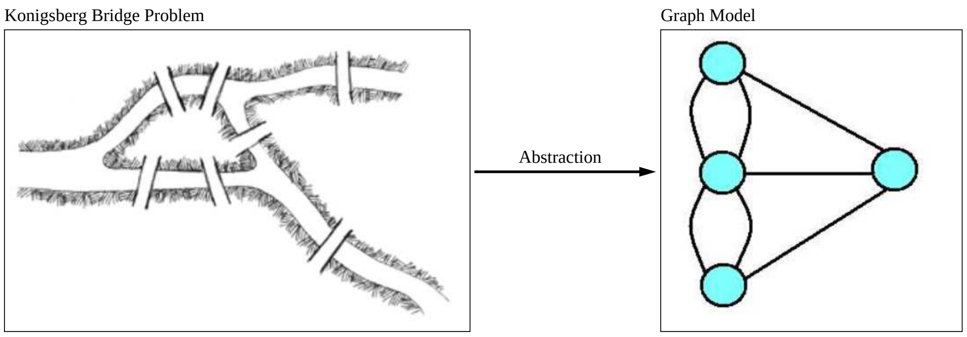


Figure 13.7. Graph model introduced by Leonhard Euler in 1735.

Graph theory dates back to 1735 when Leonhard Euler that examined the following problem: is it possible to take a walk through the town in such a way as to cross over every bridge once, and only once? Euler wanted to understand if solutions to this problem could be cast in terms of the fundamental characteristics of the graph, thereby avoiding the need for exhaustive search. After determining it is impossible, he went on to consider problems involving arbitrary numbers of land masses and bridges. You can find details of this work on the Web.

More Famous Problems on Graphs

Here are a few famous problems on graphs:

- **Hamiltonian Cycle Problem:** Is it possible to traverse each of the vertices of a graph exactly once, starting and ending at the same vertex?
- **Travelling Salesman Problem:** Find the shortest path in a graph that visits each vertex at least once, starting and ending at the same vertex?
- **Planar Graph Problem:** Is it possible to draw the edges of a graph in such a way that edges do not cross?
- **Four Coloring Problem:** Is it possible to color the vertices of a graph with at most 4 colors such that adjacent vertices get different color?
- **Marriage Problem (or bipartite perfect matching):** At what condition a set of boys will be marrying off to a set of girls such that each boy gets a girl he likes?

Mathematical Definition

Mathematical Definition. A graph is a data structure consisting of:

- A set of vertices (or nodes).
- A set of edges (or arcs) connecting the vertices.

Mathematically, a graph G = (V, E), where V is a set of vertices, E = set of edges, and each edge is formed from pair of distinct vertices in V. V and E are usually taken to be finite, and many of the well-known results are not true for infinite graphs because many of the arguments fail in the infinite case.

Notational Definitions. Here are few notational definitions:

1. The vertices belonging to an edge are called the ends, endpoints, or end vertices of the edge.

2. A vertex may exist in a graph and not belong to an edge.

3. The **order of a graph** is (the number of vertices).

4. A graph's **size** is the number of edges.

5. The **degree of a vertex** is the number of edges that connect to it, where an edge that connects to the vertex at both ends (a loop) is counted twice.

6. Vertices i and j are **adjacent** if (i, j) is an edge in the graph. The edge (i, j) is incident on vertices i and j.

7. A loop is a sequence of edges which starts and ends on the same vertex. Depending on the application at hand, these may or may not be permitted.

Figure 13.8 shows a small graph.

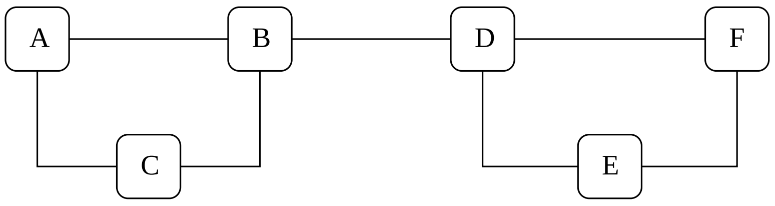


Figure 13.8. A small graph.

The order of the graph is six. The size of the graph is seven. Nodes A, C, E and F have degree 2. Nodes B and D have degree 3. Node A is adjacent to nodes B and C. The graph contains two loops, A-B-C and D-E-F.

Types of Graph

Figure 13.9 shows connectivity and edge properties in four types of graphs: undirected, directed, mixed and multigraphs.

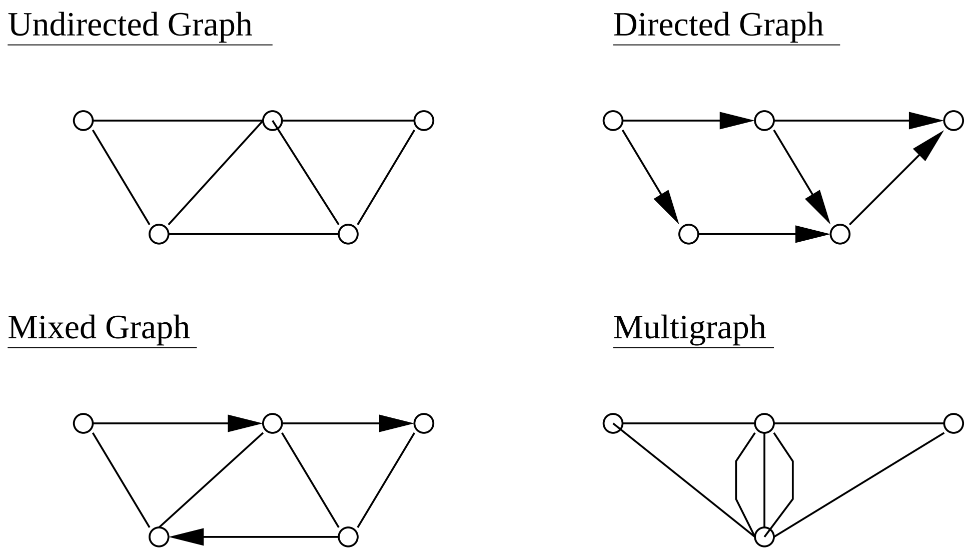


Figure 13.9. Undirected, directed, mixed and multigraph types.

Undirected Graph

An undirected graph is one in which edges have no orientation. The edge (a, b) is identical to the edge (b, a).

Directed Graph

A directed graph or digraph is an ordered pair D = (V, A) with

- V a set whose elements are called vertices or nodes, and
- A a set of ordered pairs of vertices, called arcs, directed edges, or arrows.

An arc a = (x, y) is considered to be directed from x to y:

- y is called the head and x is called the tail of the arc;
- y is said to be a direct successor of x,
- x is said to be a direct predecessor of y.

If a path leads from x to y, then y is said to be a successor of x and **reachable** from x, and x is said to be a predecessor of y.

The arc (y, x) is called the arc (x, y) inverted.

Mixed Graph

A mixed graph G is a graph in which some edges may be directed and some may be undirected. It is written as ...

- An ordered triple G = (V, E, A) with V, E, and A defined as above.

Directed and undirected graphs are special cases.

Multigraph.

Multigraphs allow for multiple edges between nodes.

Labeled Graph

A labelled graph adds names to vertices.

Weighted Graph

A graph is a weighted graph if ...

> **... a number (weight) is assigned to each edge.**

Such weights might represent, for example, costs, lengths, capacities, distances, or time etc. depending on the problem at hand. Some authors call such a graph a network.

Simple Graphs

A simple graph is ...

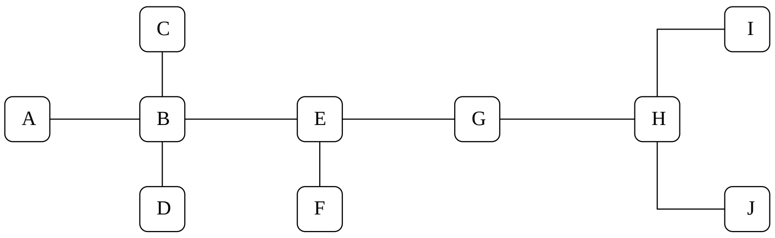
> **... an undirected graph that has no loops and no more than one edge between any two different vertices.**

A few points to note:

- The edges of a simple graph form a set, with each item in the set having a distinct pair of vertices.
- Suppose that a simple graph has n vertices. Every vertex in the graph will have degree less than n.

Trees

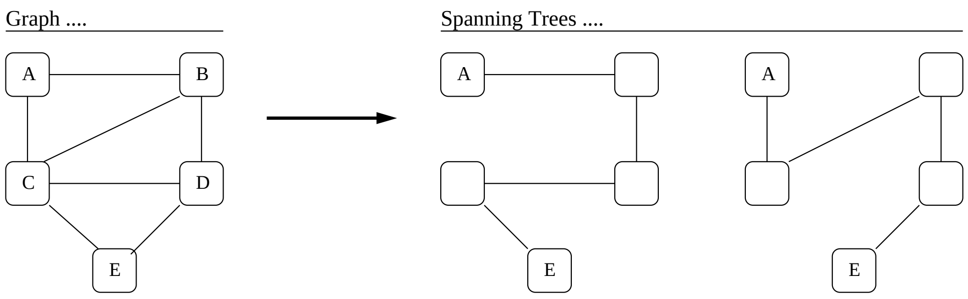
A tree is a connected simple graph without cycles (or loops).



Points to note:

- A tree with n vertices must have $n-1$ edges. Conversely, a connected graph with n vertices and $n-1$ edges must be a tree.
- A **rooted tree** is a tree with one vertex designated as a root.
- A forest is a graph without cycles. In orther words, a forest is a set of trees.

The **spanning tree** of a graph is a subgraph, which is a tree and contains all vertices of the graph.



Spanning trees are not unique, as illustrated in the adjacent figure.

Graph Paths

Definition. A graph **graph** is a sequence of distinctive vertices connected by edges.

- A path may be infinite, but a finite path always has a first vertex, called its start vertex, and a last vertex, called its end vertex.
- A **cycle** is a path such that the start vertex and end vertex are the same. The choice of the start vertex in a cycle is arbitrary.
- A graph is **connected** if there is a path between any two vertices.

Types of Path. Here are a few definitions:

- A path with no repeated vertices is called a **simple path**.
- A cycle with no repeated vertices or edges aside from the necessary repetition of the start and end vertex is a **simple cycle**.
- A simple path that includes every vertex of the graph is known as a **Hamiltonian path**.
- A simple cycle that includes every vertex of the graph is known as a **Hamiltonian cycle**.

Graph Connectivity Relationships

Definitions. Here are a few definitions:

- In a directed graph vertex v is **adjacent** to u, if there is an edge leaving v and coming to u.
- In a directed graph the **in-degree** of a vertex denotes the number of edges coming to this vertex. The **out-degree** of a vertex is the number of edges leaving the vertex.
- A graph is called **connected** if every pair of distinct vertices in the graph is connected. Otherwise, it is called **disconnected**.
- A directed graph is called **strongly connected** if there is a path from each vertex in the graph to every other vertex.
- A directed graph is called **weakly connected** if replacing all of its directed edges with undirected edges produces a connected (undirected) graph.

A strongly connected directed graph implies bi-directional connectivity of nodes – that is, a path from node u to node v is accompanied by path from node v to node u.

Directed oneway loops within a graph will have collections of nodes that form a strong subgraph.

Example. Figure 13.10 shows a graph that is weakly connected.

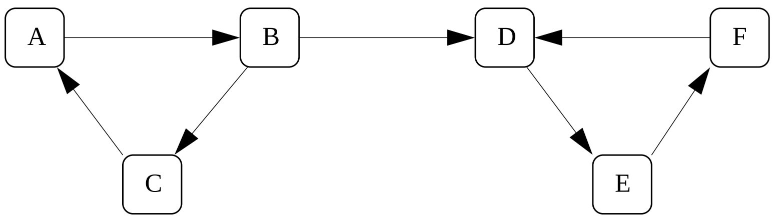
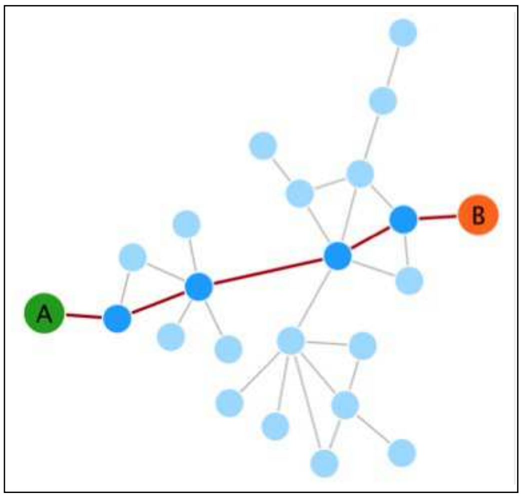


Figure 13.10. A directed graph that is weakly connected.

It is not **strongly connected** because there is no way to get from the set of nodes $[D, E, F]$ to $[A, B, C]$. Adding a directed edge from node D to node B would solve that problem.

The subgraphs defined by sets of nodes $[A, B, C]$ and $[D, E, F]$ are each strongly connected.

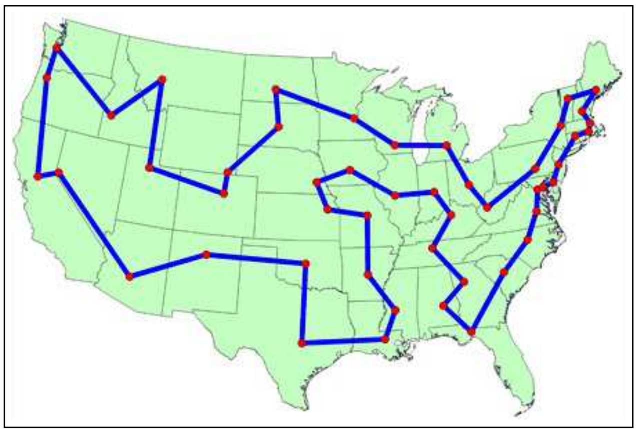
Shortest Path on a Graph. Transportation engineers are obsessed with finding ways to compute the shortest path (or least cost path) between points in a city and scheduling traffic operations to minimize cost.



Computation of the shortest begins with selection of a start node A, followed by a systematic search of graph edges to find the best – shortest or cheapest – path to destination node B. Solutions to real-world problems are complicated by a myriad of ways in which outside influences (e.g., weather, maintenance, weekday vs weekend) can affect network demand and available network capacity.

Figure 13.11. Shortest path between two points on a graph.

Travelling Salesman Problem. Given a list of city locations, the travelling salesman problem asks: What is the shortest possible route that visits each city and returns to the origin city?"



This problem is $O(n!)$ hard.

Figure 13.12. Shortest path for visiting state capitals in Lower 48.

13.3 Graph Representation

There are two standard ways to represent graphs:

- As collections of adjacency (edge) lists.
- As adjacency matrices.

Adjacency lists are ideal for the representation of the sparse graphs. Adjacency matrix representations are the preferred approach for graphs that are densely populated.

Adjacency Lists

An adjacency (or edge) list representation of a graph consists of an array (or named collection) of linked lists, one for each vertex. Each such list contains all vertices adjacent to a chosen one.

Here is an example:

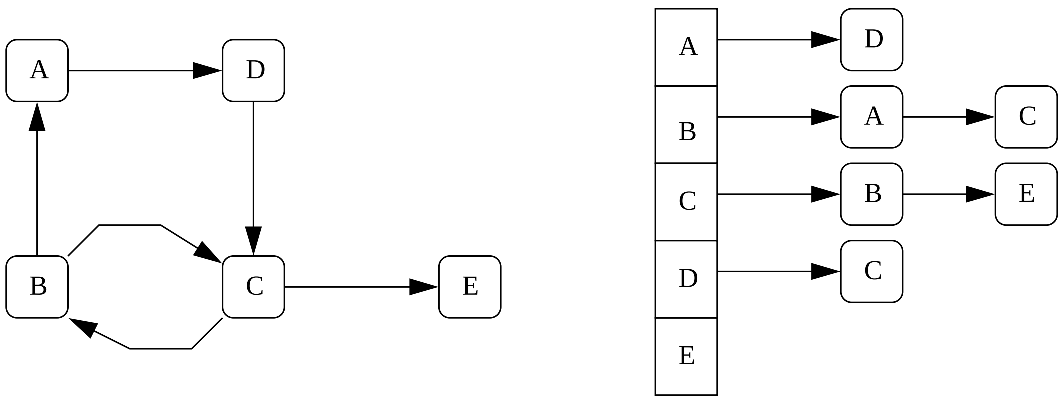


Figure 13.13. Adjacency list representation for a graph.

A HashMap would be perfect for implementing edge lists.

The main disadvantage of this approach is that there is no quick way to determine if an edge exists between two nodes.

Adjacency Matricies

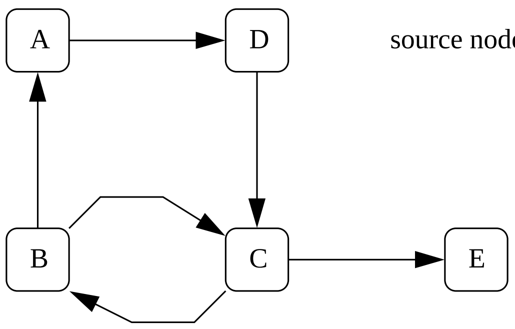
The adjacency matrix of a graph having n vertices is the n $\times$ n matrix where the non-diagonal entry a_{ij} is the number of edges from vertex i to vertex j.

Mathematically we can write,

$$a_{ij} = \begin{cases} k, \text{when k edges exist between nodes i and j,} \\ 0, \text{otherwise.} \end{cases} \tag{13.1}$$

For most graphs k will be 1. The adjacency matrix will be symmetric for undirected networks.

Example 1.



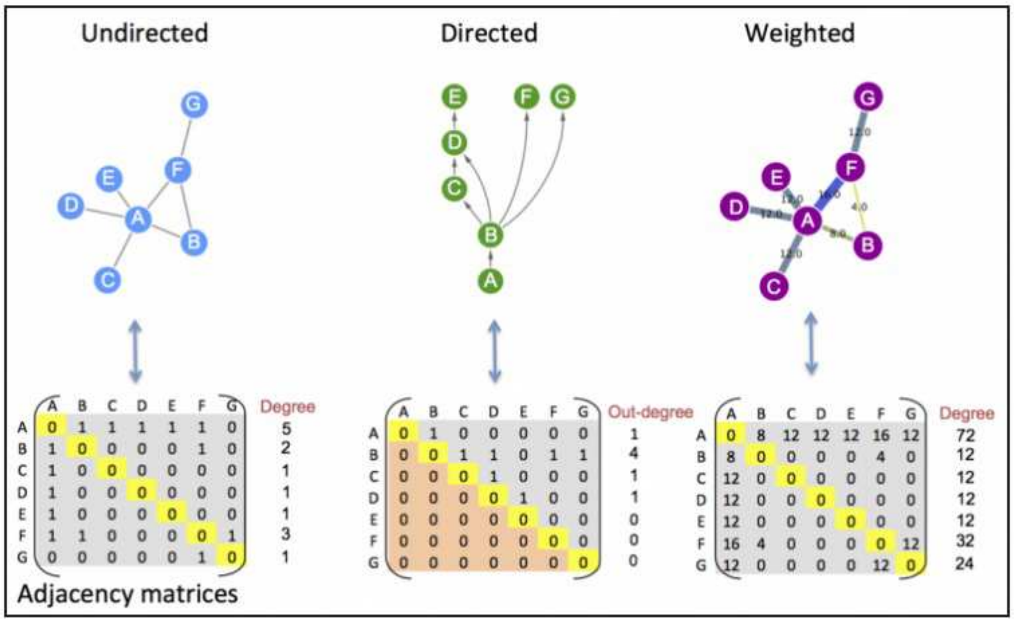
target node →

source node →

	A	B	C	D	E
A	0	0	0	1	0
B	1	0	1	0	0
C	0	1	0	0	1
D	0	0	1	0	0
E	0	0	0	0	0

Figure 13.14. Adjacency list representation for a graph.

Example 2. Many variations on this theme are possible. Figure 13.15 shows three graphs and their corresponding adjacency matrices for undirected, directed and weighted networks.



	A	B	C	D	E	F	G	Degree
A	0	1	1	1	1	1	0	5
B	1	0	0	0	0	1	0	2
C	1	0	0	0	0	0	0	1
D	1	0	0	0	0	0	0	1
E	1	0	0	0	0	0	0	1
F	1	1	0	0	0	0	1	3
G	0	0	0	0	0	1	0	1

	A	B	C	D	E	F	G	Out-degree
A	0	1	0	0	0	0	0	1
B	0	0	1	1	0	1	1	4
C	0	0	0	1	0	0	0	1
D	0	0	0	0	1	0	0	1
E	0	0	0	0	0	0	0	0
F	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0

	A	B	C	D	E	F	G	Degree
A	0	8	12	12	12	16	12	72
B	8	0	0	0	0	4	0	12
C	12	0	0	0	0	0	0	12
D	12	0	0	0	0	0	0	12
E	12	0	0	0	0	0	0	12
F	16	4	0	0	0	0	12	32
G	12	0	0	0	0	12	0	24

Figure 13.15. Schematic of adjacency matrices for undirected, directed and weighted networks.

Directed and weighted networks can make use of different numerical values in the matrix to express these more complex relationships. The sign of the values, for example, is sometimes used to indicate stimulation or inhibition.

13.4 Liang's Graph Package (Graph Representation)

This section is adapted from Chapter 27 of Liang's Java book.

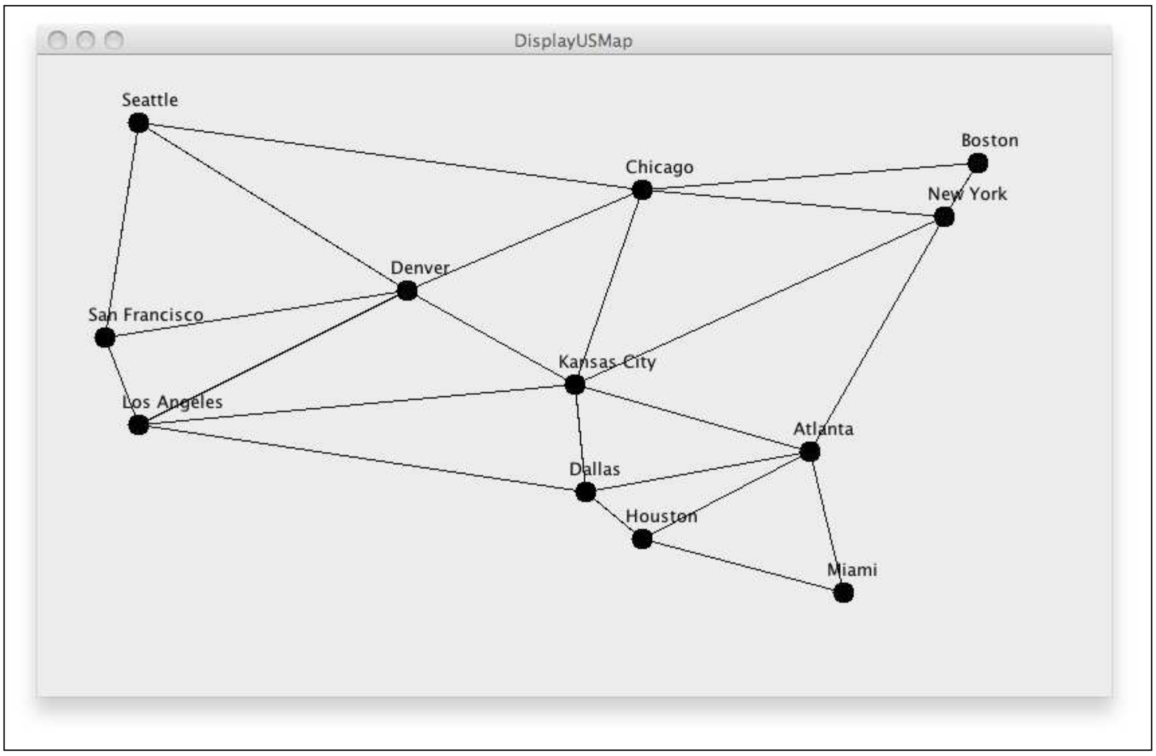


Figure 13.16. Network of US cities.

We present a variety of graph algorithms for analyzing a network of US cities. See Figure 13.16.

In algorithms that depend only on the details of city-to-city connetivity (and not geometry), the graph vertices and edges can be simply declared as arrays of character strings and integers, i.e.,

```
String[] vertices = {   "Seattle", "San Francisco",  ....  "Houston"};
int[][] edges = { { 0, 1}, { 0, 3}, { 0,  5}, .....
                  {11, 8}, {11, 9}, {11, 10}
};
```

For the generation of Figure 13.16, the array of character strings is replaced by an array of references to City objects, each initialized with their name and coordinates, i.e.,

```
private City[] vertices = { new City(      "Seattle",  75,  50),
                            new City("San Francisco",  50, 210),

                            ... city details removed ....

                            new City(        "Miami", 600, 400),
                            new City(       "Dallas", 408, 325),
                            new City(      "Houston", 450, 360) };
```

Here the coordinates are pixel coordinates – a more elegant solution would work with the actual longitude and latitude of the city and make the appropriate transformation to screeen coordinates.

Source code: Look in java-code-graphs/liang/

Graph Interface Classes and Abstract Classes

Figure 13.17 shows the hierarchy of interfaces, abstract classes and implementation classes in Liang's graph representation.

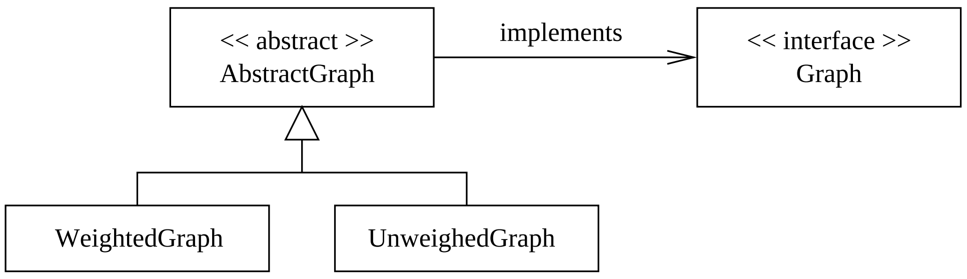


Figure 13.17. Hierarchy of graph classes and interfaces.

Modeling support is provided for unweighted and weighted graphs, stored in either an adjacecy matrix format or edge list format, and breadth-first and depth-first traversal of the graph.

File: src/liang/model/graph/Graph.java.

The graph functionality is defined through a Graph interface.

source code

```
/*
 *  ======================================================================
 *  Graph.java: Graph interface ...
 *  ======================================================================
 */

package liang.model.graph;

public interface Graph<V> {

   /** Return the number of vertices in the graph */

   public int getSize();

   /** Return the vertices in the graph */

   public java.util.List<V> getVertices();

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

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

   public int getIndex(V v);

   /** Return the neighbors of vertex with the specified index */

   public java.util.List<Integer> getNeighbors(int index);

   /** Return the degree for a specified vertex */

   public int getDegree(int v);

   /** Return the adjacency matrix */

   public int[][] getAdjacencyMatrix();

   /** Print the adjacency matrix */

   public void printAdjacencyMatrix();

   /** Print the edges */

   public void printEdges();

   /** Obtain a depth-first search tree */

   public AbstractGraph<V>.Tree dfs(int v);

   /** Obtain a breadth-first search tree */

   public AbstractGraph<V>.Tree bfs(int v);

   /** Return a Hamiltonian path from the specified vertex
     * Return null if the graph does not contain a Hamiltonian path */

   public java.util.List<Integer> getHamiltonianPath(V vertex);

   /** Return a Hamiltonian path from the specified vertex label
     * Return null if the graph does not contain a Hamiltonian path */

   public java.util.List<Integer> getHamiltonianPath(int inexe);
}
```

Graphs of various data types can be assembled – see, for example, AbstractGraph¡V¿.Tree bfs(int v). Also notice that the interface contains explicit references to the List interface.

File: src/liang/model/graph/AbstractGraph.java.

Most of the algorithmic support for the implementation of a graph interface is provided in the Abstract-Graph class. As we will soon see, unweighed and weighted graph implemenations will be modeled as extensions of the specification provided within the abstract class.

source code

```
/*  ======================================================================
 *  Abstract Graph.java: Abstract class to implement graph interface ...
 *  ======================================================================
 */

package liang.model.graph;
import java.util.*;

public abstract class AbstractGraph<V> implements Graph<V> {

   protected List<V>              vertices; // Store vertices
   protected List<List<Integer>> neighbors; // Adjacency lists

   /** Construct a graph from edges and vertices stored in arrays */

   protected AbstractGraph( int[][] edges, V[] vertices ) {
      this.vertices = new ArrayList<V>();
      for (int i = 0; i < vertices.length; i++)
        this.vertices.add(vertices[i]);

      createAdjacencyLists(edges, vertices.length);
   }

   /** Construct a graph from edges and vertices stored in List */

   protected AbstractGraph( List<Edge> edges, List<V> vertices ) {
      this.vertices = vertices;
      createAdjacencyLists(edges, vertices.size());
   }

   /** Construct a graph for integer vertices 0, 1, 2 and edge list */

   protected AbstractGraph( List<Edge> edges, int numberOfVertices ) {
      vertices = new ArrayList<V>(); // Create vertices
      for (int i = 0; i < numberOfVertices; i++) {
         vertices.add((V)(new Integer(i))); // vertices is {0, 1, ...}
      }
      createAdjacencyLists(edges, numberOfVertices);
   }

   /** Construct a graph from integer vertices 0, 1, and edge array */

   protected AbstractGraph(int[][] edges, int numberOfVertices) {
      vertices = new ArrayList<V>(); // Create vertices
      for (int i = 0; i < numberOfVertices; i++ ) {
         vertices.add((V)(new Integer(i))); // vertices is {0, 1, ...}
      }
      createAdjacencyLists(edges, numberOfVertices);
   }

   /** Create adjacency lists for each vertex */

   private void createAdjacencyLists(
      int[][] edges, int numberOfVertices) {
```

```
      // Create a linked list

      neighbors = new ArrayList<List<Integer>>();
      for (int i = 0; i < numberOfVertices; i++) {
         neighbors.add(new ArrayList<Integer>());
      }

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

   /** Create adjacency lists for each vertex */

   private void createAdjacencyLists(
      List<Edge> edges, int numberOfVertices) {

      // Create a linked list

      neighbors = new ArrayList<List<Integer>>();
      for (int i = 0; i < numberOfVertices; i++) {
         neighbors.add(new ArrayList<Integer>());
      }

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

   /** Return the number of vertices in the graph */

   public int getSize() { return vertices.size(); }

   /** Return the vertices in the graph */

   public List<V> getVertices() { return vertices; }

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

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

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

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

   /** Return the neighbors of vertex with the specified index */

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

  /** Return the degree for a specified vertex */

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

  /** Return the adjacency matrix */

  public int[][] getAdjacencyMatrix() {
    int[][] adjacencyMatrix = new int[getSize()][getSize()];

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

    return adjacencyMatrix;
  }

  /** Print the adjacency matrix */

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

      System.out.println();
    }
  }

  /** Print the edges */

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

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

  public static class Edge {
    public int u; // Starting vertex of the edge
    public int v; // Ending vertex of the edge
```

```
      /** Construct an edge for (u, v) */

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

   /** Obtain a DFS tree starting from vertex v */
   /** To be discussed in Section 27.6 */

   public Tree dfs(int v) {

    ... details of code removed ...

   }

   /** Starting bfs search from vertex v */
   /** To be discussed in Section 27.7 */

   public Tree bfs(int v) {

    ... details of code removed ...

   }

   /** Tree inner class inside the AbstractGraph class */
   /** To be discussed in Section 27.5 */

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

      ... details of code removed ...

   }

   /** Return a Hamiltonian path from the specified vertex object
     * Return null if the graph does not contain a Hamiltonian path */

   public List<Integer> getHamiltonianPath(V vertex) {
      return getHamiltonianPath(getIndex(vertex));
   }

   /** Reorder the adjacency list in increasing order of degrees */

   private void reorderNeigborsBasedOnDegree(List<Integer> list) {

      .... details of code removed ....

   }

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

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

Notice that the vertices are stored as lists, and the adjacency lists are stored as a list of lists, i.e.,

```
   protected List<V>              vertices; // Store vertices
   protected List<List<Integer>> neighbors; // Adjacency lists
```

File: src/liang/model/graph/UnweightedGraph.java.

source code

```
/*
 *  ======================================================================
 *  Unweighted Graph.java: Implementation for Unweighted Graphs ....
 *  ======================================================================
 */

package liang.model.graph;

import java.util.*;

public class UnweightedGraph<V> extends AbstractGraph<V> {

   /** Construct a graph from edges and vertices stored in arrays */

   public UnweightedGraph(int[][] edges, V[] vertices) {
      super(edges, vertices);
   }

   /** Construct a graph from edges and vertices stored in List */

   public UnweightedGraph(List<Edge> edges, List<V> vertices) {
      super(edges, vertices);
   }

   /** Construct a graph for integer vertices 0, 1, 2 and edge list */

   public UnweightedGraph(List<Edge> edges, int numberOfVertices) {
      super(edges, numberOfVertices);
   }

   /** Construct a graph from integer vertices 0, 1, and edge array */

   public UnweightedGraph(int[][] edges, int numberOfVertices) {
      super(edges, numberOfVertices);
   }
}
```

File: src/liang/model/graph/WeightedEdge.java.

source code

```
/*
 *  ======================================================================
 *  WeightedGraphEdge.java: Implementation for Weighted Graphs ...
 *  ======================================================================
 */

package liang.model.graph;

public class WeightedEdge extends AbstractGraph.Edge implements Comparable<WeightedEdge> {
   public int weight; // The weight on edge (u, v)

   /** Create a weighted edge on (u, v) */

   public WeightedEdge(int u, int v, int weight) {
      super(u, v);
      this.weight = weight;
   }

   /** Compare two edges on weights */

   public int compareTo( WeightedEdge edge ) {
      if (weight > edge.weight)
         return 1;
      else if (weight == edge.weight) {
         return 0;
      } else {
         return -1;
      }
   }
}
```

File: src/liang/model/graph/WeightedGraph.java.

source code

```
/*
 *  ======================================================================
 *  WeightedGraph.java: Implementation for Weighted Graphs ...
 *  ======================================================================
 */

package liang.model.graph;
```

```
import java.util.*;

public class WeightedGraph<V> extends AbstractGraph<V> {

   // Priority adjacency lists

   private List<PriorityQueue<WeightedEdge>> queues;

   /** Construct a WeightedGraph from edges and vertices in arrays */

   public WeightedGraph(int[][] edges, V[] vertices) {
      super(edges, vertices);
      createQueues(edges, vertices.length);
   }

   /** Construct a WeightedGraph from edges and vertices in List */

   public WeightedGraph(int[][] edges, int numberOfVertices) {
      super(edges, numberOfVertices);
      createQueues(edges, numberOfVertices);
   }

   /** Construct a WeightedGraph for vertices 0, 1, 2 and edge list */

   public WeightedGraph(List<WeightedEdge> edges, List<V> vertices) {
      super((List)edges, vertices);
      createQueues(edges, vertices.size());
   }

   /** Construct a WeightedGraph from vertices 0, 1, and edge array */

   public WeightedGraph( List<WeightedEdge> edges, int numberOfVertices ) {
      super((List)edges, numberOfVertices);
      createQueues(edges, numberOfVertices);
   }

   /** Create priority adjacency lists from edge arrays */

   private void createQueues(int[][] edges, int numberOfVertices) {
      ... details of code removed ...
   }

   /** Create priority adjacency lists from edge lists */

   private void createQueues(List<WeightedEdge> edges, int numberOfVertices) {
      ... details of code removed ...
   }

   /** Display edges with weights */

   public void printWeightedEdges() {
      ... details of code removed ...
   }
   /** Get a minimum spanning tree rooted at vertex 0 */

   public MST getMinimumSpanningTree() {
      return getMinimumSpanningTree(0);
   }

   /** Get a minimum spanning tree rooted at a specified vertex */

   public MST getMinimumSpanningTree(int startingIndex) {
      ... details of code removed ...
   }

   /** MST is an inner class in WeightedGraph */

   public class MST extends Tree {
      ... details of code removed ...
   }

   /** Find single source shortest paths */

   public ShortestPathTree getShortestPath(int sourceIndex) {
      ... details of code removed ...
   }

   public void addVertex(V vertex) {
      super.addVertex(vertex);
      queues.add(new PriorityQueue<WeightedEdge>());
   }

   public void addEdge(int u, int v, int weight) {
      super.addEdge(u, v);
      queues.get(u).add(new WeightedEdge(u, v, weight));
      queues.get(v).add(new WeightedEdge(v, u, weight));
   }
}
```

Example 1. A Simple Graph Representation

Problem Statement. This example does two things:

1. We build an unweighted graph model for Figure 13.16, and then investigate properties of the graph, such as retrieving the number of vertices and the adjacency relationships.
2. We build a small unweighted graph defined by five nodes and an arraylist of AbstractGraph.Edge (this is an inner class) objects.

File: src/demo/TestGraph.java.

In this program we exercise the abstract graph and graph interface.

source code

```
/*
 *  =========================================================================
 *  TestGraph.java: Exercise abstract graph and graph interface ....
 *  =========================================================================
 */

package demo;

import liang.model.graph.*;

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

      int[][] edges = {
          {0, 1}, {0, 3}, {0, 5},
          {1, 0}, {1, 2}, {1, 3},
          {2, 1}, {2, 3}, {2, 4}, {2, 10},
          {3, 0}, {3, 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}
      };

      System.out.println("Run TestGraph()                          ");
      System.out.println("=========================================");

      Graph<String> graph1 = new UnweightedGraph<String>(edges, vertices);

      System.out.println("The number of vertices in graph1: " + graph1.getSize());
      System.out.println("The vertex with index 1 is " + graph1.getVertex(1));
      System.out.println("The index for Miami is " + graph1.getIndex("Miami"));
      System.out.println("The edges for graph1:");
      graph1.printEdges();

      System.out.println("Adjacency matrix for graph1:");
      graph1.printAdjacencyMatrix();

      // List of Edge objects for graph in Figure 27.3(a)

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

      java.util.ArrayList<AbstractGraph.Edge>
           edgeList = new java.util.ArrayList<AbstractGraph.Edge>();

      edgeList.add(new AbstractGraph.Edge(0, 2));
      edgeList.add(new AbstractGraph.Edge(1, 2));
      edgeList.add(new AbstractGraph.Edge(2, 4));
      edgeList.add(new AbstractGraph.Edge(3, 4));

      // Create a graph with 5 vertices

      Graph<String> graph2 =
           new UnweightedGraph<String> (edgeList, java.util.Arrays.asList(names));

      System.out.println("The number of vertices in graph2: " + graph2.getSize());
      System.out.println("The edges for graph2:"); graph2.printEdges();
      System.out.println("\nAdjacency matrix for graph2:");
      graph2.printAdjacencyMatrix();

      for (int i = 0; i < 5; i++)
         System.out.println("vertex " + i + ": " + graph2.getVertex(i));

      System.out.println("=========================================");
   }
}
```

Input and Output. The script of program input and output is:

```
Script started on Mon Apr  9 17:32:42 2012
prompt >>
prompt >> ant run01
Buildfile: /Users/austin/ence688r.d/java-code-graphs/liang/build.xml

compile:
    [javac] /Users/austin/ence688r.d/java-code-graphs/liang/build.xml:9:
            warning: 'includeantruntime' was not set,
            defaulting to build.sysclasspath=last; set to false for repeatable builds
    [javac] Compiling 1 source file to /Users/austin/ence688r.d/java-code-graphs/liang/bin

run01:
     [java] Run TestGraph()
     [java] =========================================
     [java] The number of vertices in graph1: 12
     [java] The vertex with index 1 is San Francisco
     [java] The index for Miami is 9
     [java] The edges for graph1:
     [java] Vertex 0: (0, 1) (0, 3) (0, 5)
     [java] Vertex 1: (1, 0) (1, 2) (1, 3)
     [java] Vertex 2: (2, 1) (2, 3) (2, 4) (2, 10)
     [java] Vertex 3: (3, 0) (3, 1) (3, 2) (3, 4) (3, 5)
     [java] Vertex 4: (4, 2) (4, 3) (4, 5) (4, 7) (4, 8) (4, 10)
     [java] Vertex 5: (5, 0) (5, 3) (5, 4) (5, 6) (5, 7)
     [java] Vertex 6: (6, 5) (6, 7)
     [java] Vertex 7: (7, 4) (7, 5) (7, 6) (7, 8)
     [java] Vertex 8: (8, 4) (8, 7) (8, 9) (8, 10) (8, 11)
     [java] Vertex 9: (9, 8) (9, 11)
     [java] Vertex 10: (10, 2) (10, 4) (10, 8) (10, 11)
     [java] Vertex 11: (11, 8) (11, 9) (11, 10)
     [java] Adjacency matrix for graph1:
     [java] 0 1 0 1 0 1 0 0 0 0 0 0
     [java] 1 0 1 1 0 0 0 0 0 0 0 0
     [java] 0 1 0 1 1 0 0 0 0 0 1 0
     [java] 1 1 1 0 1 1 0 0 0 0 0 0
     [java] 0 0 1 1 0 1 0 1 1 0 1 0
     [java] 1 0 0 1 1 0 1 1 0 0 0 0
     [java] 0 0 0 0 0 1 0 1 0 0 0 0
     [java] 0 0 0 0 1 1 1 0 1 0 0 0
     [java] 0 0 0 0 1 0 0 1 0 1 1 1
     [java] 0 0 0 0 0 0 0 0 1 0 0 1
     [java] 0 0 1 0 1 0 0 0 1 0 0 1
     [java] 0 0 0 0 0 0 0 0 1 1 1 0
     [java] The number of vertices in graph2: 5
     [java] The edges for graph2:
     [java] Vertex 0: (0, 2)
     [java] Vertex 1: (1, 2)
     [java] Vertex 2: (2, 4)
     [java] Vertex 3: (3, 4)
     [java] Vertex 4:
     [java]
     [java] Adjacency matrix for graph2:
     [java] 0 0 1 0 0
     [java] 0 0 1 0 0
     [java] 0 0 0 0 1
     [java] 0 0 0 0 1
     [java] 0 0 0 0 0
     [java] vertex 0: Peter
     [java] vertex 1: Jane
     [java] vertex 2: Mark
     [java] vertex 3: Cindy
     [java] vertex 4: Wendy
     [java] =========================================

BUILD SUCCESSFUL
Total time: 3 seconds
prompt >> exit
Script done on Mon Apr  9 17:32:58 2012
```

13.5 Algorithms for Tree Traversal

Tree structures may be traversed in a variety of ways.

Breath-First Tree Search

A breadth-first tree search (BFS) ...

> traverses and prints each level of the tree left to right before moving onto the next level and repeating the process.

For a small tree this process looks like:

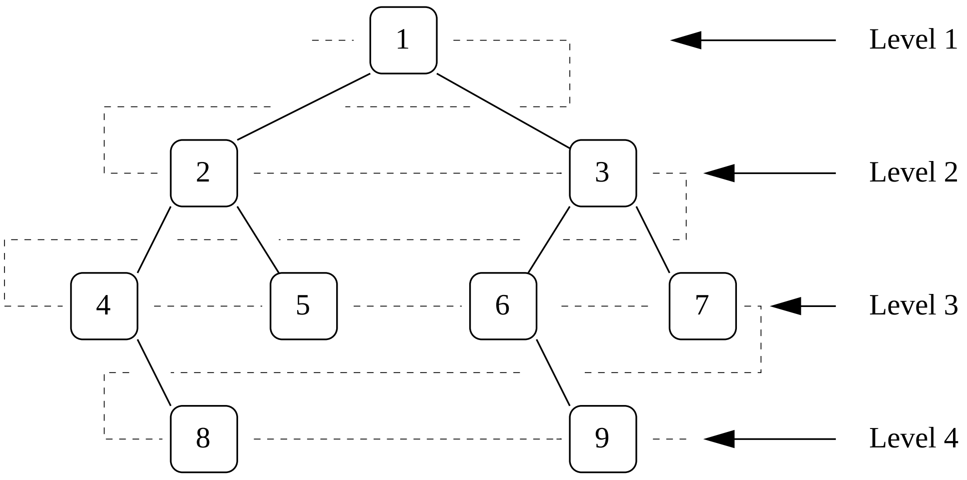


Figure 13.18. Breadth-first search of a small tree.

and the nodes will be visited and printed in the order: 1, 2, 3, 4, 5, 6, 7, 8 and 9.

There are two basic approaches to traversing the tree:

Method 1. Use a function to print a given level. The basic strategy is as follows:

```
// Method to print level order traversal of tree ...

printLevelorder(tree)
for d = 1 to height(tree)
    printGivenLevel(tree, d);

// Method to print all nodes at a given level ...

printGivenLevel(tree, level)
   if tree is NULL then return;
   if level is 1, then
      print(tree->data);
   else if level greater than 1, then
```

```
          printGivenLevel(tree->left, level-1);
          printGivenLevel(tree->right, level-1);
```

Method 2. For each node, first the node is visited and then its child nodes are put in a FIFO queue. The basic strategy is as follows:

```
printLevelorder(tree)

1) Create an empty queue q

2) temp_node = root    // Start from root ...

3) Loop while temp_node is not NULL
   (a) Print temp_node->data.
   (b) Enqueue temp_nodes children (first left then right children) to q
   (c) Dequeue a node from q and assign its value to temp_node
```

In the worst case (a skewed tree), method 1 has time complexity $O(n^2)$. Method 2 uses the queue to reduce the time complexity to $O(n)$. Much better!.

Example Code. See `java-code-tree` handed out in class.

Depth-First Tree Search

A depth-first tree search (DFS) ...

> ... starts at the root at the root of the tree explores as far as possible (i.e., deeper and deeper) along each branch until it hits a node that has no children.

Algorithms that use depth-first search as a building block include:

- Solving puzzles with only one solution, such as mazes.
- Finding connected components,
- Topological sorting (details to come), and
- Finding strongly connected components.

There are three basic approaches to depth-first traversal of a tree (they differ in the order in which a node is processed):

Method 1. Pre-order traversal prints the content of a node when it is first visited.

The implementation is as follows:

```
Preorder( u) {
   1. Print content of node u.
   2. Call Preorder(u.left);
   3. Call Preorder(u.right);
}
```

The main program calls Preorder(root).

Method 2. In-order traversal prints content of node just before we visit right child.

The implementation is as follows:

```
Inorder( u) {
   1. Call Inorder(u.left);
   2. Print content of node u.
   3. Call Inorder(u.right);
}
```

The main program calls Preorder(root).

Method 3. Post-order traversal prints the content of a tree when it is last visited.

The implementation is as follows:

```
Postorder( u) {
   1. Call Postorder(u.left);
   2. Call Postorder(u.right);
   3. Print content of node u.
}
```

The main program calls Preorder(root).

Points to Note:

1. For a binary tree each node will be **visited three times**, the times before the DFS and the times after each DFS.

2. Pre- and post-order are well defined for non-binary trees.

Examples. Applying the three strategies to the example problem (see Figure 13.18):

- Preorder: 1, 2, 4, 8, 5, 3, 6, 9 and 7.
- Inorder: 4, 8, 2, 5, 1, 6, 9, 3 and 7.
- PostOrder: 8, 4, 5, 2, 9, 6, 7, 3 and 1.

Example Code. See `java-code-tree` handed out in class.

13.6 Algorithms for Graph Traversal

Problems that can be expressed and solved in terms of search over a graph include:

- Finding the best path through a graph (for routing and map directions)
- Topologically sorting a graph.
- Determining whether a graph is an directed acylic graph (DAG).
- Finding all reachable nodes.

Algorithms for graph traversal build upon those for tree traversal, and add features for the presence/absence of loops. In an undirected graph we follow all edges; in a directed graph we follow only out-edges.

DAG's do not have loops. Understanding whether or not a graph contains loops can simplify the evaluation of computations (e.g., in spreadsheet analysis) because DAGs can be traversed in one pass. Spreadsheet graphs that contain dependency relatins that form loops need to be updated via iteration.

In engineering, lists of reachable nodes are important for studies involving cause-and-effect, traceability, and disaster/resiliency analysis. In computer science, lists of reachable nodes are used for garbage collection, and for search strategies for single- and multi-player games.

Tricolor Algorithm

Abstractly, graph traversal can be expressed in terms of the tricolor algorithm due to Dijkstra and others.

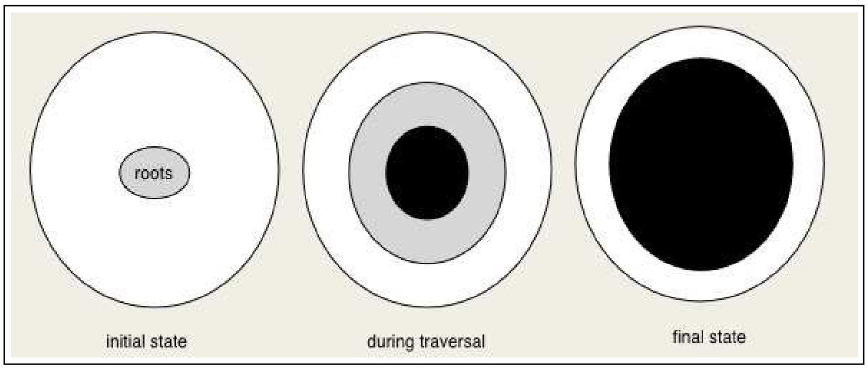


Figure 13.19. Algorithm progress as depicted by initial (white), in traversal (grey) and final (black) states.

In this algorithm, graph nodes are assigned one of three colors that can change over time:

- White nodes are undiscovered nodes that have not been seen yet in the current traversal and may even be unreachable.
- Black nodes are nodes that are reachable and that the algorithm is done with.

- Gray nodes are nodes that have been discovered but that the algorithm is not done with yet. These nodes are on a frontier between white and black.

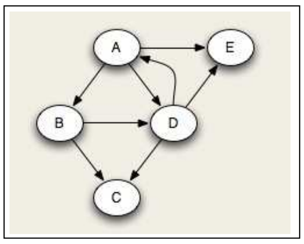
The progress of an algorithm is depicted as follows: Initially there are no black nodes and the roots are gray. As the algorithm progresses, white nodes turn into gray nodes and gray nodes turn into black nodes. Eventually there are no gray nodes left and the algorithm is done.

Breath-First Graph Search

The breadth-first search (BFS) algorithm ...

> systematically explores nodes in the order of their distance from the roots, where distance is defined as the minimum path length from a root to the node.

Algorithm pseudo-code. Let s be the source node. The BFS algorithms proceeds as follows:



```
1. frontier = new Queue()
2. mark root visited (set root.distance = 0)
3. frontier.push(root)
4. while frontier not empty {
      Vertex v = frontier.pop()
      for each successor v’ of v {
         if v’ unvisited {
            frontier.push(v’)
     mark v’ visited (v’.distance = v.distance + 1)
         }
      }
   }
```

In BFS, white nodes are those not marked as visited, gray nodes are those marked as visited and are in fronter, and the black nodes are visited nodes no longer in frontier. Instead of using a visited flag, we can keep track of a node’s distance in the field v.distance. When a new node is discovered, its distance is set to be one greater than its predecessor v.

Small Example. Running this algorithm on the adjacent graph generates the following sequence of nodes passing through the queue:

```
A0  B1  D1  E1  C2
```

Each node is annotated by its minimum distance from the source node A. Note that we’re pushing onto the right of the queue and popping from the left. Nodes are popped in distance order:

```
A, B, D, E, C.
```

Note. This strategy is useful for problems involving examination of the shortest path through the graph to something.

Depth-First Graph Search

The depth-first search (DFS) strategy:

> starts at the root (selecting some node as the root in the graph case) and explores as far as possible (i.e., deeper and deeper) along each branch until it hits a node that has no children. The search then backtracks, returning to the most recent node it hasn't finished exploring.

Algorithm pseudo-code.

```
DFS(Vertex v) {
   mark v visited
   set color of v to gray
   for each successor v′ of v {
      if v′ not yet visited {
         DFS(v′)
      }
   }
   set color of v to black
}
```

This algorithm can be thought of as a person walking through the graph, following arrows until a dead end is reached. The algorithm never visits a node twice except when backtracking.

Small Example. Running this code on the small test graph

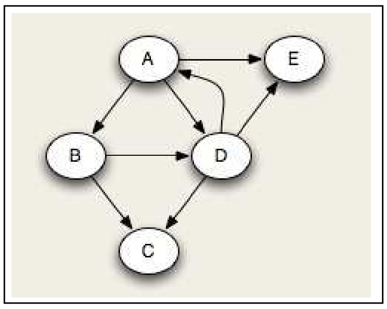


Figure 13.20. Small test graph.

yields the sequence of graph colorings shown in Figure 13.21.

Notice that at any given time there is a single path of gray nodes leading from the starting node and leading to the current node v. This path corresponds to the stack in the earlier implementation, although the nodes end up being visited in a different order because the recursive algorithm only marks one successor gray at a time.

Also notice that if we want to reach the whole graph then a single graph search might not be enough.

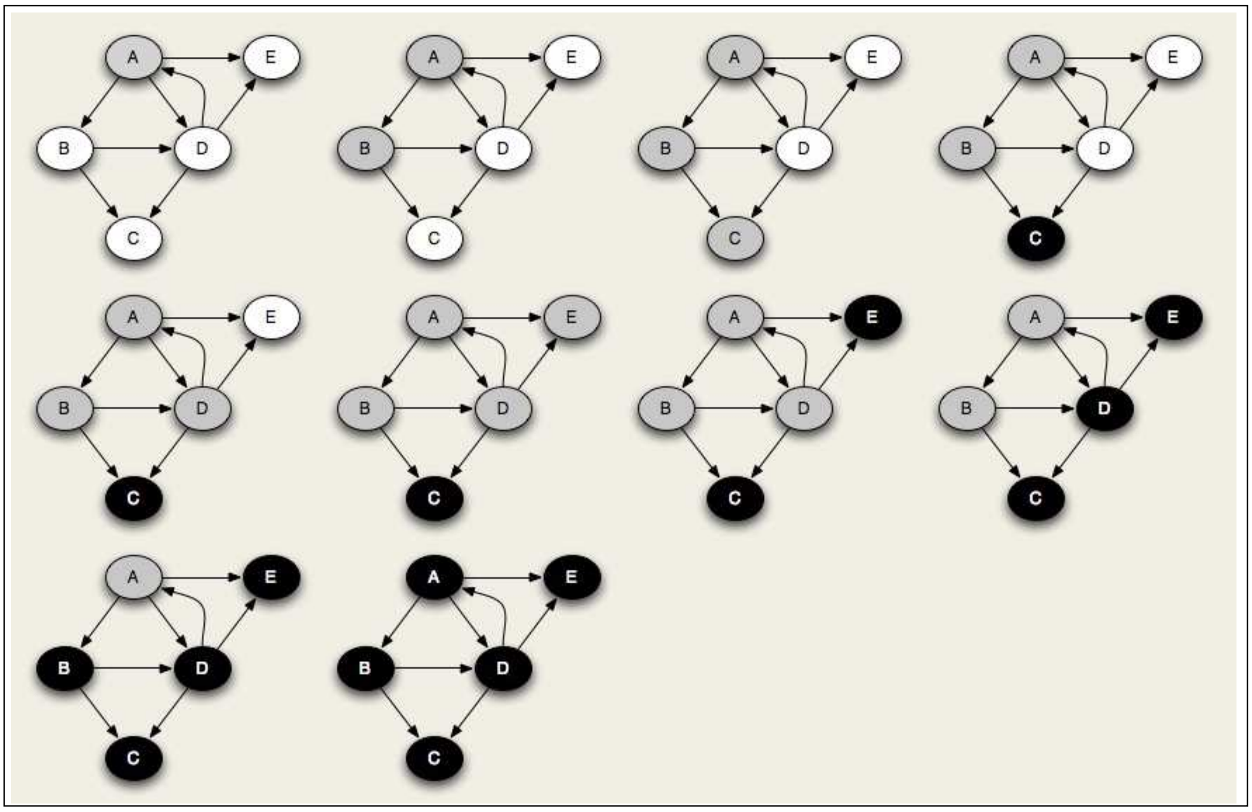


Figure 13.21. Step-by-step traversal of graph with depth first search.

13.7 Liang's Graph Package (Searching Graphs)

Example 2. Breadth-First Search

Problem Statement. In this example we exercise a breath-first search routine on the graph shown in Figure 13.16 with "Chicago" selected as the root node in the graph.

File: src/demo/TestBFS.java.

source code

```
/*
 *  ========================================================================
 *  TestBFS.java: Test breadth first search routine ..
 *  ========================================================================
 */

package demo;

import liang.model.graph.*;

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}
     };

     System.out.println("Run TestBFS()                            ");
     System.out.println("========================================");

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

     java.util.List<Integer> searchOrders = bfs.getSearchOrders();
     System.out.println(bfs.getNumberOfVerticesFound() +
         " vertices are searched in this order:");
     for (int i = 0; i < searchOrders.size(); i++)
         System.out.println(graph.getVertex( searchOrders.get(i)) );
     for (int i = 0; i < searchOrders.size(); i++)
        if (bfs.getParent(i) != -1)
           System.out.println("parent of " + graph.getVertex(i) +
                              " is " + graph.getVertex(bfs.getParent(i)));

     System.out.println("=========================================");
  }
}
```

The program output is:

```
Script started on Mon Apr  9 17:42:03 2012
prompt >> ant run03
Buildfile: /Users/austin/ence688r.d/java-code-graphs/liang/build.xml

compile:
run03:
     [java] Run TestBFS()
     [java] =========================================
     [java] 12 vertices are searched in this order:
     [java] Chicago
     [java] Seattle
     [java] Denver
     [java] Kansas City
     [java] Boston
     [java] New York
     [java] San Francisco
     [java] Los Angeles
     [java] Atlanta
     [java] Dallas
     [java] Miami
     [java] Houston
     [java] parent of Seattle is Chicago
     [java] parent of San Francisco is Seattle
     [java] parent of Los Angeles is Denver
     [java] parent of Denver is Chicago
     [java] parent of Kansas City is Chicago
     [java] parent of Boston is Chicago
     [java] parent of New York is Chicago
     [java] parent of Atlanta is Kansas City
     [java] parent of Miami is Atlanta
     [java] parent of Dallas is Kansas City
     [java] parent of Houston is Atlanta
     [java] =========================================

BUILD SUCCESSFUL
Total time: 1 second
prompt >>
prompt >> exit
Script done on Mon Apr  9 17:42:11 2012
```

Example 3. Depth First Search

Problem Statement. In this example we conduct a depth-first search of the graph shown in Figure 13.16 with "Chicago" selected as the root node in the graph.

File: src/demo/TestDFS.java.

In this example we exercise the depth-first-search routine.

source code

```
/*
 *  ========================================================================
 *  TestDFS.java: Test depth-first-search routine.....
 *  ========================================================================
 */

package demo;

import liang.model.graph.*;

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}
     };

     System.out.println("Run TestDFS()                            ");
     System.out.println("========================================");

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

     java.util.List<Integer> searchOrders = dfs.getSearchOrders();
     System.out.println(dfs.getNumberOfVerticesFound() +
                   " vertices are searched in this DFS order:");

     for (int i = 0; i < searchOrders.size(); i++)
          System.out.print(graph.getVertex(searchOrders.get(i)) + " ");
     System.out.println();

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

     System.out.println("=========================================");
  }
}
```

The program output is:

```
Script started on Mon Apr  9 18:25:36 2012
prompt >>
prompt >> ant run02
Buildfile: /Users/austin/ence688r.d/java-code-graphs/liang/build.xml

run02:
     [java] Run TestDFS()
     [java] =========================================
     [java] 12 vertices are searched in this DFS order:
     [java] Chicago
            Seattle
            San Francisco
            Los Angeles
            Denver
            Kansas City
            New York
            Boston
            Atlanta
            Miami
            Houston
            Dallas
     [java] parent of Seattle is Chicago
     [java] parent of San Francisco is Seattle
     [java] parent of Los Angeles is San Francisco
     [java] parent of Denver is Los Angeles
     [java] parent of Kansas City is Denver
     [java] parent of Boston is New York
     [java] parent of New York is Kansas City
     [java] parent of Atlanta is New York
     [java] parent of Miami is Atlanta
     [java] parent of Dallas is Houston
     [java] parent of Houston is Miami
     [java] =========================================

BUILD SUCCESSFUL
Total time: 1 second
prompt >> exit
Script done on Mon Apr  9 18:25:50 2012
```

Example 4. Weighted Graph Algorithms

File: src/demo/TestWeightedGraph.java.

In this example we add weights – geographical distance between cities – to the graph edges.

source code

```
/*
 *  ======================================================================
 *  TestWeightedGraph.java: Test routine weighted graphs ....
 *  ======================================================================
 */

package demo;

import liang.model.graph.*;
import liang.view.*;

public class TestWeightedGraph {
   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,   807}, {0, 3, 1331}, {0, 5, 2097},
                        {1, 0,   807}, {1, 2,  381}, {1, 3, 1267},
                        {2, 1,   381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},
                        {3, 0,  1331}, {3, 1, 1267},
                        {3, 2,  1015}, {3, 4,  599}, {3, 5, 1003},
                        {4, 2,  1663}, {4, 3,  599}, {4, 5, 533},
                        {4, 7,  1260}, {4, 8,  864}, {4, 10, 496},
                        {5, 0,  2097}, {5, 3, 1003}, {5, 4, 533},
                        {5, 6,   983}, {5, 7,  787},
                        {6, 5,   983}, {6, 7,  214},
                        {7, 4,  1260}, {7, 5,  787}, {7, 6, 214}, {7, 8, 888},
                        {8, 4,   864}, {8, 7,  888},
                        {8, 9,   661}, {8, 10, 781}, {8, 11, 810},
                        {9, 8,   661}, {9, 11, 1187},
                        {10, 2, 1435}, {10, 4,  496}, {10, 8, 781}, {10, 11, 239},
                        {11, 8,  810}, {11, 9, 1187}, {11, 10, 239}
      };

      System.out.println("Run TestWeightedGraph()                  ");
      System.out.println("========================================");

      WeightedGraph<String> graph1 = new WeightedGraph<String>(edges, vertices);

      System.out.println("The number of vertices in graph1: " + graph1.getSize());
      System.out.println("The vertex with index 1 is " + graph1.getVertex(1));
      System.out.println("The index for Miami is " + graph1.getIndex("Miami"));
      System.out.println("The edges for graph1:");
      System.out.println("===================:");
      graph1.printWeightedEdges();
      System.out.println("===================");

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

      WeightedGraph<Integer> graph2 = new WeightedGraph<Integer>(edges, 5);
      System.out.println("\nThe edges for graph2:");
      graph2.printWeightedEdges();

      System.out.println("=========================================");
  }
}
```

The program output is:

```
Script started on Mon Apr  9 18:31:13 2012
prompt >> ant run04
Buildfile: /Users/austin/ence688r.d/java-code-graphs/liang/build.xml

compile:
    [javac] /Users/austin/ence688r.d/java-code-graphs/liang/build.xml:9:
    warning: 'includeantruntime' was not set,
    defaulting to build.sysclasspath=last; set to false for repeatable builds

run04:
     [java] Run TestWeightedGraph()
     [java] =========================================
     [java] The number of vertices in graph1: 12
     [java] The vertex with index 1 is San Francisco
     [java] The index for Miami is 9
     [java] The edges for graph1:
     [java] ===================:
     [java] Vertex 0: (0, 1, 807) (0, 3, 1331) (0, 5, 2097)
     [java] Vertex 1: (1, 2, 381) (1, 0, 807) (1, 3, 1267)
     [java] Vertex 2: (2, 1, 381) (2, 3, 1015) (2, 4, 1663) (2, 10, 1435)
     [java] Vertex 3: (3, 4, 599) (3, 5, 1003) (3, 1, 1267) (3, 0, 1331) (3, 2, 1015)
     [java] Vertex 4: (4, 10, 496) (4, 8, 864) (4, 5, 533) (4, 2, 1663) (4, 7, 1260) (4, 3, 5
     [java] Vertex 5: (5, 4, 533) (5, 7, 787) (5, 3, 1003) (5, 0, 2097) (5, 6, 983)
     [java] Vertex 6: (6, 7, 214) (6, 5, 983)
     [java] Vertex 7: (7, 6, 214) (7, 8, 888) (7, 5, 787) (7, 4, 1260)
     [java] Vertex 8: (8, 9, 661) (8, 10, 781) (8, 4, 864) (8, 7, 888) (8, 11, 810)
     [java] Vertex 9: (9, 8, 661) (9, 11, 1187)
     [java] Vertex 10: (10, 11, 239) (10, 4, 496) (10, 8, 781) (10, 2, 1435)
     [java] Vertex 11: (11, 10, 239) (11, 9, 1187) (11, 8, 810)
     [java] ===================
     [java]
     [java] The edges for graph2:
     [java] Vertex 0: (0, 1, 2) (0, 3, 8)
     [java] Vertex 1: (1, 0, 2) (1, 2, 7) (1, 3, 3)
     [java] Vertex 2: (2, 3, 4) (2, 1, 7) (2, 4, 5)
     [java] Vertex 3: (3, 1, 3) (3, 4, 6) (3, 2, 4) (3, 0, 8)
     [java] Vertex 4: (4, 2, 5) (4, 3, 6)
     [java] =========================================

BUILD SUCCESSFUL
Total time: 1 second
prompt >> exit
Script done on Mon Apr  9 18:31:20 2012
```

13.8 Horstmann's Framework for Selectable Shapes

In this section we develop code to create a scene of selectable compound shapes. Figure 13.22 shows, for example, a scene composed of car, house, rectangle, circle, polygon and rectangular track shapes.

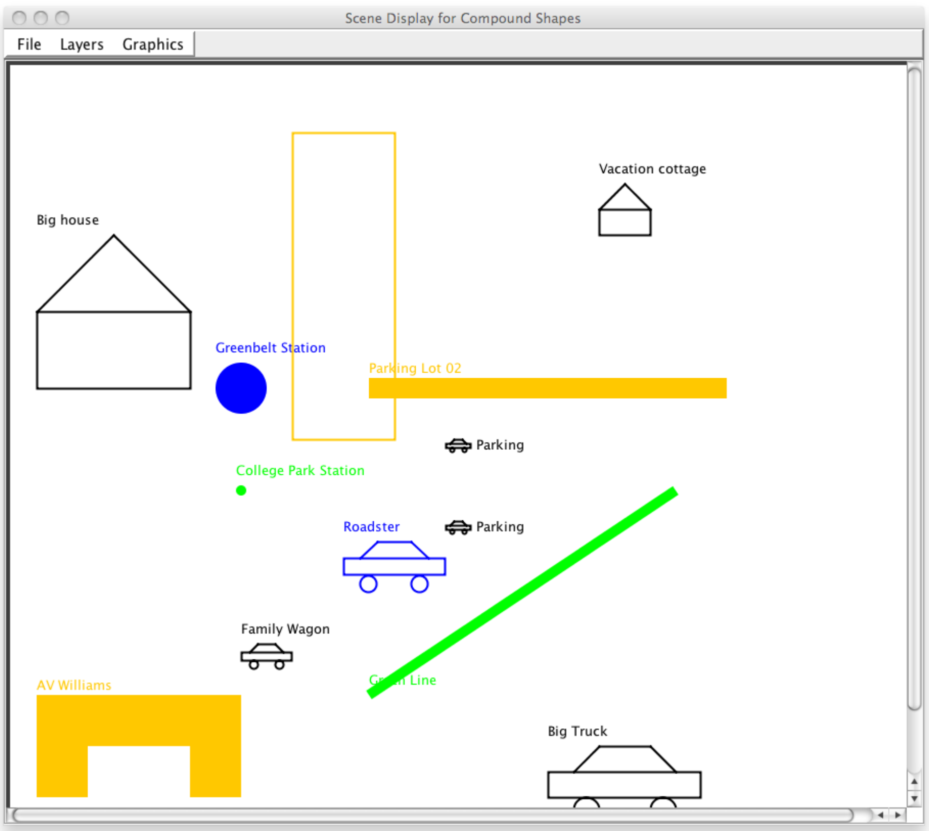


Figure 13.22. Scene of selectable compound shapes.

Source code: Can be downloaded as a zip file. Go to the class web page, click on the Swing Examples link, and then scoot to the bottom of the web page.

The source code can be conveniently organized into three groups: a generic shape interface, selectable shape abstractions, and compound shape instances.

```
=====================================================================
Shape Abstractions                       Shape Interfaces
------------------------                 ------------------
SelectableShape.java                     SceneShape.java
CompoundShape.java

Compound Shape Instances
------------------------

CarShape.java           HouseShape.java          RectangleShape.java
```

```
CircleShape.java        PolygonShape.jav         TrackShape.java
===================================================================
```

The relationship among classes for selectable compound shapes is as follows:

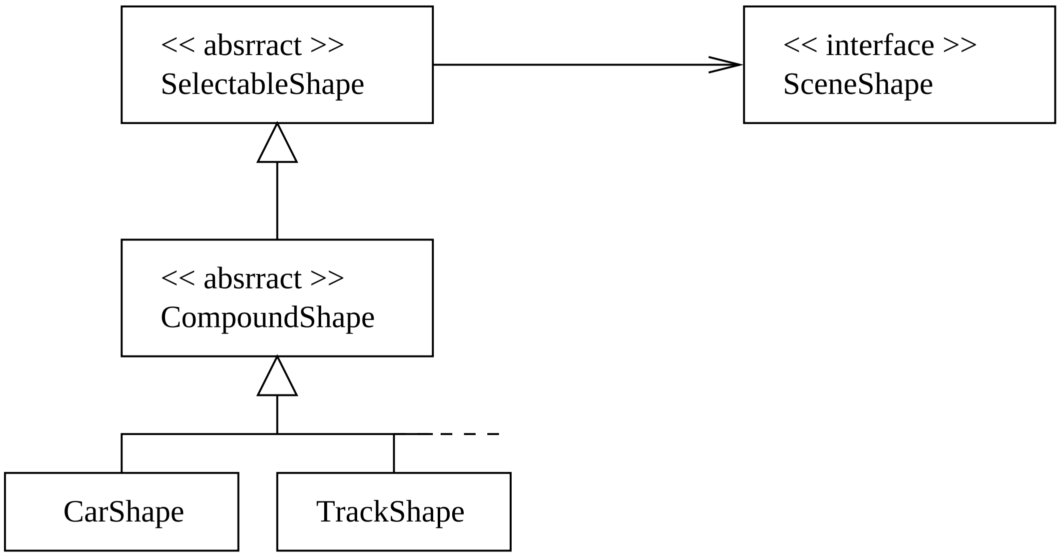


Figure 13.23. Relationship among classes for selectable compound shapes.

Part 1. SceneShape Abstractions and Interface

SceneShape.java. The sceneshape interface contains a list of declarations for methods that will need to be implemented by the hierarchy of abstract classes SelectableShape.java and CompoundShape.java.

source code

```
/*
 * ====================================================
 * Sceneshape.java: Interface to part of a scene...
 * ====================================================
 */

package shape;

import java.awt.*;
import java.awt.geom.*;

public interface SceneShape {

   // Draws this item.

   void draw(Graphics2D g2);

   // Draws the selection adornment of this item.
   void drawSelection(Graphics2D g2);

   // Sets the selection state of this item.

   void setSelected(boolean b);

   // Gets the selection state of this item.

   boolean isSelected();

   // Translates this item by a given amount.

   void translate( int dx, int dy );

   // Tests whether this item contains a given point.

   boolean contains(Point2D p);
   String  getName();
}
```

Notice that separate methods are provided for drawing the shape, and then redrawing the shape after it has been selected. The contains() method will determine whether or not a coordinate point is contained within the compound shape.

SelectableShape.java. This class manages the selection state of a compound shape.

source code

```
/*
 * ==================================================================
 * SelectableShape.java:  A shape that manages its selection state.
 * ==================================================================
 */

package shape;

import java.awt.*;
import java.awt.geom.*;

public abstract class SelectableShape implements SceneShape {
   private boolean selected;

   public void setSelected(boolean b) {
      selected = b;
   }

   public boolean isSelected() {
      return selected;
   }

   public void drawSelection(Graphics2D g2) {
      translate(1, 1);
      draw(g2);
      translate(1, 1);
      draw(g2);
      translate(-2, -2);
   }
}
```

CompoundShape.java. A compound scene shape that is composed of multiple geometric shapes.

source code

```
/**
  * ==================================================================================
  * CompoundShape.java: A scene shape that is composed of multiple geometric shapes.
  * ==================================================================================
  */

package shape;

import java.awt.*;
import java.awt.geom.*;
import java.awt.BasicStroke;
import java.awt.Color;

public abstract class CompoundShape extends SelectableShape {
   private Color    color = null;        // Color of compound shape ...
   private Color oldColor = Color.black; // Default color is black ...
   protected String sName = null;        // Name of compound shape ....
   protected double     x, y;  // Anchor point for compound shape ...
   protected double    width;  // Width  of compound shape ....
   protected double   height;  // Height of compound shape ....
   private GeneralPath  path;  // General path ...
   protected int textOffSetX =   0; // Default x- offset for text label.
   protected int textOffSetY = -10; // Default y- offset for text label.
   boolean filledShape = false;
   boolean shapeActive = false;

   public CompoundShape() {
      path = new GeneralPath();
   }

   protected void add(Shape s) {
      path.append(s, false);
   }

   public boolean contains( Point2D pt ) {
      boolean insideX = ( pt.getX() > x ) && ( pt.getX() < (x + width));
      boolean insideY = ( pt.getY() < y ) && ( pt.getY() > (y - height));

      // Set shapeActive to true when cursor is inside shape ...

      if ( insideX && insideY == true )
         shapeActive = true;
      else
         shapeActive = false;

      return ( insideX && insideY );
   }

   public void translate(int dx, int dy) {
      path.transform( AffineTransform.getTranslateInstance(dx, dy) );
   }

   // Set/get name of compound shape ....

   public void setName ( String sName ) {
      this.sName = sName;
   }

   public String getName () {
      return sName;
   }

   // Set x- and y-offsets for text string ...

   public void setTextOffSetX( int textOffSetX ) {
      this.textOffSetX = textOffSetX;
   }

   public void setTextOffSetY( int textOffSetY ) {
      this.textOffSetY = textOffSetY;
   }

   // Set filled shape flag ....

   public void setColor( Color color ) {
      this.color = this.oldColor = color;
   }

   // Set filled shape flag ....

   public void setFilledShape ( boolean filledShape ) {
      this.filledShape = filledShape;
   }

   // Draw shape ....

   public void draw( Graphics2D g2D ) {
      g2D.translate(  x,  y );
      g2D.scale( 1.0, -1.0 );

      // Highlight shape when "cursor over shape" ....

      if (shapeActive == false)
          g2D.setColor( oldColor );
      else {
          g2D.setColor( Color.red );
      }
      // Set flag for filled shape ...

      if (filledShape == false)
          g2D.draw(path);
      else
          g2D.fill(path);

      if (shapeActive == false)
          g2D.setColor( oldColor );

      // Add label to shape ....

      if (sName != null)
          g2D.drawString( sName, textOffSetX, textOffSetY );

      // Reset affine transformation ....

      g2D.scale( 1.0, -1.0 );
      g2D.translate( -x, -y );
   }
}
```

A first cut implementation of contains() is provided – the boundary of the compound shape is roughly represented by a rectangle

TrackShape.java We define track shape segments as a means for representing edges in transportation graphs.

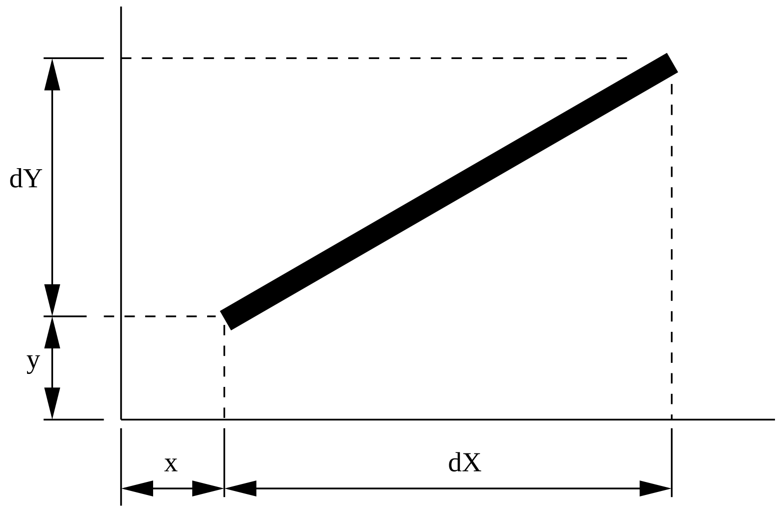


Figure 13.24. Schematic for the definition of a track shape element.

As illustrated in Figure 13.24, elements are defined by an (x,y) anchor point plus offsets dX and dY.

source code

```
/**
  *  =====================================================================
  *  TrackShape.java: Construct track shape element ...
  *  =====================================================================
  */

package shape;

import java.util.*;
import java.awt.*;
import java.awt.geom.*;

public class TrackShape extends CompoundShape {
   GeneralPath star = new GeneralPath();

   public TrackShape( int x, int y, double dX, double dY, double width ) {
      double dX0, dY0;
      double dX1, dY1;
      double dX2, dY2;
      double dX3, dY3;

      // Save corner reference point for shape ...

      this.x = (double) x;
      this.y = (double) y;

      // Compute orientation of the track element ...

      double dAngle = getAngle( dX, dY );

      // Set the initial coordinate of the GeneralPath

      dX0 =  (width/2.0)*Math.sin(dAngle);
      dY0 = -(width/2.0)*Math.cos(dAngle);

      dX1 = -(width/2.0)*Math.sin(dAngle);
      dY1 =  (width/2.0)*Math.cos(dAngle);

      dX2 = dX - (width/2.0)*Math.sin(dAngle);
      dY2 = dY + (width/2.0)*Math.cos(dAngle);

      dX3 = dX + (width/2.0)*Math.sin(dAngle);
      dY3 = dY - (width/2.0)*Math.cos(dAngle);

      // Create the track polygon ....

      star.moveTo( dX0, dY0 );
      star.lineTo( dX1, dY1 );
      star.lineTo( dX2, dY2 );
      star.lineTo( dX3, dY3 );
      star.closePath();

      add(star);
   }
   // Compute angle for coordinates in four quadrants ....

   public double getAngle( double dX, double dY ) {
       double angle = 0.0;

       if ( dY >= 0.0 && dX >= 0.0 )
            angle = Math.atan( dY/dX );
       if ( dY >= 0.0 && dX < 0.0 )
            angle = Math.PI + Math.atan( dY/dX );
       if ( dY  < 0.0 && dX < 0.0 )
            angle = Math.PI + Math.atan( dY/dX );
       if ( dY  < 0.0 && dX >= 0.0 )
            angle = 2*Math.PI + Math.atan( dY/dX );

       return angle;
   }

   // Test to see if a point is contained within the track element.

   public boolean contains( Point2D pt ) {
      double xCoord = pt.getX() - this.x;
      double yCoord = this.y - pt.getY();
      Point2D.Double testPoint = new Point2D.Double( xCoord, yCoord );

      // Set shapeActive to true when cursor is inside shape ...

      if ( star.contains ( testPoint ) == true )
         shapeActive = true;
      else
         shapeActive = false;

      return star.contains ( testPoint );
   }
}
```

From a composite shape point of view, the track shape is modeled as a general path, i.e.,

```
   GeneralPath star = new GeneralPath();
```

plus a sequence of path moves, i.e.,

```
   star.moveTo( dX0, dY0 );
   star.lineTo( dX1, dY1 );
   star.lineTo( dX2, dY2 );
   star.lineTo( dX3, dY3 );
   star.closePath();
```

Finally, the star general path is added to the list of paths defined in the compound shape. A similar procedure is used for the assembly of HouseShapes, CarShapes, and so forth.

Part 2. SceneShape Assembly

Figure 13.25 shows a class diagram for assembling and displaying a scene of compound shapes.

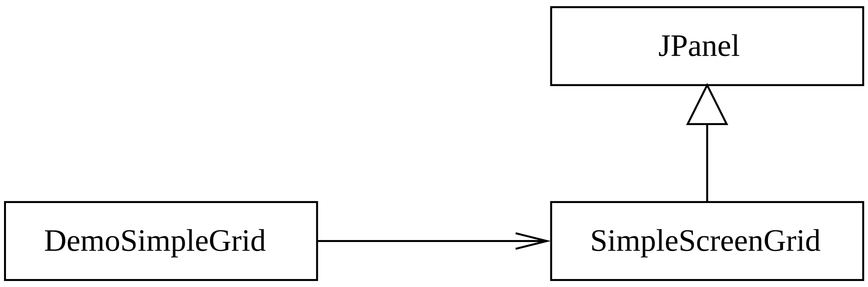


Figure 13.25. Class hierarchy for a scene of compound shapes.

DemoSimpleGrid creates the frame. It calls SimpleScreen to create and populate a panel of compound shapes.

DemoSimpleGrid.java

source code

```
/**
  *  ======================================================================
  *  DemoSimpleGrid.java. Create simple GUI with panel ...
  *
  *  Written By : Mark Austin                                   October 2011
  *  ======================================================================
  */

import java.lang.Math.*;
import java.applet.*;
import java.util.*;
import java.io.*;
import java.text.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.Color;
import java.awt.font.*;
import java.awt.image.*;
import java.awt.geom.*;   // Needed for affine transformation....
import java.net.URL;

public class DemoSimpleGrid {
    public static void main( String args[] ) {

       // Create graphics screen ....

       SimpleScreen canvas = new SimpleScreen();
       canvas.setBackground( Color.white );
```

```
        // 5. Create a scroll pane and add the panel to it.

        JScrollPane scrollCanvas = new JScrollPane( canvas,
                      JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
                      JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);

        // Create menu toolbar ...

        JToolBar toolBar = new SimpleScreenToolBar( canvas );

        // Create buttons for panel along bottom of screen ...

        final int NoButtons = 3;
        JButton buttons[] = new JButton [ NoButtons ];

        buttons[0] = new JButton ("Clear");
        buttons[0].addActionListener( new ButtonAction( buttons[0], canvas ));
        buttons[1] = new JButton ("Switch Grid");
        buttons[1].addActionListener( new ButtonAction( buttons[1], canvas ));
        buttons[2] = new JButton ("Border");
        buttons[2].addActionListener( new ButtonAction( buttons[2], canvas ));

        // Create panel. Add buttons to panel.

        JPanel p1 = new JPanel();
        for(int ii = 1; ii <= NoButtons; ii++ )
            p1.add( buttons[ii-1] );

        JPanel panel = new JPanel();
        panel.setLayout( new BorderLayout() );
        panel.add(      toolBar, BorderLayout.NORTH );
        panel.add( scrollCanvas, BorderLayout.CENTER );

        JFrame frame = new JFrame("Scene Display for Compound Shapes");
        frame.getContentPane().setLayout( new BorderLayout() );
        frame.getContentPane().add( panel );
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setSize( 900, 800);
        frame.setVisible(true);
    }
}

/*
 *  =================================================================
 *  This class listens for action events associated with the buttons.
 *  =================================================================
 */

class ButtonAction implements ActionListener {
      private JButton b;
      private SimpleScreen gs;

      public ButtonAction ( JButton b, SimpleScreen gs ) {
         this.b  = b;
         this.gs = gs;
      }

      public void actionPerformed ( ActionEvent e ) {
         String s = new String( e.getActionCommand() );

         // Clear Screen ....

         if( s.compareTo("Clear") == 0 ) { gs.clearScreen(); }

         // Draw Grid ....

         if( s.compareTo("Switch Grid") == 0 ) {
             if ( gs.grop.getGrid() == false )
                  gs.grop.setGrid( true );
             else
                  gs.grop.setGrid( false );

             gs.paint();
         }

         // Draw border around region ...

         if( s.compareTo("Border") == 0 ) {
             if ( gs.grop.getBorder() == false )
                  gs.grop.setBorder( true );
             else
                  gs.grop.setBorder( false );

             gs.paint();
         }
      }
}
```

SimpleScreen.java. SimpleScreen extends JPanel to create a panel. It then creates an arraylist of compound shapes (i.e., via shapes = new ArrayList¡SceneShape¿();).

source code

```
/**
  *  ======================================================================
  *  SimpleScreen.java. Create simple GUI with panel ...
  *
  *  Written By : Mark Austin                                  October 2011
  *  ======================================================================
  */

import java.lang.Math.*;
import java.applet.*;
import java.util.*;
import java.io.*;
import java.text.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.Color;
import java.awt.font.*;
import java.awt.image.*;
import java.awt.geom.*;   // Needed for affine transformation....
import java.net.URL;

public class SimpleScreen extends JPanel {
    private AffineTransform            at; // save grid transform ....
    private ArrayList<SceneShape>  shapes;
    GraphicsGrid      grid;
    GraphicsOperation grop;
    private Dimension size;
    private Graphics    gs;
    int      width, height;
    boolean  DEBUG = false;

    // Matrix transform for pixel to viewpoint coordinates ....

    double[] theMatrix = new double[6];

    // Desired coordinate limits.

    private double MinX =  -200.0;
    private double MinY =  -200.0;
    private double MaxX =   500.0;
    private double MaxY =   500.0;

    private int xBorder     = 30;
    private int yBorder     = 30;

    boolean overShape = false;

    //  Constructor for simple screen ....

    SimpleScreen () {
       grid   = new GraphicsGrid();
       grop   = new GraphicsOperation();
       shapes = new ArrayList<SceneShape>();

       CompoundShape car01   = new CarShape(   0,    0,  50);
       car01.setName("Family Wagon");

       CompoundShape car02   = new CarShape( 100,  100, 100);
       car02.setName("Roadster");
       car02.setColor( Color.blue );

       CompoundShape car03   = new CarShape( 300, -100, 150);
       car03.setName("Big Truck");

       CompoundShape car04   = new CarShape( 200, 200,  25);
       car04.setTextOffSetX( 30 );
```

```
car04.setTextOffSetY( 10 );
car04.setName("Parking");
CompoundShape car05   = new CarShape( 200, 120,  25);
car05.setTextOffSetX( 30 );
car05.setTextOffSetY( 10 );
car05.setName("Parking");

CompoundShape house01 = new HouseShape( -200, 400, 150);
house01.setName("Big house");
house01.setTextOffSetX( 0 );

CompoundShape house02 = new HouseShape(  350, 450,  50);
house02.setName("Vacation cottage");

CompoundShape circle01 = new CircleShape(  0, 250,  50);
circle01.setName("Greenbelt Station");
circle01.setFilledShape( true );
circle01.setColor( Color.blue );

CompoundShape circle02 = new CircleShape(  0, 150,  10);
circle02.setName("College Park Station");
circle02.setFilledShape( true );
circle02.setColor( Color.green );

CompoundShape parking01 = new RectangleShape(  300, 250,  350, 20 );
parking01.setName("Parking Lot 01");
parking01.setTextOffSetX(    0 );
parking01.setTextOffSetY(   -5 );
parking01.setFilledShape( true );
parking01.setColor( Color.orange );

CompoundShape parking02 = new RectangleShape(  100, 350,  100, 300 );
parking01.setName("Parking Lot 02");
parking02.setColor( Color.orange );

// Create a generic polygon shape ....

double xCoord[] = { 0,  50,   50, 150, 150, 200,   200,    0 };
double yCoord[] = { 0,   0,  -50, -50,   0,   0,  -100, -100 };
CompoundShape polygon01 = new PolygonShape( -200, -150, xCoord, yCoord );
polygon01.setFilledShape( true );
polygon01.setColor( Color.orange );
polygon01.setName("AV Williams");
polygon01.setTextOffSetX(    0 );
polygon01.setTextOffSetY( -105 );

// Create a generic track element shape ....

double dx    =  300.0;
double dy    = -200.0;
double width =   10.0;
CompoundShape track01 = new TrackShape(  125, -50, dx, dy, width );
track01.setFilledShape( true );
track01.setColor( Color.green );
track01.setName("Green Line");
```

```
      // Build array of shapes....

      shapes.add( car01 );
      shapes.add( car02 );
      shapes.add( car03 );
      shapes.add( car04 );
      shapes.add( car05 );
      shapes.add( house01 );
      shapes.add( house02 );
      shapes.add( circle01 );
      shapes.add( circle02 );
      shapes.add( parking01 );
      shapes.add( parking02 );
      shapes.add( polygon01 );
      shapes.add( track01 );

      setPreferredSize( new Dimension( 900, 800) );
      setBackground( Color.WHITE );
      setBorder(BorderFactory.createLineBorder( Color.DARK_GRAY, 4) );

      addMouseMotionListener( new MouseMotionAdapter() {
         public void mouseMoved(MouseEvent event) {
            Point mousePoint = event.getPoint();
            boolean newOverShape = false;
            String overShapeName = null;

            // Get point from mouse moved event and covert
            // to grid coordinates ....

            double dx = mousePoint.getX();
            double dy = mousePoint.getY();
            double dVx = dx - xBorder + MinX;
            double dVy = height - yBorder - dy + MinY;

            newOverShape = false;
            for (SceneShape s : shapes) {
               if ( s.contains( new Point ( (int) dVx, (int) dVy ) ) ) {
                    newOverShape  = true;
                    overShapeName = s.getName();
               }
            }

            if ( newOverShape != overShape ) {
                 overShape = newOverShape;
                 repaint();
            }

         }
       });
   }

   public void paint() {
      Graphics g = getGraphics();
      super.paintComponent(g);
      paintComponent(g);
   }

   public void update(Graphics g) {}

   public void paintComponent(Graphics g) {
       super.paintComponent(g);
       Dimension d = getSize();
       size   = getSize();
       height = getSize().height;
       width  = getSize().width;

       Graphics2D g2D = (Graphics2D) g.create();

       g2D.setRenderingHint( RenderingHints.KEY_ANTIALIASING,
                             RenderingHints.VALUE_ANTIALIAS_ON);

       // Transform origin to lower right-hand corner ..

       at = new AffineTransform();
       at.translate(  xBorder, height - yBorder );
       at.scale( 1, -1);
       g2D.setTransform (at);

       // Translate to origin ....

       g2D.translate( -MinX, -MinY );

       // Draw grid .....

       if ( grop.getGrid() == true ) {
            grid.drawGrid( g2D, MinX, MinY, MaxX, MaxY );
       }

       // Draw border around grid .....

       if ( grop.getBorder() == true ) {
            grid.drawBorder( g2D, MinX, MinY, MaxX, MaxY );
       }

       // Draw scene shapes .....

       g2D.setStroke( new BasicStroke( 2 ));
       for (SceneShape s : shapes) {
          s.draw( g2D );
       }
   }

   // Get graphics and fill in background ....

   public void clearScreen () {
      gs = getGraphics();
      Graphics2D g2D = (Graphics2D) gs;

      g2D.setColor( Color.white );
      g2D.fillRect( 0, 0, size.width-1, size.height-1 );
   }

   // Draw border around graphics screen ....

   void drawBorder() {
      gs = getGraphics();
      Graphics2D g2D = (Graphics2D) gs;

      // Transform origin to lower right-hand corner ..

      AffineTransform at = new AffineTransform();
      at.translate(  xBorder, height - yBorder );
      at.scale( 1, -1);
      g2D.setTransform (at);

      // Translate to origin and draw border ...

      g2D.translate( -MinX, -MinY );
      grid.drawBorder( g2D, MinX, MinY, MaxX, MaxY );
   }

   // Draw grid on graphics screen ....

   void drawGrid() {
      grop.setGrid(true);
      gs = getGraphics();
      Graphics2D g2D = (Graphics2D) gs;

      // Transform origin to lower right-hand corner ..

      AffineTransform at = new AffineTransform();
      at.translate(  xBorder, height - yBorder );
      at.scale( 1, -1);
      g2D.setTransform (at);

      // Translate to origin and draw grid ...

      g2D.translate( -MinX, -MinY );
      grid.drawGrid( g2D, MinX, MinY, MaxX, MaxY );
   }
}
```

Points to note are as follows:

1. The mousemotion listener detects when the cursor enters the space of a new object – the screen is repainted with the shape highlighted.

13.9 Horstmann's Simple Graph Editor

In this section we look at the graph representation used in Horstmann's simple graph editor. See Figure 13.26.

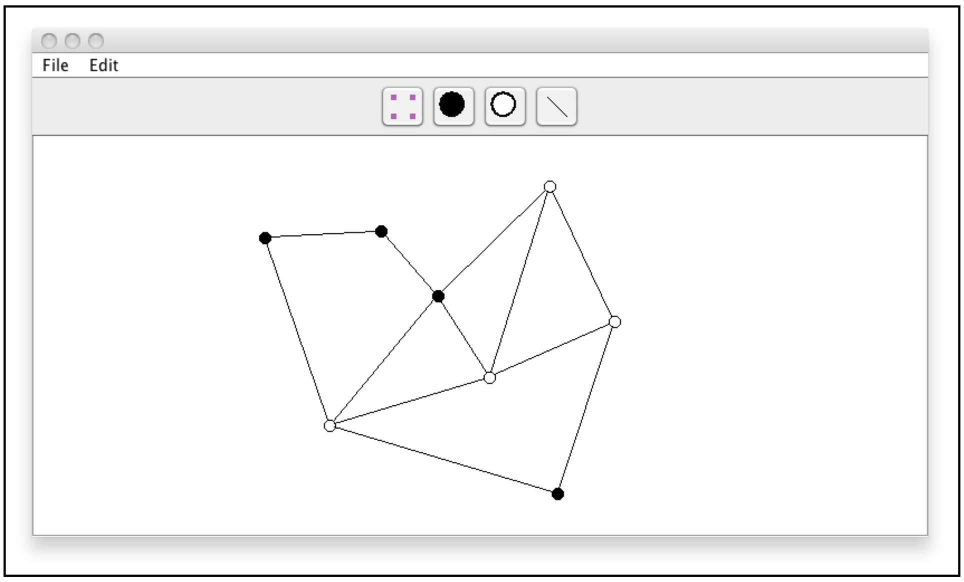


Figure 13.26. Screendump of Horstmann's graph editor.

Users and select and drag circle nodes onto the canvas and then connect nodes with straight-line edges. Inside the toolbar, the button with four small dots represents a neutral state during which the graph layout can be adjusted by selecting and dragging the nodes.

Software Architecture

We have organized the source code into a model and view hierarchy, i.e.,

```
total 0
0 drwxr-xr-x  3 austin  staff  102 Apr  9 21:24 demo
0 drwxr-xr-x  4 austin  staff  136 Feb 10 17:39 horstmann

./demo:
total 8
8 -rw-r--r--@ 1 austin  staff  506 Apr  9 21:24 SimpleGraphEditor.java

./horstmann:
total 0
0 drwxr-xr-x  4 austin  staff  136 Feb 10 17:15 model
0 drwxr-xr-x  4 austin  staff  136 Feb 10 17:43 view

./horstmann/model:
total 0
0 drwxr-xr-x  6 austin  staff  204 Apr  9 20:51 graph
```

```
0 drwxr-xr-x  5 austin  staff  170 Apr  9 21:27 simplegraph

./horstmann/model/graph:
total 40
 8 -rw-r--r--@ 1 austin  staff  1605 Apr  9 20:48 AbstractEdge.java
 8 -rw-r--r--@ 1 austin  staff  1408 Apr  9 20:49 Edge.java
16 -rw-r--r--@ 1 austin  staff  4397 Apr  9 20:50 Graph.java
 8 -rw-r--r--@ 1 austin  staff  1373 Apr  9 20:51 Node.java

./horstmann/model/simplegraph:
total 24
8 -rw-r--r--@ 1 austin  staff  2177 Apr  9 21:27 CircleNode.java
8 -rw-r--r--@ 1 austin  staff  1284 Apr  9 21:27 PointNode.java
8 -rw-r--r--@ 1 austin  staff   765 Apr  9 21:27 SimpleGraph.java

./horstmann/view/editor:
total 96
 8 -rw-r--r--@ 1 austin  staff  1049 Apr  9 20:40 EnumEditor.java
 8 -rw-r--r--@ 1 austin  staff  2372 Apr  9 20:41 FormLayout.java
16 -rw-r--r--@ 1 austin  staff  4715 Apr  9 21:16 GraphFrame.java
16 -rw-r--r--@ 1 austin  staff  7261 Apr  9 21:28 GraphPanel.java
 8 -rw-r--r--@ 1 austin  staff  1144 Apr  9 21:31 LineEdge.java
 8 -rw-r--r--@ 1 austin  staff   887 Apr  9 20:44 LineStyle.java
16 -rw-r--r--@ 1 austin  staff  6690 Apr  9 20:45 PropertySheet.java
16 -rw-r--r--@ 1 austin  staff  4567 Apr  9 21:15 ToolBar.java
```

Figure 13.27 shows the essential details among classes that contribute to the graph model.

Test Program

SimpleGraphEditor.java

source code

```
/**
  *  ============================================================
  *  SimpleGraphEditor.java: A program for creating simple graphs
  *  ============================================================
  */

package demo;

import javax.swing.*;

import horstmann.view.editor.GraphFrame;
import horstmann.model.simplegraph.SimpleGraph;

public class SimpleGraphEditor {
   public static void main(String[] args) {
      JFrame frame = new GraphFrame( new SimpleGraph() );
      frame.setVisible(true);
   }
}
```

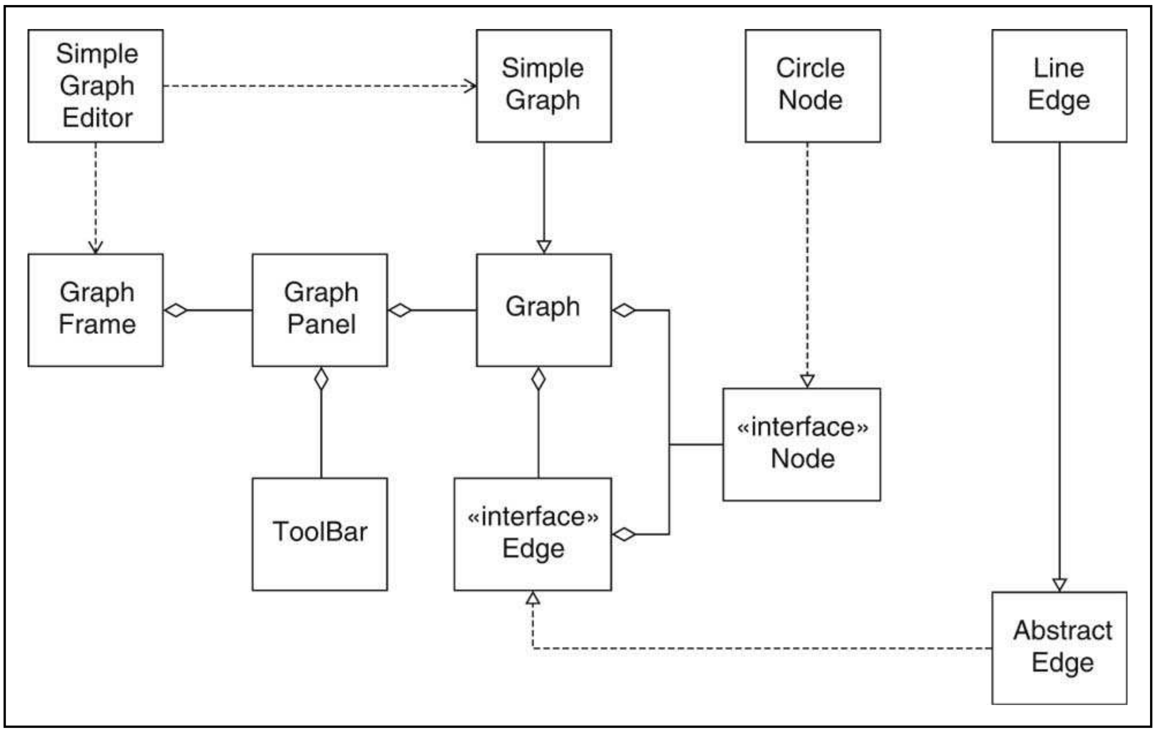


Figure 13.27. Graph classes (source: Horstmann).

Graph Model (horstmann.model.graph)

The graph model is comprised of the following classes: AbstractEdge.java, Edge.java, Graph.java and Node.java. The details are as follows:

AbstractEdge.java.

source code

```
/**
  * ======================================================================
  * AbtractEdge.java: A class that supplies convenience implementations for
  * a number of methods in the Edge interface type.
  * ======================================================================
  */

package horstmann.model.graph;

import java.awt.*;
import java.awt.geom.*;

public abstract class AbstractEdge implements Edge {
   public Object clone() {
      try {
         return super.clone();
      } catch (CloneNotSupportedException exception) {
         return null;
      }
   }

   public void connect(Node s, Node e) {
      start = s;
      end = e;
   }

   public Node getStart() {
      return start;
   }

   public Node getEnd() {
      return end;
   }

   public Rectangle2D getBounds(Graphics2D g2) {
      Line2D conn = getConnectionPoints();
      Rectangle2D r = new Rectangle2D.Double();
      r.setFrameFromDiagonal(conn.getX1(), conn.getY1(),
         conn.getX2(), conn.getY2());
      return r;
   }

   public Line2D getConnectionPoints() {
      Rectangle2D startBounds = start.getBounds();
      Rectangle2D endBounds = end.getBounds();
      Point2D startCenter = new Point2D.Double(
         startBounds.getCenterX(), startBounds.getCenterY());
      Point2D endCenter = new Point2D.Double(
         endBounds.getCenterX(), endBounds.getCenterY());
      return new Line2D.Double(
         start.getConnectionPoint(endCenter),
         end.getConnectionPoint(startCenter));
   }

   private Node start;
   private Node end;
}
```

Edge.java.

source code

```
/**
  * ====================================================
  * Edge.java: An edge in a graph.
  * ====================================================
  */

package horstmann.model.graph;

import java.awt.*;
import java.awt.geom.*;
import java.io.*;

public interface Edge extends Serializable, Cloneable {

   // Draw the edge.

   void draw(Graphics2D g2);

   // Tests whether the edge contains a point.

   boolean contains(Point2D aPoint);

   // Connects this edge to two nodes.

   void connect(Node aStart, Node anEnd);

   // Gets the starting node.

   Node getStart();

   // Gets the ending node.

   Node getEnd();
   // Gets the points at which this edge is connected to its nodes.

   Line2D getConnectionPoints();

   // Gets the smallest rectangle that bounds this edge.

   Rectangle2D getBounds(Graphics2D g2);

   Object clone();
}
```

Graph.java.

source code

```
/**
  * ==============================================================
  * Graph.java:  A graph consisting of selectable nodes and edges.
  * ==============================================================
  */

package horstmann.model.graph;

import java.awt.*;
import java.awt.geom.*;
import java.io.*;
import java.util.*;
import java.util.List;

public abstract class Graph implements Serializable {

   // Constructs a graph with no nodes or edges.

   public Graph() {
      nodes = new ArrayList<Node>();
      edges = new ArrayList<Edge>();
   }

   // Adds an edge to the graph that joins the nodes containing
   // the given points. If the points aren’t both inside nodes,
   // then no edge is added.

   public boolean connect(Edge e, Point2D p1, Point2D p2) {
      Node n1 = findNode(p1);
      Node n2 = findNode(p2);
      if (n1 != null && n2 != null)
      {
         e.connect(n1, n2);
         edges.add(e);
         return true;
      }
      return false;
   }
```

```
// Adds a node to the graph so that the top left corner of
// the bounding rectangle is at the given point.

public boolean add(Node n, Point2D p) {
   Rectangle2D bounds = n.getBounds();
   n.translate(p.getX() - bounds.getX(), p.getY() - bounds.getY());
   nodes.add(n);
   return true;
}

// Finds a node containing the given point.

public Node findNode(Point2D p) {
   for (int i = nodes.size() - 1; i >= 0; i--) {
      Node n =  nodes.get(i);
      if (n.contains(p)) return n;
   }
   return null;
}

// Finds an edge containing the given point.

public Edge findEdge(Point2D p) {
   for (int i = edges.size() - 1; i >= 0; i--)
   {
      Edge e = edges.get(i);
      if (e.contains(p)) return e;
   }
   return null;
}

// Draws the graph.

public void draw(Graphics2D g2) {
   for (Node n : nodes)
      n.draw(g2);

   for (Edge e : edges)
      e.draw(g2);
}

// Removes a node and all edges that start or end with that node

public void removeNode(Node n) {
   for (int i = edges.size() - 1; i >= 0; i--) {
      Edge e =  edges.get(i);
      if (e.getStart() == n || e.getEnd() == n)
         edges.remove(e);
   }
   nodes.remove(n);
}

// Removes an edge from the graph.
```

```
   public void removeEdge(Edge e) {
      edges.remove(e);
   }

   // Gets the smallest rectangle enclosing the graph

   public Rectangle2D getBounds(Graphics2D g2) {
      Rectangle2D r = null;
      for (Node n : nodes) {
         Rectangle2D b = n.getBounds();
         if (r == null) r = b;
         else r.add(b);
      }

      for (Edge e : edges)
         r.add(e.getBounds(g2));
      return r == null ? new Rectangle2D.Double() : r;
   }

   // Gets the node types of a particular graph type.

   public abstract Node[] getNodePrototypes();

   // Gets the edge types of a particular graph type.

   public abstract Edge[] getEdgePrototypes();

   // Gets the nodes of this graph.

   public List<Node> getNodes() {
      return Collections.unmodifiableList(nodes);
   }

   // Gets the edges of this graph.

   public List<Edge> getEdges() {
      return Collections.unmodifiableList(edges);
   }

   private ArrayList<Node> nodes;
   private ArrayList<Edge> edges;
}
```

Node.java.

source code

```
/**
  * ======================================================
  * Node.java: A node in a graph.
  * ======================================================
  */
```

```
package horstmann.model.graph;

import java.awt.*;
import java.awt.geom.*;
import java.io.*;

public interface Node extends Serializable, Cloneable {

   /**
      Draw the node.
      @param g2 the graphics context
   */

   void draw(Graphics2D g2);

   /**
      Translates the node by a given amount.
      @param dx the amount to translate in the x-direction
      @param dy the amount to translate in the y-direction
   */

   void translate(double dx, double dy);

   /**
      Tests whether the node contains a point.
      @param aPoint the point to test
      @return true if this node contains aPoint
   */

   boolean contains(Point2D aPoint);

   /**
      Get the best connection point to connect this node
      with another node. This should be a point on the boundary
      of the shape of this node.
      @param aPoint an exterior point that is to be joined
      with this node
      @return the recommended connection point
   */

   Point2D getConnectionPoint(Point2D aPoint);

   /**
      Get the bounding rectangle of the shape of this node
      @return the bounding rectangle
   */

   Rectangle2D getBounds();

   Object clone();
}
```

Simple Graph Model (horstmann.model.simplegraph)

The simple graph model is defined by three classes; CircleNode.java, PointNode.java and Simple-Graph.java.

CircleNode.java

source code

```
/**
  * ============================================================
  * CircleNode.java: A circular node that is filled with a color.
  * ============================================================
  */

package horstmann.model.simplegraph;

import java.awt.*;
import java.awt.geom.*;

import horstmann.model.graph.*;

public class CircleNode implements Node {

   public CircleNode(Color aColor) {
      size = DEFAULT_SIZE;
      x = 0;
      y = 0;
      color = aColor;
   }

   public void setColor(Color aColor) {
      color = aColor;
   }

   public Color getColor() {
      return color;
   }

   public Object clone() {
      try {
         return super.clone();
      } catch (CloneNotSupportedException exception) {
         return null;
      }
   }

   public void draw(Graphics2D g2) {
      Ellipse2D circle = new Ellipse2D.Double( x, y, size, size);
      Color oldColor = g2.getColor();
      g2.setColor(color);
      g2.fill(circle);
      g2.setColor(oldColor);
      g2.draw(circle);
   }
   public void translate(double dx, double dy) {
      x += dx;
      y += dy;
   }

   public boolean contains(Point2D p) {
      Ellipse2D circle = new Ellipse2D.Double(
            x, y, size, size);
      return circle.contains(p);
   }

   public Rectangle2D getBounds() {
      return new Rectangle2D.Double( x, y, size, size);
   }

   public Point2D getConnectionPoint(Point2D other) {
      double centerX = x + size / 2;
      double centerY = y + size / 2;
      double dx = other.getX() - centerX;
      double dy = other.getY() - centerY;
      double distance = Math.sqrt(dx * dx + dy * dy);
      if (distance == 0) return other;
      else return new Point2D.Double(
            centerX + dx * (size / 2) / distance,
            centerY + dy * (size / 2) / distance);
   }

   private double x;
   private double y;
   private double size;
   private Color color;
   private static final int DEFAULT_SIZE = 10;
}
```

PointNode.java

source code

```
/**
  * ==================================================================================
  * PointNode.java: An inivisible node that is used in the toolbar to draw an edge.
  * ==================================================================================
  */

package horstmann.model.simplegraph;

import java.awt.*;
import java.awt.geom.*;

import horstmann.model.graph.*;

public class PointNode implements Node {
   // Constructs a point node with coordinates (0, 0) ...

   public PointNode() {
      point = new Point2D.Double();
   }

   public void draw(Graphics2D g2) {}

   public void translate(double dx, double dy) {
      point.setLocation(point.getX() + dx,
         point.getY() + dy);
   }

   public boolean contains(Point2D p) {
      return false;
   }

   public Rectangle2D getBounds() {
      return new Rectangle2D.Double(point.getX(), point.getY(), 0, 0);
   }

   public Point2D getConnectionPoint(Point2D other) {
      return point;
   }

   public Object clone() {
      try
      {
         return super.clone();
      }
      catch (CloneNotSupportedException exception)
      {
         return null;
      }
   }

   private Point2D point;
}
```

SimpleGraph.java

source code

```
/**
  *  ==================================================================
  *  SimpleGraph.java: A simple graph with round nodes and straight edges.
  *  ==================================================================
  */

package horstmann.model.simplegraph;

import java.awt.*;
import java.util.*;
import horstmann.model.graph.*;
import horstmann.view.editor.*;

public class SimpleGraph extends Graph {
   public Node[] getNodePrototypes() {
      Node[] nodeTypes = { new CircleNode(Color.BLACK), new CircleNode(Color.WHITE) };
      return nodeTypes;
   }

   public Edge[] getEdgePrototypes() {
      Edge[] edgeTypes = { new LineEdge() };
      return edgeTypes;
   }
}
```

Simple Graph View Editor (horstmann.view.editor)

The simple graph view editor package is comprised of the following classes:

```
   EnumEditor.java          GraphFrame.java        LineEdge.java
   PropertySheet.java       FormLayout.java        GraphPanel.java
   LineStyle.java           ToolBar.java
```

13.10 JGraphT (Graph Representation)

JGraphT is a free Java graph library that provides mathematical graph-theory objects and algorithms. JGraphT supports various types of graphs including:

- Directed and undirected graphs.
- Graphs with weighted / unweighted / labeled or any user-defined edges.
- Various edge multiplicity options, including: simple-graphs, multigraphs, pseudographs.
- Unmodifiable graphs - allow modules to provide ”read-only” access to internal graphs.
- Listenable graphs - allow external listeners to track modification events.
- Subgraphs graphs that are auto-updating subgraph views on other graphs.
- All compositions of above graphs.

Although powerful, JGraphT is designed to be simple and type-safe (via Java generics). For example, graph vertices can be of any objects. You can create graphs based on: Strings, URLs, XML documents, etc; you can even create graphs of graphs!

JGraphT employs a combination of interfaces and abstract classes to ensure complete and consistent implementations.

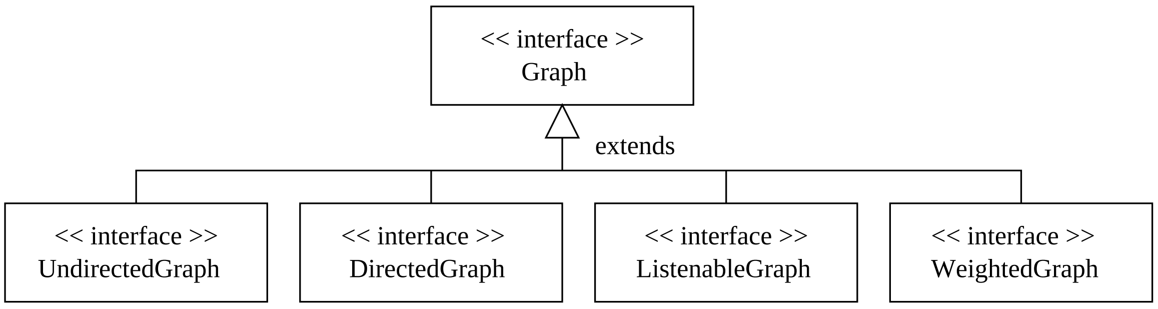
JGraphT Interface Hierarchy

1. **Graph.java.** This is the root interface in the graph hierarchy. It provides for a mathematical graph-theory graph object `G(V,E)` that contains a set `V` of vertices and a set `E` of edges. Each edge e=(v1,v2) in E connects vertex v1 to vertex v2.
2. **UndirectedGraph.java.** This is the root interface for all graphs whose all edges are undirected.
3. **DirectedGraph.java.** This is the root interface for all graphs whose all edges are directed.
4. **ListenableGraph.java.** This is the root interface for graphs that provide for listeners on structural change events.
5. **WeightedGraph.java.** This is the root interface for all graphs whose edges have non-uniform weights.

JGraphT Abstract Class Hierarchy

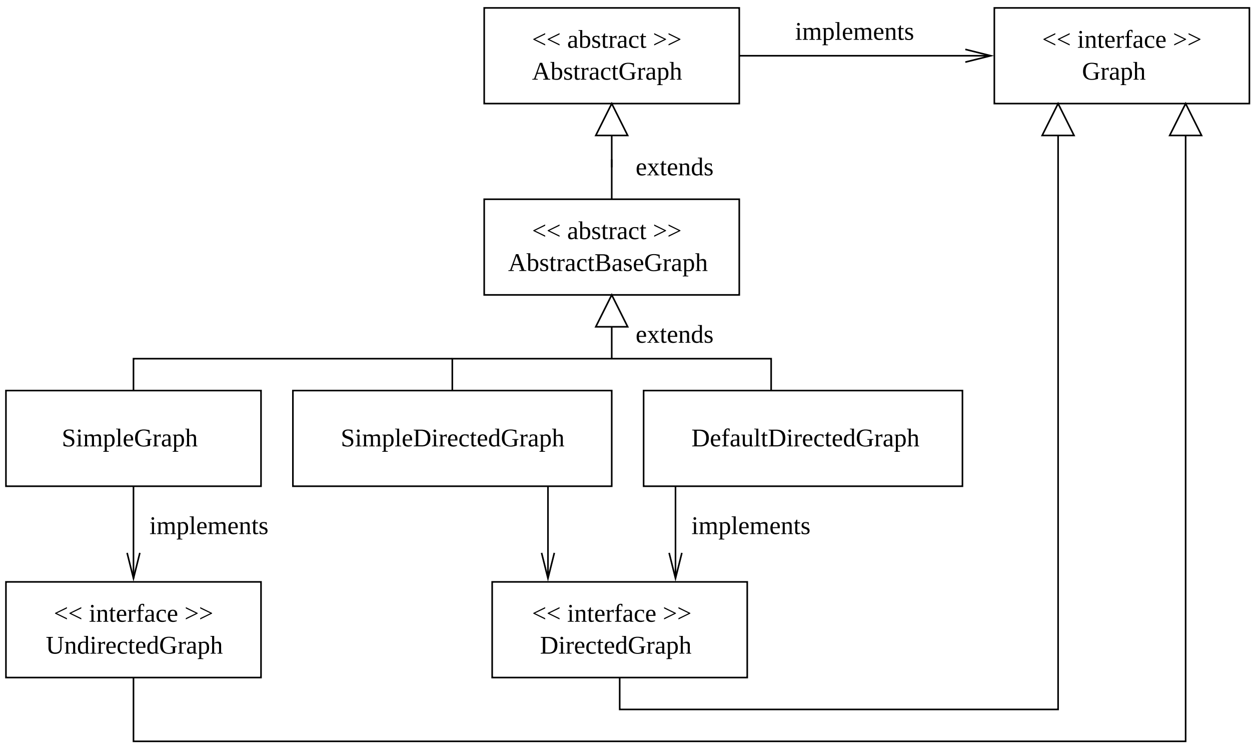
As a production library jgrapht provides a complex hierarchy of abstract classes, rooted at AbstractGraph. Part of the hierarchy includes:

1. **AbstractGraph.java.** This class provides a skeletal implementation of the Graph interface that is applicable to both directed graphs and undirected graphs.



<< interface >> Graph
extends
<< interface >> UndirectedGraph
<< interface >> DirectedGraph
<< interface >> ListenableGraph
<< interface >> WeightedGraph

Figure 13.28. Hierarchy of interface classes in JGraphT.



<< abstract >> AbstractGraph
implements
<< interface >> Graph
extends
<< abstract >> AbstractBaseGraph
extends
SimpleGraph
SimpleDirectedGraph
DefaultDirectedGraph
implements
implements
<< interface >> UndirectedGraph
<< interface >> DirectedGraph

Figure 13.29. Hierarchy of abstract classes and interface implementations in JGraphT.

2. **AbstractBaseGraph.java.** This class provides the most general implementation of the Graph interface. Subclasses of AbstractBaseGraph will add restrictions to get more specific graphs (e.g., directed graphs). This graph implementation guarantees deterministic vertex and edge set ordering (e.g., via LinkedHashMap and LinkedHashSet).

AbstractGraph provides very high-level methods for graph operations such as removal of all graph edges. No reference is made to how the vertices and edges will be stored. AbstractBaseGraph fills in these details, e.g.,

```
public abstract class AbstractBaseGraph<V, E> extends AbstractGraph<V, E>
       implements Graph<V, E>  {

    private EdgeFactory<V, E> edgeFactory;
    private EdgeSetFactory<V, E> edgeSetFactory;
    private Map<E, IntrusiveEdge> edgeMap;
    private transient Set<E> unmodifiableEdgeSet   = null;
    private transient Set<V> unmodifiableVertexSet = null;
    private Specifics specifics;

    ......

}
```

Here we see that edges and vertices will be stored via implementations of the Set interface.

Types of Graph supported in JGraphT

1. **SimpleGraph.java.** A simple graph is an undirected graph for which at most one edge connects any two vertices, and loops are not permitted.

2. **SimpleDirectedGraph.java.** A simple directed graph is a directed graph in which neither multiple edges between any two vertices nor loops are permitted.

3. **DefaultDirectedGraph.java.** A default directed graph is a non-simple directed graph that permits loops, but prohibits multiple edges between any two vertices.

3. **DefaultListenableGraph.java.** A graph backed by the the graph specified at the constructor, which can be listened to by GraphListener and by VertexSetListeners. Operations on this graph pass through to the the backing graph. Any modification made to this graph or the backing graph will be reflected by the other graph.

Details of the JgraphT Graph Interface

The root interface in the graph hierarchy. A mathematical graph-theory graph object `G(V,E)` contains a set V of vertices and a set E of edges. Each edge e=(v1,v2) in E connects vertex v1 to vertex v2. Implementation of this interface can provide simple-graphs, multigraphs, pseudographs etc.

This library works best when vertices represent arbitrary objects and edges represent the relationships between them. Vertex and edge instances may be shared by more than one graph.

Through the use of java generics, a graph can be typed to specific classes for vertices V and edges E<T>. Such a graph can contain vertices of type V and all sub-types and Edges of type E and all sub-types.

The abbreviated source code is as follows:

source code

```
/* ==========================================================
 * Graph.java: JGraphT Graph interface ...
 *
 * (C) Copyright 2003-2007, by Barak Naveh and Contributors.
 *
 * Original Author:  Barak Naveh
 * Contributor(s):   John V. Sichi
 *                   Christian Hammer
 *
 * $Id: Graph.java 568 2007-09-30 00:12:18Z perfecthash $
 * ==========================================================
 *
 */

package org.jgrapht;

import java.util.*;

public interface Graph<V, E> {

    /**
     * Returns a set of all edges connecting source vertex to target vertex if
     * such vertices exist in this graph. If any of the vertices does not exist
     * or is null, returns null. If both vertices
     * exist but no edges found, returns an empty set.
     *
     * In undirected graphs, some of the returned edges may have their source
     * and target vertices in the opposite order. In simple graphs the returned
     * set is either singleton set or empty set.
     */

    public Set<E> getAllEdges(V sourceVertex, V targetVertex);

    /**
     * Returns an edge connecting source vertex to target vertex if such
     * vertices and such edge exist in this graph. Otherwise returns
     * null. If any of the specified vertices is <code>null</code>
     * returns null
     *
     * In undirected graphs, the returned edge may have its source and target
     * vertices in the opposite order.
     */

    public E getEdge(V sourceVertex, V targetVertex);

    /**
     * Returns the edge factory using which this graph creates new edges. The
     * edge factory is defined when the graph is constructed and must not be
     * modified.
     */

    public EdgeFactory<V, E> getEdgeFactory();

    /**
     * Creates a new edge in this graph, going from the source vertex to the
     * target vertex, and returns the created edge. Some graphs do not allow
     * edge-multiplicity. In such cases, if the graph already contains an edge
     * from the specified source to the specified target, than this method does
     * not change the graph and returns <code>null</code>.
     *
     * The source and target vertices must already be contained in this
     * graph. If they are not found in graph IllegalArgumentException is
     * thrown.
     *
     * This method creates the new edge <code>e</code> using this graph’s
     * EdgeFactory. For the new edge to be added e
     * must not be equal to any other edge the graph (even if the graph
     * allows edge-multiplicity). More formally, the graph must not contain any
     * edge e2 such that e2.equals(e). If such
     * e2< is found then the newly created edge e is
     * abandoned, the method leaves this graph unchanged returns null.
     */

    public E addEdge(V sourceVertex, V targetVertex);

    /**
     * Adds the specified edge to this graph, going from the source vertex to
     * the target vertex. More formally, adds the specified edge, <code>
     * e</code>, to this graph if this graph contains no edge <code>e2</code>
     * such that <code>e2.equals(e)</code>. If this graph already contains such
     * an edge, the call leaves this graph unchanged and returns <tt>false</tt>.
     * Some graphs do not allow edge-multiplicity. In such cases, if the graph
     * already contains an edge from the specified source to the specified
     * target, than this method does not change the graph and returns <code>
     * false</code>. If the edge was added to the graph, returns <code>
     * true</code>.
     *
     * The source and target vertices must already be contained in this
     * graph. If they are not found in graph IllegalArgumentException is
     * thrown.
     */

    public boolean addEdge(V sourceVertex, V targetVertex, E e);
```

```
/**
 * Adds the specified vertex to this graph if not already present. More
 * formally, adds the specified vertex, <code>v</code>, to this graph if
 * this graph contains no vertex <code>u</code> such that <code>
 * u.equals(v)</code>. If this graph already contains such vertex, the call
 * leaves this graph unchanged and returns <tt>false</tt>. In combination
 * with the restriction on constructors, this ensures that graphs never
 * contain duplicate vertices.
 */

public boolean addVertex(V v);

/**
 * Returns true if and only if this graph contains an edge going
 * from the source vertex to the target vertex. In undirected graphs the
 * same result is obtained when source and target are inverted. If any of
 * the specified vertices does not exist in the graph, or if is
 * null, returns false.
 */

public boolean containsEdge(V sourceVertex, V targetVertex);

/**
 * Returns <tt>true</tt> if this graph contains the specified edge. More
 * formally, returns <tt>true</tt> if and only if this graph contains an
 * edge <code>e2</code> such that <code>e.equals(e2)</code>. If the
 * specified edge is <code>null</code> returns <code>false</code>.
 */

public boolean containsEdge(E e);

/**
 * Returns <tt>true</tt> if this graph contains the specified vertex. More
 * formally, returns <tt>true</tt> if and only if this graph contains a
 * vertex <code>u</code> such that <code>u.equals(v)</code>. If the
 * specified vertex is <code>null</code> returns <code>false</code>.
 */

public boolean containsVertex(V v);

/**
 * Returns a set of the edges contained in this graph. The set is backed by
 * the graph, so changes to the graph are reflected in the set. If the graph
 * is modified while an iteration over the set is in progress, the results
 * of the iteration are undefined.
 *
 * <p>The graph implementation may maintain a particular set ordering (e.g.
 * via {@link java.util.LinkedHashSet}) for deterministic iteration, but
 * this is not required. It is the responsibility of callers who rely on
 * this behavior to only use graph implementations which support it.</p>
 */

public Set<E> edgeSet();
```

```
/**
 * Returns a set of all edges touching the specified vertex. If no edges are
 * touching the specified vertex returns an empty set.
 */

public Set<E> edgesOf(V vertex);

/**
 * Removes all the edges in this graph that are also contained in the
 * specified edge collection. After this call returns, this graph will
 * contain no edges in common with the specified edges. This method will
 * invoke the {@link #removeEdge(Object)} method.
 */

public boolean removeAllEdges(Collection<? extends E> edges);

/**
 * Removes all the edges going from the specified source vertex to the
 * specified target vertex, and returns a set of all removed edges. Returns
 * <code>null</code> if any of the specified vertices does not exist in the
 * graph. If both vertices exist but no edge is found, returns an empty set.
 * This method will either invoke the {@link #removeEdge(Object)} method, or
 * the {@link #removeEdge(Object, Object)} method.
 */

public Set<E> removeAllEdges(V sourceVertex, V targetVertex);

/**
 * Removes all the vertices in this graph that are also contained in the
 * specified vertex collection. After this call returns, this graph will
 * contain no vertices in common with the specified vertices. This method
 * will invoke the {@link #removeVertex(Object)} method.
 */

public boolean removeAllVertices(Collection<? extends V> vertices);

/**
 * Removes an edge going from source vertex to target vertex, if such
 * vertices and such edge exist in this graph. Returns the edge if removed
 * or <code>null</code> otherwise.
 */

public E removeEdge(V sourceVertex, V targetVertex);

/**
 * Removes the specified edge from the graph. Removes the specified edge
 * from this graph if it is present. More formally, removes an edge <code>
 * e2</code> such that <code>e2.equals(e)</code>, if the graph contains such
 * edge. Returns <tt>true</tt> if the graph contained the specified edge.
 * (The graph will not contain the specified edge once the call returns).
 */

public boolean removeEdge(E e);

/**
```

```
     * Removes the specified vertex from this graph including all its touching
     * edges if present. More formally, if the graph contains a vertex <code>
     * u</code> such that <code>u.equals(v)</code>, the call removes all edges
     * that touch <code>u</code> and then removes <code>u</code> itself. If no
     * such <code>u</code> is found, the call leaves the graph unchanged.
     * Returns <tt>true</tt> if the graph contained the specified vertex. (The
     * graph will not contain the specified vertex once the call returns).
     */

    public boolean removeVertex(V v);

    /**
     * Returns a set of the vertices contained in this graph. The set is backed
     * by the graph, so changes to the graph are reflected in the set. If the
     * graph is modified while an iteration over the set is in progress, the
     * results of the iteration are undefined.
     */

    public Set<V> vertexSet();

    /**
     * Returns the source vertex of an edge. For an undirected graph, source and
     * target are distinguishable designations (but without any mathematical
     * meaning).
     */

    public V getEdgeSource(E e);

    /**
     * Returns the target vertex of an edge. For an undirected graph, source and
     * target are distinguishable designations (but without any mathematical
     * meaning).
     */

    public V getEdgeTarget(E e);

    /**
     * Returns the weight assigned to a given edge. Unweighted graphs return 1.0
     * (as defined by {@link WeightedGraph#DEFAULT_EDGE_WEIGHT}), allowing
     * weighted-graph algorithms to apply to them where meaningful.
     */

    public double getEdgeWeight(E e);
}
```

13.11 JGraphT (Simple Graph Operations)

Example 1. Assemble a Collection of Graph Nodes

Problem Statement. In this example we create and print a collection of “character string” nodes, i.e.,

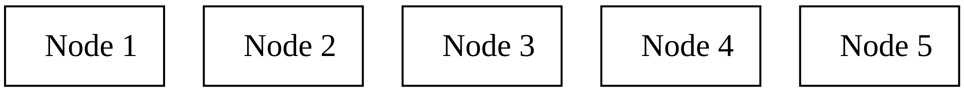


Figure 13.30. Collection of five nodes

There are no edges in this example.

File: demo/jgrapht/SimpleDirectedGraph.java.

source code

```
/*
 * ===========================================================
 * TestComponent.java. Create and print a collection of nodes.
 * ===========================================================
 */

package jgrapht;

import java.util.*;

import org.jgrapht.*;
import org.jgrapht.graph.*;
import org.jgrapht.alg.*;

public class SimpleDirectedGraph {
   public static void main(String args[]) {

      // Create a directed graph of "string" nodes ....

      DirectedGraph<String, DefaultEdge> g =
         new DefaultDirectedGraph<String,DefaultEdge>(DefaultEdge.class);

      g.addVertex("Node 1");
      g.addVertex("Node 2");
      g.addVertex("Node 3");
      g.addVertex("Node 4");
      g.addVertex("Node 5");

      System.out.println( g.toString() );
   }
}
```

Script of Input/Output. Here is a short script of input and output for the program execution.

```
Script started on Mon Apr  9 10:43:11 2012
prompt >> ant run01
Buildfile: /Users/austin/ence688r.d/java-code-graphs/build.xml

compile:
    [javac] /Users/austin/ence688r.d/java-code-graphs/build.xml:8:
    warning: 'includeantruntime' was not set, defaulting to build.sysclasspath=last;
    set to false for repeatable builds

run01:
     [java] ([Node 1, Node 2, Node 3, Node 4, Node 5], [])

BUILD SUCCESSFUL
Total time: 1 second
prompt >>
prompt >> exit
Script done on Mon Apr  9 10:43:20 2012
```

Example 2. Operations on Simple Directed Graphs

Problem Statement. In this example, we assemble the directed graph shown in Figure 13.31 ...

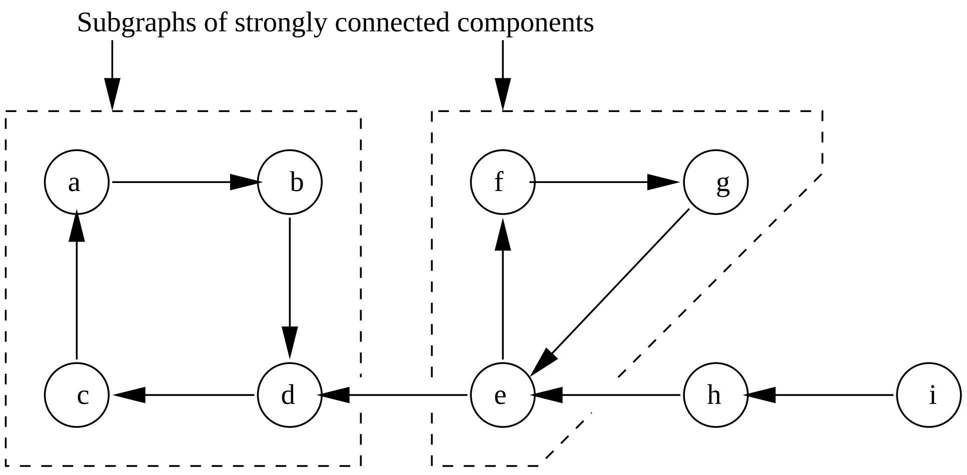


Figure 13.31. Directed graph and annotations for strongly connected subgraphs.

and then,

1. Compute the set (or subsets) of strongly connected components,
2. Compute the shortest path from node i to node c.
3. Compute the shortest path from node c to node i.

Recall that a directed graph is **strongly connected** if it ...

> **... contains a directed path from u to v and a directed path from v to u for every pair of vertices u, v.**

A valid path from node `c` to node `i` does not exist, so the path algorithm should simply return `null`.

File: demo/jgrapht/TestDirectedGraph.java.

source code

```
/**
 * ======================================================================
 * TestDirectedGraph.java. Demonstrate operations that can be performed on
 *                         directed graphs.
 *
 * ======================================================================
 */

package jgrapht;
```

```
import java.util.List;

import org.jgrapht.alg.*;
import org.jgrapht.*;
import org.jgrapht.graph.*;

public class TestDirectedGraph {
    public static void main(String args[]) {

        System.out.println("TestDirectedGraph ...              ");
        System.out.println("===================================");

        // Constructs a directed graph with the specified vertices and edges

        DirectedGraph<String, DefaultEdge> directedGraph =
            new DefaultDirectedGraph<String, DefaultEdge> (DefaultEdge.class);

        directedGraph.addVertex("a");
        directedGraph.addVertex("b");
        directedGraph.addVertex("c");
        directedGraph.addVertex("d");
        directedGraph.addVertex("e");
        directedGraph.addVertex("f");
        directedGraph.addVertex("g");
        directedGraph.addVertex("h");
        directedGraph.addVertex("i");
        directedGraph.addEdge("a", "b");
        directedGraph.addEdge("b", "d");
        directedGraph.addEdge("d", "c");
        directedGraph.addEdge("c", "a");
        directedGraph.addEdge("e", "d");
        directedGraph.addEdge("e", "f");
        directedGraph.addEdge("f", "g");
        directedGraph.addEdge("g", "e");
        directedGraph.addEdge("h", "e");
        directedGraph.addEdge("i", "h");

        // Computes all the strongly connected components of the directed graph

        StrongConnectivityInspector sci = new StrongConnectivityInspector(directedGraph);
        List stronglyConnectedSubgraphs = sci.stronglyConnectedSubgraphs();

        // Prints the strongly connected components

        System.out.println("Strongly connected components:");
        for (int i = 0; i < stronglyConnectedSubgraphs.size(); i++) {
            System.out.println(stronglyConnectedSubgraphs.get(i));
        }
        System.out.println();

        // Prints the shortest path from vertex i to vertex c. This certainly
        // exists for our particular directed graph.

        System.out.println("Shortest path from i to c:");
        List path = DijkstraShortestPath.findPathBetween(directedGraph, "i", "c");
        System.out.println(path + "\n");

        // Prints the shortest path from vertex c to vertex i. This path does
        // NOT exist for our particular directed graph. Hence the path is
        // empty and the variable "path" must be null.

        System.out.println("Shortest path from c to i:");
        path = DijkstraShortestPath.findPathBetween(directedGraph, "c", "i");
        System.out.println(path);
    }
}
```

Input and Output. The script of input/output for the program execution is as follows:

```
Script started on Mon Apr  9 10:45:38 2012
prompt >> ant run02
Buildfile: /Users/austin/ence688r.d/java-code-graphs/build.xml

compile:
    [javac] /Users/austin/ence688r.d/java-code-graphs/build.xml:8:
    warning: 'includeantruntime' was not set, defaulting to build.sysclasspath=last;
    set to false for repeatable builds

run02:
     [java] TestDirectedGraph ...
     [java] ===================================
     [java] Strongly connected components:
     [java] ([i], [])
     [java] ([h], [])
     [java] ([e, f, g], [(e,f), (f,g), (g,e)])
     [java] ([a, b, c, d], [(a,b), (b,d), (d,c), (c,a)])
     [java]
     [java] Shortest path from i to c:
     [java] [(i : h), (h : e), (e : d), (d : c)]
     [java]
     [java] Shortest path from c to i:
     [java] null

BUILD SUCCESSFUL
Total time: 1 second
prompt >>
prompt >> exit
Script done on Mon Apr  9 10:45:47 2012
```

Example 3. A Strongly Connected Graph with Cycles

Problem Statement. This program creates directed graphs as shown on the left- and right-hand sides of Figure 13.32.

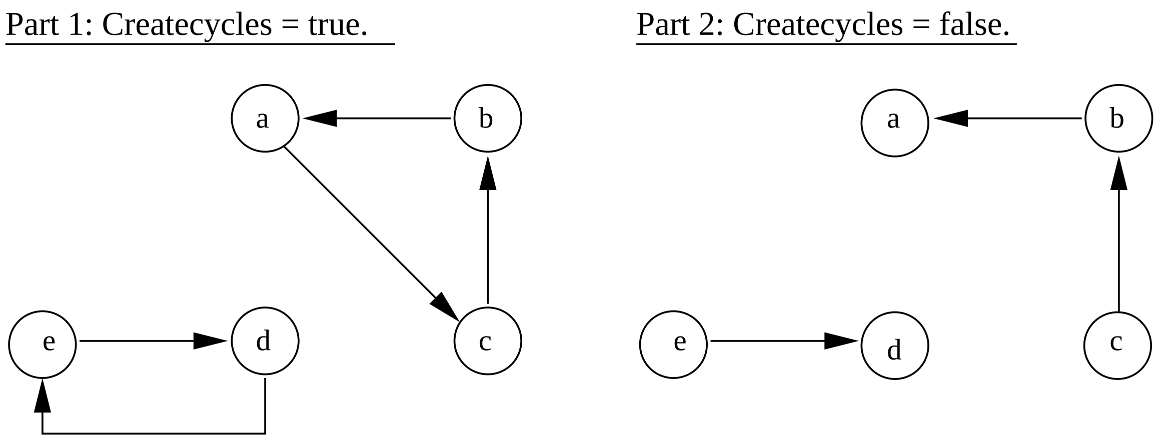


Figure 13.32. Test graph containing cycles.

For each graph, algorithms find and print the details of any implicit loops (cycles).

File: demo/jgrapht/TestDirectedGraph2.java.

source code

```
/**
  *  ====================================================================
  *  TestDirectedGraph2.java: Create a depencency chart, directed graph,
  *  then locate and print any implicit loops (cycles).
  *  ====================================================================
  */

package jgrapht;

import java.util.Iterator;
import java.util.Set;

import org.jgrapht.alg.CycleDetector;
import org.jgrapht.traverse.TopologicalOrderIterator;
import org.jgrapht.graph.DefaultDirectedGraph;
import org.jgrapht.graph.DefaultEdge;

public class TestDirectedGraph2 {

   /**
     *  @param createCycles true - create a directed graph which contains
     * cycles.  false - create a directed graph which does not contain any cycles.
     */

   public static void test(boolean createCycles) {
```

```
CycleDetector<String, DefaultEdge> cycleDetector;
DefaultDirectedGraph<String, DefaultEdge> g;

g = new DefaultDirectedGraph<String, DefaultEdge>( DefaultEdge.class );

// Add vertices, e.g. equations.

g.addVertex("a");
g.addVertex("b");
g.addVertex("c");
g.addVertex("d");
g.addVertex("e");

// Add edges, e.g. dependencies.
// 2 cycles,
//    a = f(b)
//    b = f(c)
//    c = f(a)
// and
//    d = f(e)
//    e = f(d)

g.addEdge("b", "a");
g.addEdge("c", "b");

if (createCycles) {
    g.addEdge("a", "c");
}
g.addEdge("e", "d");
if (createCycles) {
   g.addEdge("d", "e");
}

// Print details of assembled graph

System.out.println( g.toString() );

// Test: Are there cycles in the dependencies?

cycleDetector = new CycleDetector<String, DefaultEdge>(g);

// Cycle(s) detected.

if (cycleDetector.detectCycles()) {
   Iterator<String> iterator;
   Set<String> cycleVertices;
   Set<String> subCycle;
   String cycle;

   System.out.println("Cycles detected.");

   // Get all vertices involved in cycles.

   cycleVertices = cycleDetector.findCycles();
```

```
         // Loop through vertices trying to find disjoint cycles.

         while (! cycleVertices.isEmpty()) {
            System.out.println("Cycle:");

            // Get a vertex involved in a cycle.
            iterator = cycleVertices.iterator();
            cycle = iterator.next();

            // Get all vertices involved with this vertex.
            subCycle = cycleDetector.findCyclesContainingVertex(cycle);
            for (String sub : subCycle) {
               System.out.println("   " + sub);
               // Remove vertex so that this cycle is not encountered
               // again.
               cycleVertices.remove(sub);
            }
         }
      }

      // No cycles.  Just output properly ordered vertices.

      else {
         String v;
         TopologicalOrderIterator<String, DefaultEdge> orderIterator;

         orderIterator = new TopologicalOrderIterator<String, DefaultEdge>(g);
         System.out.println("\nOrdering:");
         while (orderIterator.hasNext()) {
            v = orderIterator.next();
            System.out.println(v);
         }
      }
   }

   public static void main(String [] args) {

      System.out.println("Case 1: Create graph with cycles.");
      System.out.println("================================= ");

      test(true);

      System.out.println("Case 2: Create graph with no cycles.");
      System.out.println("================================== ");

      test(false);

      System.out.println("All done");
      System.exit(0);
   }
}
```

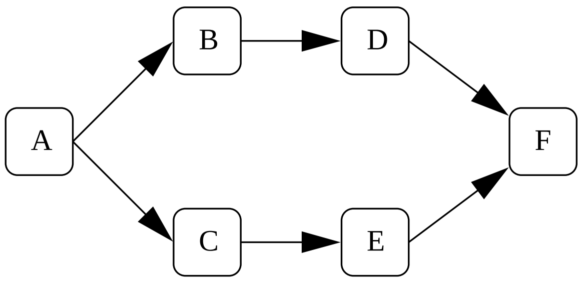
Input and Output. The script of input and output is as follows:

```
Script started on Wed Apr 11 18:28:39 2012
prompt >> ant run03
Buildfile: /Users/austin/ence688r.d/java-code-graphs/build.xml

compile:
    [javac] /Users/austin/ence688r.d/java-code-graphs/build.xml:8:
    warning: 'includeantruntime' was not set, defaulting to build.sysclasspath=last;
    set to false for repeatable builds

run03:
     [java] Case 1: Create graph with cycles.
     [java] ==================================
     [java] ([a, b, c, d, e], [(b,a), (c,b), (a,c), (e,d), (d,e)])
     [java] Cycles detected.
     [java] Cycle:
     [java]    d
     [java]    e
     [java] Cycle:
     [java]    b
     [java]    c
     [java]    a
     [java] Case 2: Create graph with no cycles.
     [java] ===================================
     [java] ([a, b, c, d, e], [(b,a), (c,b), (e,d)])
     [java]
     [java] Ordering:
     [java] c
     [java] e
     [java] b
     [java] d
     [java] a
     [java] All done

BUILD SUCCESSFUL
Total time: 1 second
prompt >> exit
Script done on Wed Apr 11 18:28:47 2012
```

13.12 Dependency Graphs

Definition

A dependency graph is a directed graph representing dependencies among objects.



The adjacent figure expresses the following list of dependency relationships: Node A depends on nodes B and C. Node B depends on D. Node C depends on E. And nodes D and E both depend on F.

From this list we can state: Node A depends on nodes B, C, D, E and F. Also, if node F fails, then it will impact nodes B, C, D, E and A.

Problems

Dependency graphs crop up in a wide variety of problems and domains:

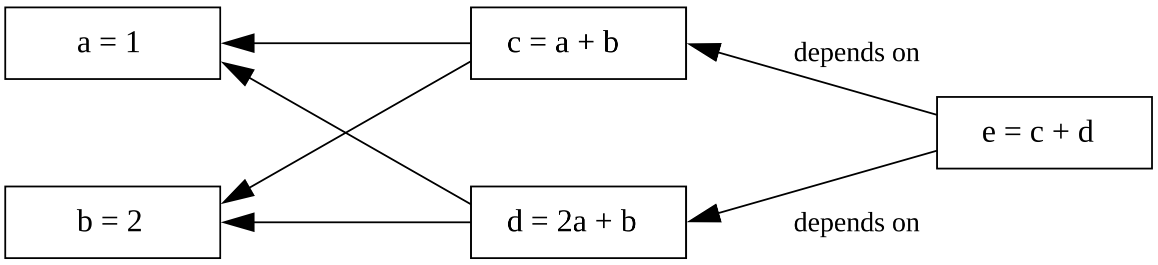
- Spreadsheet calculators.
- Job shop and construction site scheduling.
- Traceability analysis of requirements (Systems Engineering).
- Design Structure Matrices (Systems Engineering).
- To represent relationships among classes in a software program (in this sense, class diagrams can be viewed as dependency graphs).
- Build programs such as Apache Ant use dependency graphs to determine the order for compiling a software package.
- Automated software installers (to make sure dependent packages are up-to-date).
- Garbage collection algorithms in Java.

Example. Dependency Relationships in Arithmentic Expressions.

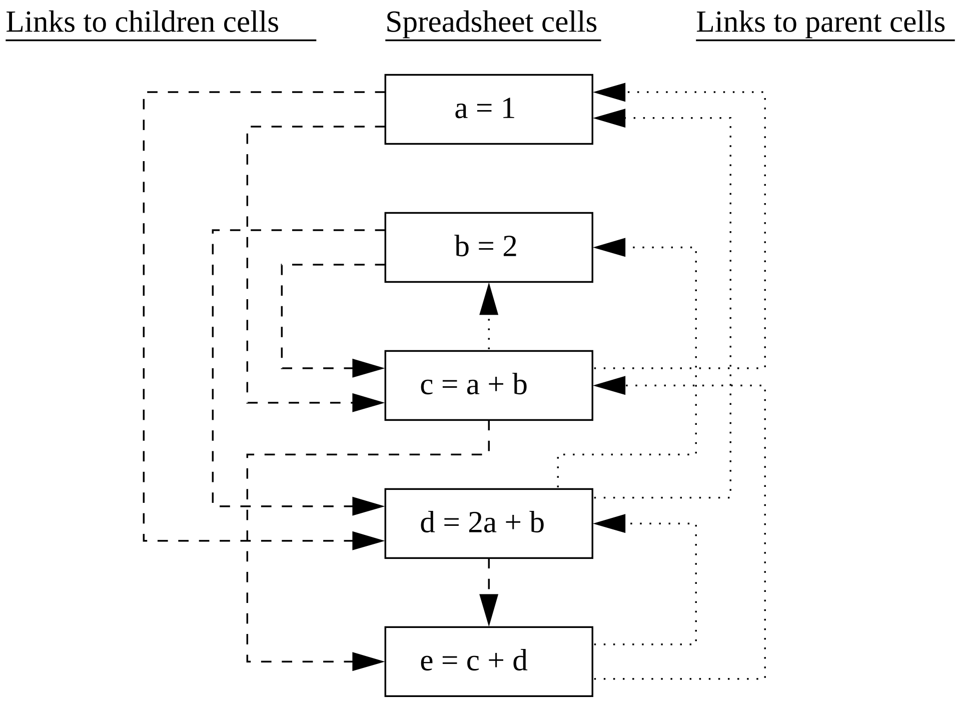
Consider the following set of equations:

$$a = 1 \tag{13.2}$$
$$b = 2 \tag{13.3}$$
$$c = a + b \tag{13.4}$$
$$d = 2 * a + b \tag{13.5}$$
$$e = c + d \tag{13.6}$$

The graph of dependencies is as follows:



The corresponding parent and child links among spreadsheet cells are:



13.13 JGraphT (Dependency Graph Operations)

Example 4. Dependency Graph for Human Organs and Body Systems

Problem Statement. The nodes within a JGraphT graph can be any Java objects – the graph structure itself describes the relationship among these objects.

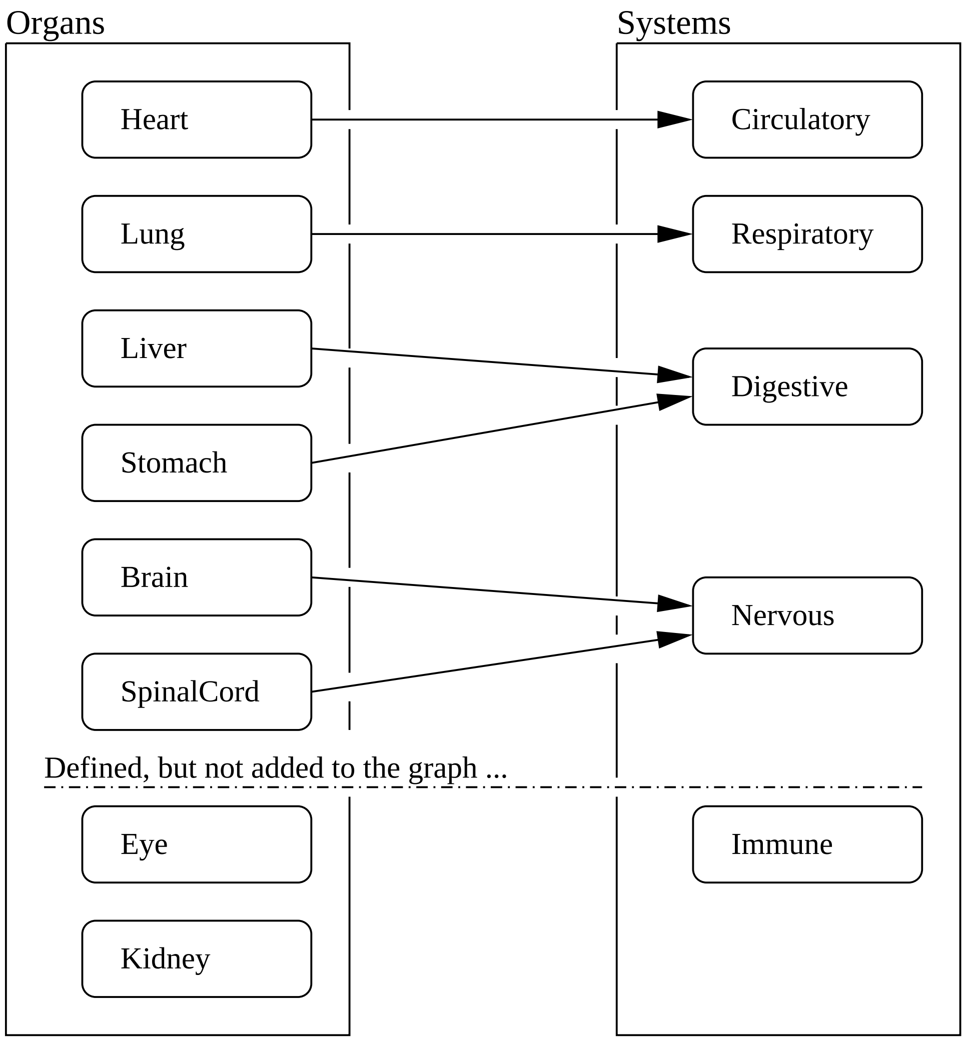


Figure 13.33. Graph of human organs and body systems.

As illustrated in Figure 13.33, in this example we will build a graph – actually, a disconneted graph – of dependency relationships between human organs and their membership in body systems. For example, the heart is part of the circulatory body system.

File: demo/jgrapht/HumanOrgansGraph.java.

source code

```
/*
 *  ======================================================================
 *  HumanOrgansGraph.java: Demo of JGraphT API.
 *
 *  Example code from Wicked Cool Java (No Starch Press)
 *  Copyright (C) 2005 Brian D. Eubanks
 *
 *  Slightly modified by Mark Austin                           November 2011
 *  ======================================================================
 */

package jgrapht;

import java.util.List;
import java.util.Set;

import org.jgrapht.*;
import org.jgrapht.alg.*;
import org.jgrapht.graph.*;
import org.jgrapht.graph.ListenableDirectedGraph;

public class HumanOrgansGraph {

   enum  Organs { HEART, LUNG, LIVER, STOMACH, BRAIN, SPINALCORD, EYE, KIDNEY };
   enum Systems { CIRCULATORY, DIGESTIVE, NERVOUS, RESPIRATORY, IMMUNE };

   ListenableDirectedGraph graph = null;

   /*
    * Create an instance using the provided graph.
    * @param g The graph to use, or null to create a new one.
    */

   public HumanOrgansGraph(ListenableDirectedGraph g) {
      if (g == null) {
         g = new ListenableDirectedGraph<Enum, DefaultEdge> (DefaultEdge.class);
      }

      graph = g;

      // Add vertices to the graph

      g.addVertex(Organs.HEART);
      g.addVertex(Organs.LUNG);
      g.addVertex(Organs.BRAIN);
      g.addVertex(Organs.STOMACH);
      g.addVertex(Organs.LIVER);
      g.addVertex(Organs.SPINALCORD);

      g.addVertex(Systems.CIRCULATORY);
      g.addVertex(Systems.NERVOUS);
      g.addVertex(Systems.DIGESTIVE);
      g.addVertex(Systems.RESPIRATORY);

      // Link the vertices by edges
      g.addEdge(       Organs.HEART, Systems.CIRCULATORY);
      g.addEdge(        Organs.LUNG, Systems.RESPIRATORY);
      g.addEdge(       Organs.BRAIN,     Systems.NERVOUS);
      g.addEdge(  Organs.SPINALCORD,     Systems.NERVOUS);
      g.addEdge(     Organs.STOMACH,   Systems.DIGESTIVE);
      g.addEdge(       Organs.LIVER,   Systems.DIGESTIVE);

      // Simple test to see if certain vertices are in the graph ...

      System.out.println("");
      System.out.println("Test for existence of organs/systems");
      System.out.println("====================================");

      System.out.println("Test: Graph contains Systems.NERVOUS : " +
                         g.containsVertex ( Systems.NERVOUS ) );
      System.out.println("Test: Graph contains Organs.EYE : " +
                         g.containsVertex ( Organs.EYE ) );

      // Traverse the edges connected to DIGESTIVE vertex ...

      System.out.println("");
      System.out.println("Count set relationships in graph");
      System.out.println("====================================");

      Set digestiveLinks = g.edgesOf( Systems.DIGESTIVE );
      System.out.printf("There are %3d digestive organs in the graph\n",
                        digestiveLinks.size() );
      Set lungLinks = g.edgesOf( Organs.LUNG );
      System.out.printf("The lung is part of %3d systems in the graph\n",
                        lungLinks.size() );

      // Walk along incoming edges and print source vertices ...

      System.out.println("");
      System.out.println("Print list of organs in the digestive system");
      System.out.println("============================================");

      for (Object item : digestiveLinks) {
           DefaultEdge edge = (DefaultEdge) item;
           Object source = graph.getEdgeSource( edge );
           System.out.println( "Source: " + source.toString() );
      }

      System.out.println("============================================");
   }

   // Exericse methods in human organ graph ...

   public static void main(String[] args) {

      System.out.println("Assemble Graph of Human Organ Systems");
      System.out.println("=====================================");

      new HumanOrgansGraph(null);
   }
}
```

Input and Output. The script of input and output is as follows:

```
Script started on Mon Apr  9 11:01:35 2012
prompt >> ant run04
Buildfile: /Users/austin/ence688r.d/java-code-graphs/build.xml

compile:
    [javac] /Users/austin/ence688r.d/java-code-graphs/build.xml:8:
            warning: 'includeantruntime' was not set,
            defaulting to build.sysclasspath=last;
            set to false for repeatable builds

run04:
     [java] Assemble Graph of Human Organ Systems
     [java] =====================================
     [java]
     [java] Test for existence of organs/systems
     [java] ====================================
     [java] Test: Graph contains Systems.NERVOUS : true
     [java] Test: Graph contains Organs.EYE : false
     [java]
     [java] Count set relationships in graph
     [java] ====================================
     [java] There are   2 digestive organs in the graph
     [java] The lung is part of   1 systems in the graph
     [java]
     [java] Print list of organs in the digestive system
     [java] ============================================
     [java] Source: STOMACH
     [java] Source: LIVER
     [java] ============================================

BUILD SUCCESSFUL
Total time: 1 second
prompt >>
prompt >> exit
Script done on Mon Apr  9 11:01:45 2012
```

A few points to note:

1. The graph is implemented as a listenable directed graph, i.e.,

```
Graph g = new ListenableDirectedGraph<Enum, DefaultEdge> (DefaultEdge.class);
```

Here, Graph is the interface implemented by ListenableDirectedGraph. The syntax,

```
<Enum, DefaultEdge> (DefaultEdge.class);
```

tells the compiler that the graph nodes and edges will be of type Eunum (for enumerated data types) and DefaultEdge, respectively.

2. For convenience, the sets of human organ and body system entities are defined as enumerated data types, i.e.,

```
enum  Organs { HEART, LUNG, ...  EYE, KIDNEY };
enum Systems { CIRCULATORY, ... RESPIRATORY, IMMUNE };
```

Entities are added to the graph by calling the addVertex() method, e.g.,

```
g.addVertex( Organs.HEART );
```

And relationships between organs and body systems is defined by adding an edge to the graph, e.g.,

```
g.addEdge( Organs.HEART, Systems.CIRCULATORY );
```

Notice that eye and kidney are defined as body organs, and immune is defined as a body system, but none of these entities are actually added to the graph.

3. When assembly of the graph is complete, tests are conducted to see if certain vertices are in the graph. The test program also traverse the graph to count the number of organs belonging to the digestive systems, and to see which system the lungs belong.

Example 5. Demonstrate Topological Sort

A topological sort (or topological ordering) of a directed graph is ...

> **... a linear ordering of its vertices such that, for every edge u-v, u comes before v in the ordering.**

For example, the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another. In this case, ...

> **... a topological ordering is just a valid sequence for the tasks.**

Topological ordering is possible if and only if the graph has no directed cycles/loops, and in such cases, multiple valid orderings may be possible.

Problem Statement. In this example, we exercise we compute the assembly sequence for the structural configuration shown in Figure 13.34.

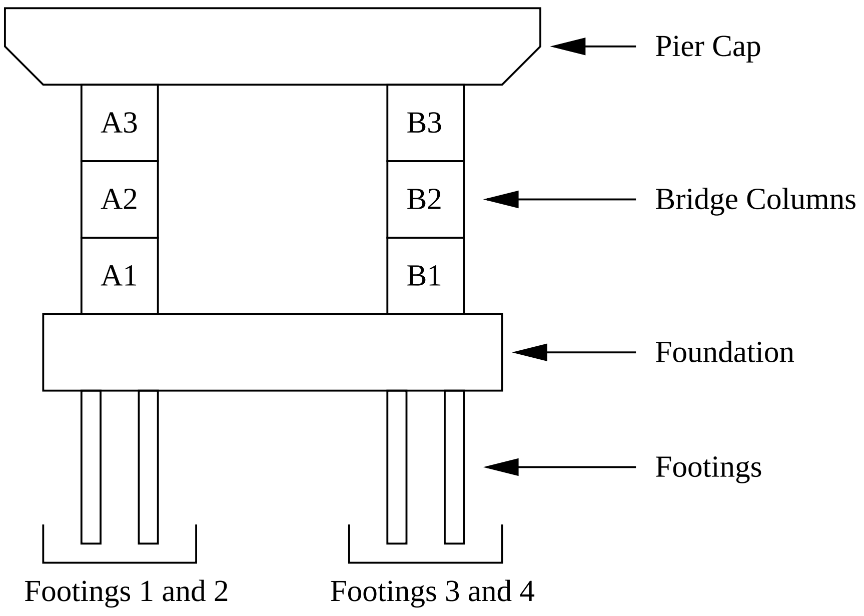


Figure 13.34. Front elevation view of a bridge pier, foundation and footings.

The modeling procedure is as follows:

1. Create a graph with one vertex for each of the structural components (e.g., Footing1, ColA!, and so forth).
2. Specify dependency relationships between the components.
3. Compute a valid assembly sequence through a topological ordering of the graph vertices.

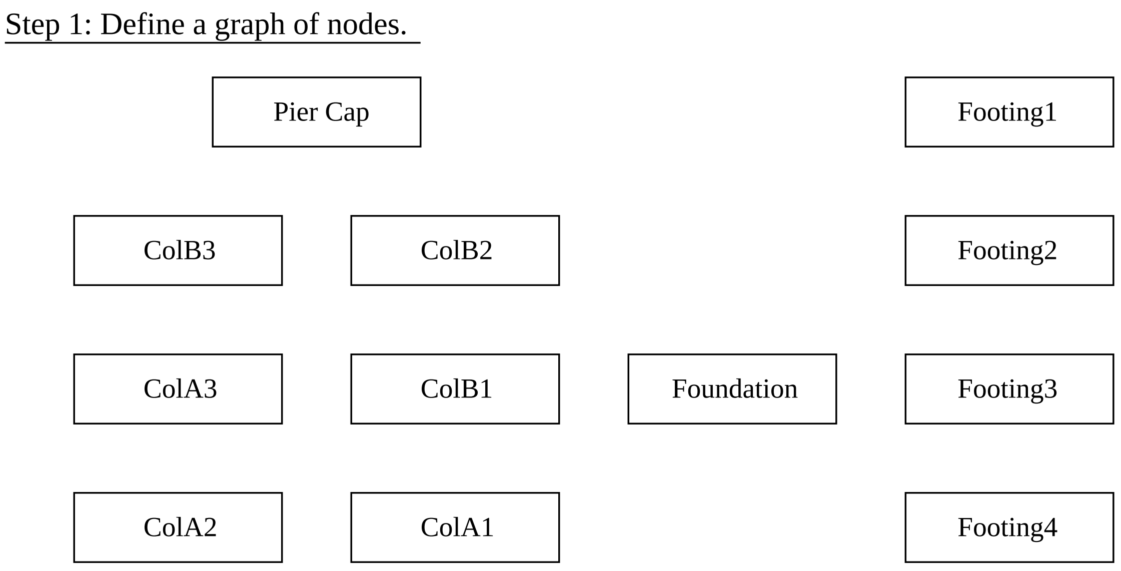


Figure 13.35. Graph of bridge components (and no graph edges).

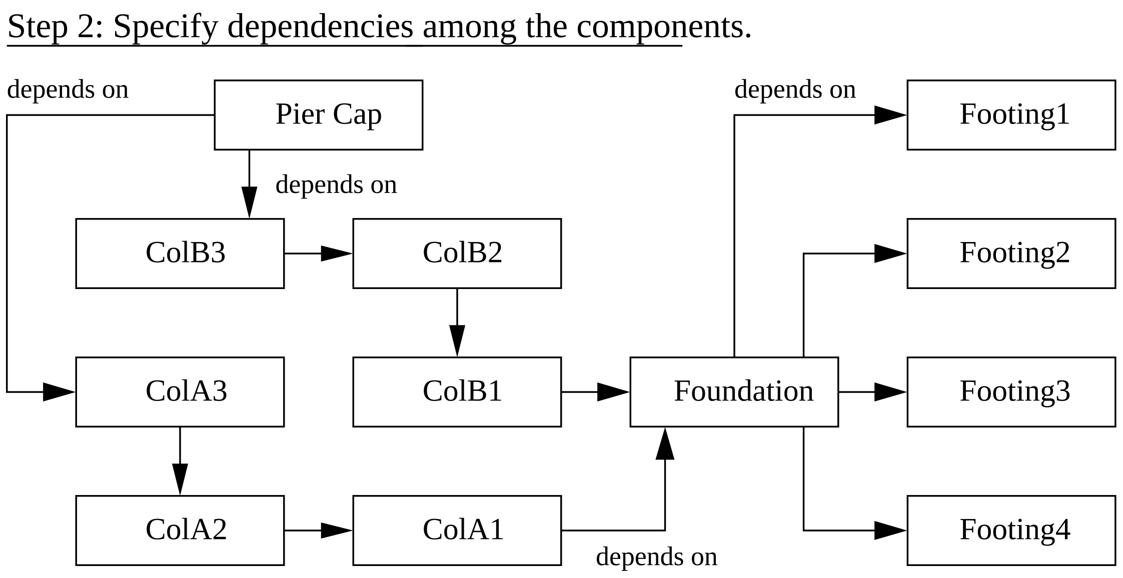


Figure 13.36. Graph of bridge components and dependency relationships for assembly.

Figures 13.35 and 13.36 show the graph configuration at steps 1 and 2 of the procedure.

File: demo/jgrapht/TopologicalBlocks.java.

source code

```
/* ==================================================================
 * TopologicalBlocks. Simulate assembly sequence for a stack of blocks.
 * ==================================================================
 */

package jgrapht;

import java.util.Iterator;

import org.jgrapht.*;
import org.jgrapht.DirectedGraph;
import org.jgrapht.graph.*;
import org.jgrapht.traverse.TopologicalOrderIterator;
import org.jgrapht.graph.DefaultDirectedGraph;
import org.jgrapht.graph.DefaultEdge;
import org.jgrapht.traverse.*;

public class TopologicalBlocks {

    public static void main ( String args[] ) {
        TopologicalBlocks run = new TopologicalBlocks();
        run.testBridge();
    }

    public void testBridge() {

        DirectedGraph<String, DefaultEdge> graph =
           new DefaultDirectedGraph<String, DefaultEdge>( DefaultEdge.class );

        String v[] = new String[12];

        v[0] = "ColA1";
        v[2] = "ColA2";
        v[4] = "ColA3";

        v[1] = "COlB1";
        v[3] = "ColB2";
        v[5] = "ColB3";

        v[7] = "Foundation";
        v[6] = "Pier Cap";

        v[8]  = "Footing1";
        v[9]  = "Footing2";
        v[10] = "Footing3";
        v[11] = "Footing4";

        // Throw blocks into library in random order ...

        graph.addVertex(v[11]);
        graph.addVertex(v[10]);
        graph.addVertex(v[9]);
        graph.addVertex(v[8]);

        graph.addVertex(v[7]);
        graph.addVertex(v[6]);
        graph.addVertex(v[0]);
        graph.addVertex(v[1]);
        graph.addVertex(v[2]);
        graph.addVertex(v[3]);
        graph.addVertex(v[4]);
        graph.addVertex(v[5]);

        // =====================================
        // Specify dependencies among blocks ...
        // =====================================

        // Need lay foundation before building the columns ....

        graph.addEdge(v[7], v[0]);
        graph.addEdge(v[7], v[1]);

        // Specify assembly order for Pier A...

        graph.addEdge(v[0], v[2]);
        graph.addEdge(v[2], v[4]);

        // Specify assembly order for Pier B...

        graph.addEdge(v[1], v[3]);
        graph.addEdge(v[3], v[5]);

        // Add pier cap to columns ...

        graph.addEdge(v[4], v[6]);
        graph.addEdge(v[5], v[6]);

        // Put footings under foundation ...

        graph.addEdge( v[8], v[7]);
        graph.addEdge( v[9], v[7]);
        graph.addEdge(v[10], v[7]);
        graph.addEdge(v[11], v[7]);

        graph.addEdge( v[8],  v[9]);
        graph.addEdge( v[9], v[10]);

        Iterator<String> iter = new TopologicalOrderIterator<String, DefaultEdge>(graph);

        System.out.println("Sequence of Assembly");
        System.out.println("====================");

        int i = 0;
        while (iter.hasNext() != false) {
            String s = (String) iter.next();
```

```
            System.out.println( s.toString() );
        }
    }
}
```

Input and Output.

```
Script started on Mon Apr  9 11:07:05 2012
prompt >> ant run07
Buildfile: /Users/austin/ence688r.d/java-code-graphs/build.xml

compile:
    [javac] /Users/austin/ence688r.d/java-code-graphs/build.xml:8:
            warning: 'includeantruntime' was not set,
            defaulting to build.sysclasspath=last; set to false for repeatable builds

run07:
     [java] Sequence of Assembly
     [java] ====================
     [java] Footing4
     [java] Footing1
     [java] Footing2
     [java] Footing3
     [java] Foundation
     [java] ColA1
     [java] COlB1
     [java] ColA2
     [java] ColB2
     [java] ColA3
     [java] ColB3
     [java] Pier Cap

BUILD SUCCESSFUL
Total time: 1 second
prompt >>
prompt >> exit
Script done on Mon Apr  9 11:07:15 2012
```

Example 6. Graphs of Relationship Edges

Problem Statement.

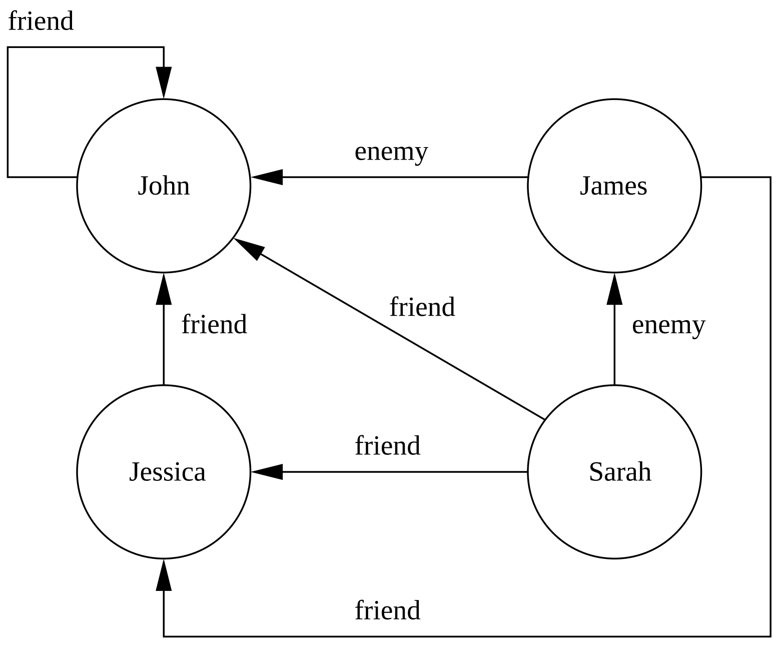


Figure 13.37. Graph of labeled relationship edges.

File: demo/jgrapht/TestLabeledEdges.java.

source code

```
package jgrapht;

import java.util.ArrayList;
import org.jgrapht.DirectedGraph;
import org.jgrapht.graph.DirectedMultigraph;
import org.jgrapht.graph.ClassBasedEdgeFactory;
import jgrapht.RelationshipEdge;

public class TestLabeledEdges {

    private static final String friend = "friend";
    private static final String  enemy = "enemy";

    public static void main(String[] args) {

        System.out.println("In TestLabeledEdges.main()....");
        System.out.println("-----------------------------");

        DirectedGraph<String, RelationshipEdge> graph =
            new DirectedMultigraph<String, RelationshipEdge>(
                    new ClassBasedEdgeFactory<String, RelationshipEdge>(RelationshipEdge.clas
        ArrayList<String> people = new ArrayList<String>();
        people.add("John");
        people.add("James");
        people.add("Sarah");
        people.add("Jessica");

        // John is everyone’s friend

        for (String person : people) {
            graph.addVertex(person);
            graph.addEdge(people.get(0), person,
                          new RelationshipEdge<String>(people.get(0), person, friend));
        }

        // Apparently James doesn’t really like John

        graph.addEdge("James", "John", new RelationshipEdge<String>("James", "John", enemy));

        // Jessica is Sarah and James’s friend

        graph.addEdge("Jessica", "Sarah",
                      new RelationshipEdge<String>("Jessica", "Sarah", friend));
        graph.addEdge("Jessica", "James",
                      new RelationshipEdge<String>("Jessica", "James", friend));

        // But Sarah doesn’t really like James

        graph.addEdge("Sarah", "James",
                      new RelationshipEdge<String>("Sarah", "James", enemy));

        // Print relationship graph ....

        System.out.println ("Part 1: Relationship Graph");
        System.out.println ("==========================");

        System.out.println (graph.toString() );

        // Print list of edge relationships

        System.out.println ("");
        System.out.println ("Part 2: Enumeration of Edge Relationships");
        System.out.println ("=========================================");

        System.out.println ( graph.edgeSet().toString() );

        for (RelationshipEdge edge : graph.edgeSet()) {
            if ( edge.equals(enemy) ) {
                System.out.printf("%s is an enemy of %s\n", edge.getV1(), edge.getV2());
            } else if ( edge.equals(friend) ) {
                System.out.printf("%s is a friend of %s\n", edge.getV1(), edge.getV2());
            }
        }
    }
}
```

File: demo/jgrapht/RelationshipEdge.java.

Here is the source code for a customized edge that stores a character string relationship (i.e., “friend” or “enemy”).

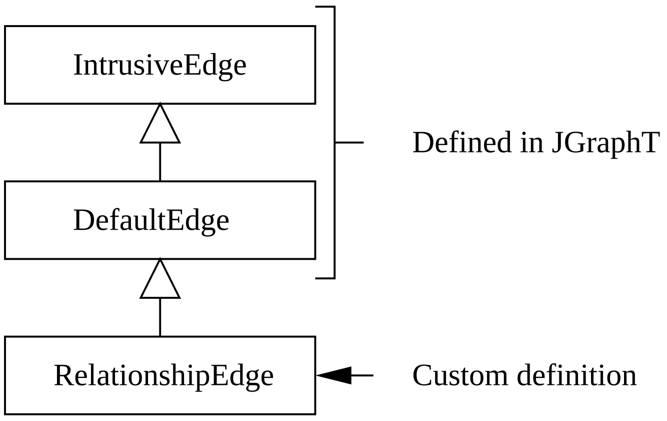


Figure 13.38. Class hierarchy for the definition of relationship edges.

As illustrated in Figure 13.38, the class relationship edge is defined as an extension of DefaultEdge, which is provided as part of JGraphT.

source code

```
package jgrapht;

import java.util.ArrayList;
import org.jgrapht.*;
import org.jgrapht.graph.*;

public class RelationshipEdge<V> extends DefaultEdge {
   private V v1;
   private V v2;
   private String label;

   public RelationshipEdge(V v1, V v2, String label) {
      this.v1 = v1;
      this.v2 = v2;
      this.label = label;
   }

   public V getV1() {
      return v1;
   }

   public V getV2() {
      return v2;
   }

   public boolean equals( String label ) {
      if ( label.equals( this.label ) == true )
           return true;
      else
           return false;
   }

   public String toString() {
      return label;
   }
}
```

Program Input and Output.

```
Script started on Thu Apr 12 12:02:04 2012
prompt >> ant run05

compile:
    [javac] /Users/austin/ence688r.d/java-code-graphs/build.xml:8:
    warning: ’includeantruntime’ was not set,
    defaulting to build.sysclasspath=last; set to false for repeatable builds

run05:
     [java] In TestLabeledEdges.main()....
     [java] ------------------------------
     [java]
     [java] Part 1: Relationship Graph
     [java] ==========================
     [java] ( [John, James, Sarah, Jessica] ,
             [ friend = (John,John),
               friend = (John,James),
               friend = (John,Sarah),
               friend = (John,Jessica),
               enemy  = (James,John),
               friend = (Jessica,Sarah),
               friend = (Jessica,James),
               enemy  = (Sarah,James)] )
     [java]
     [java] Part 2: Enumeration of Edge Relationships
     [java] =========================================
     [java] [friend, friend, friend, friend, enemy, friend, friend, enemy]
     [java] John is a friend of John
     [java] John is a friend of James
     [java] John is a friend of Sarah
     [java] John is a friend of Jessica
     [java] James is an enemy of John
     [java] Jessica is a friend of Sarah
     [java] Jessica is a friend of James
     [java] Sarah is an enemy of James

BUILD SUCCESSFUL
Total time: 1 second
prompt >> exit
Script done on Thu Apr 12 12:02:18 2012
```

13.14 Simplified Model of Washington DC Metro System

Problem Statement. In this example we use JGraphT to build a model of the Washington DC Metro System.

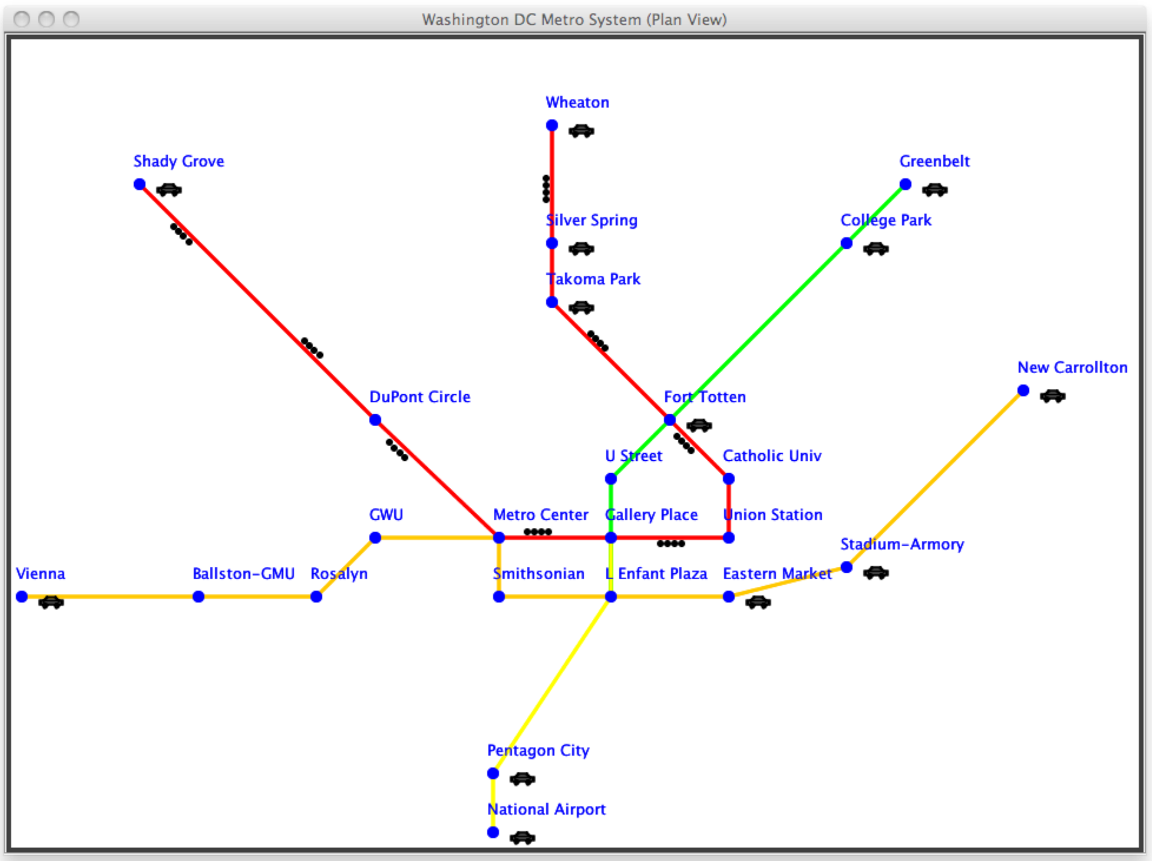


Figure 13.39. Plan view for a fragment of the Washington DC Metro System.

Figure 13.39 shows the fragment of the metro system that will be considered. The graph model will be implemented in two steps:

1. Create and test a symbol table for the storage of metro station objects.
2. Use JGraphT to systematically assemble a graph model for a fragment of the DC Metro System.

Finally, the algorithms within JGraphT will be called for route planning for "College Park" station to "National Airport" and "College Park" station to "New Carollton" station.

Part 1: Create Symbol Table for MetroStations

Source Code: MetroStation.java.

source code

```
/*
 *  ===================================================================
 *  MetroStation.java: Representation for a Metrostation.
 *  ===================================================================
 */

package metro;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import metro.MetroBubble;
import metro.Coordinate;

public class MetroStation {
    protected boolean parking  = false;
    protected boolean security = false;
    protected boolean busroute = false;
    protected boolean waypoint = true;
    protected List    onTrack;
    String  stationName;
    String      mapFile;
    MetroBubble hint;
    Coordinate coord;

    // Contructor methods ...

    public MetroStation() {}

    public MetroStation( String name, double dX, double dY ) {
        stationName = name;
        onTrack = new ArrayList();  // Initialize array list of line colors
        coord   = new Coordinate (dX, dY);
        hint    = new MetroBubble();
    }

    public MetroStation( String name, double dX, double dY,
                         boolean park, boolean sec, boolean br ) {
        stationName = name;
        onTrack = new ArrayList();  // Initialize array list of line colors
        coord   = new Coordinate (dX, dY);
        hint    = new MetroBubble();

        parking  = park;
        security = sec;
        busroute = br;
    }

    public void setMapFile( String file ){
        mapFile = file;
    }

    // Deal with bubble message ....

    public void setBubbleMessage ( String sMessage ) {
        hint.setMessage( sMessage );
    }

    // Deal with station name ....

    public void setStationName ( String sName ) {
        stationName = sName;
    }

    public String getStationName () {
        return stationName;
    }

    // Deal with station coordinates ....

    public void setCoord ( Coordinate c ) {
        coord = c;
    }

    public Coordinate findCoord () {
        return coord;
    }

    // Assign metro station to a track ....

    public void add( String track ) {
       onTrack.add ( track );
    }

    // Deal with bus route ...

    public void setWayPoint( boolean waypoint ) {
       this.waypoint = waypoint;
    }

    public boolean getWayPoint() {
        return waypoint;
    }

    // Deal with bus route ...

    public void setBusRoute( boolean busroute ) {
       this.busroute = busroute;
    }

    public boolean getBusRoute() {
        return busroute;
    }
    // Deal with security ...

    public void setSecurity( boolean security ) {
       this.security = security;
    }

    public boolean getSecurity() {
        return security;
    }

    // Deal with parking ...

    public void setParking( boolean park ) {
       parking = park;
    }

    public boolean getParking() {
        return parking;
    }

    // Convert description of metro station to a string ...

    public String toString() {
       String s = "MetroStation(\"" + stationName + "\")\n" +
                  "   Coordinates = (" + coord.getX() + "," + coord.getY() + ")\n" +
                  "   Parking = " + parking + "\n";

       // Walk along array list and add line names to string "s" ....

       if ( onTrack.size() > 0 ) {
          s = s.concat( "   Track   = { " );
          for (int i = 0; i < onTrack.size(); i = i + 1)
             s = s.concat( onTrack.get(i) + " ");
             s = s.concat("}\n");
         }

       return s;
   }
}
```

Source Code: SymbolTable.java.

source code

```
/*
 *  ======================================================================
 *  Compilation:  javac SymbolTable.java
 *  Execution:    java SymbolTable
 *
 *  Symbol table implementation using Java’s HashMap library.
 *  If you add a key-value pair and the key is already present,
 *  the new key-value pair replaces the old one.
 *  ======================================================================
 */

package testmetro;

import java.util.HashMap;
import java.util.Set;
import java.util.Iterator;
import java.util.Map;

import metro.*;

public class SymbolTable {

    private HashMap st = new HashMap();

    public void put(String key, Object value) { st.put(key, value);   }
    public Object get(String key)             { return st.get(key);   }
    public String toString()                  { return st.toString(); }

    // Return an array contains all of the keys

    public String[] keys() {
        Set keyvalues = st.entrySet();
        String[] keys = new String[st.size()];
        Iterator it = keyvalues.iterator();
        for (int i = 0; i < st.size(); i++) {
            Map.Entry entry = (Map.Entry) it.next();
            keys[i] = (String) entry.getKey();
        }
        return keys;
    }

    // Exercise symbol table ...

    public static void main(String[] args) {

        SymbolTable st = new SymbolTable();

        // Create metro stations ....

        MetroStation gA = new MetroStation(           "Greenbelt",  4.0,   9.5 );
        gA.setParking( true );
        MetroStation gB = new MetroStation(        "College Park",  3.5,   8.0 );
        gB.setParking( true );
        MetroStation gC = new MetroStation(       "Silver Spring",  0.0,   9.0 );
        gC.setParking( true );
        MetroStation gD = new MetroStation(         "Fort Totten",  0.0,   2.0 );
        gD.setParking( false );
        MetroStation gE = new MetroStation(       "Union Station",  0.0,   0.0 );
        gE.setParking( false );
        MetroStation gF = new MetroStation(       "DuPont Circle", -3.0,   0.0 );
        gF.setParking( false );
        MetroStation gG = new MetroStation( "Catholic University",  0.0,   1.0 );
        gG.setParking( false );
        // Insert (key, value pairs) for metro stations ....

        st.put(           "Greenbelt",  gA );
        st.put(        "College Park",  gB );
        st.put(       "Silver Spring",  gC );
        st.put(         "Fort Totten",  gD );
        st.put(       "Union Station",  gE );
        st.put(       "DuPont Circle",  gF );
        st.put( "Catholic University",  gG );

        // Define stations along the green and red lines ....

        String redLine[] = {    "Silver Spring",    "Fort Totten",
                          "Catholic University",  "Union Station",
                                "DuPont Circle" };
        String greenLine[] = {      "Greenbelt",   "College Park",
                                  "Fort Totten" };

        // Add track assignments to Metro Station Descriptions ....

        for ( int i = 0; i < redLine.length; i = i + 1 ) {
              MetroStation m = (MetroStation) st.get( redLine[i] );
              m.add("Red");
        }

        for ( int j = 0; j < greenLine.length; j = j + 1 ) {
              MetroStation m = (MetroStation) st.get( greenLine[j] );
              m.add("Green");
        }

        // Ride along track and retrieve station information.

        System.out.println( st.get(   "Fort Totten").toString() );

        // Use toString() method to print contents of symbol table ....

        System.out.println( st.toString() );
    }
}
```

An abbreviated script of program input/output is as follows:

```
Script started on Fri Apr 13 10:30:41 2012
prompt >> ant run10
Buildfile: /Users/austin/ence688r.d/java-code-graphs/build.xml

compile:
    [javac] /Users/austin/ence688r.d/java-code-graphs/build.xml:8:
            warning: 'includeantruntime' was not set, defaulting to build.sysclasspath=last;

run10:
     [java] MetroStation("Fort Totten")
     [java]    Coordinates = (0.0,2.0)
     [java]    Parking = false
     [java]    Track   = { Red Green }
     [java]
     [java] { Catholic University=MetroStation("Catholic University")
     [java]    Coordinates = (0.0,1.0)
     [java]    Parking = false
     [java]    Track   = { Red }
     [java] , DuPont Circle=MetroStation("DuPont Circle")
     [java]    Coordinates = (-3.0,0.0)
     [java]    Parking = false
     [java]    Track   = { Red }

     ... lines of output removed ...

     [java] , College Park=MetroStation("College Park")
     [java]    Coordinates = (3.5,8.0)
     [java]    Parking = true
     [java]    Track   = { Green }
     [java] }

BUILD SUCCESSFUL
Total time: 3 seconds
prompt >> exit
Script done on Fri Apr 13 10:30:51 2012
```

Figure 13.40 shows the layout of memory for the symbol table, hashmap, and metrostation objects stored inside the symbol table.

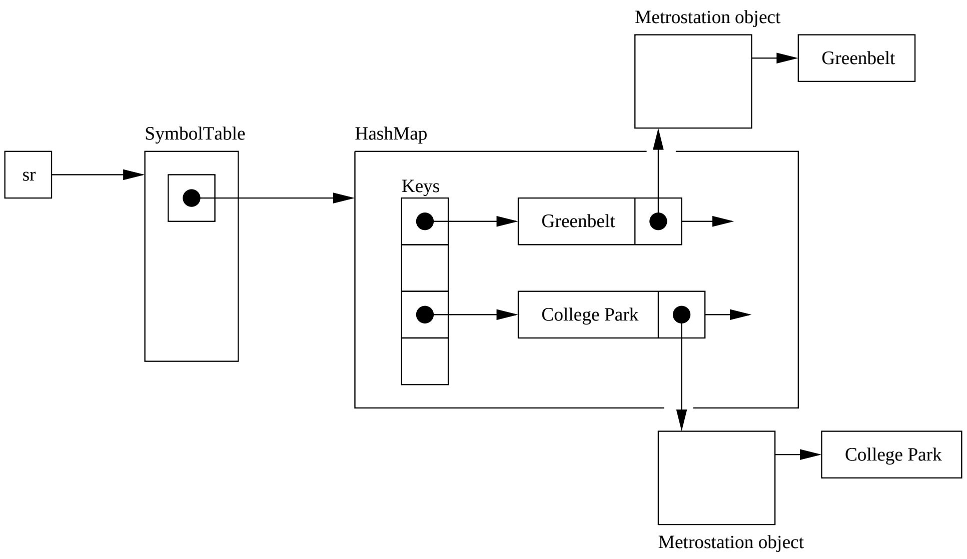


Figure 13.40. Layout of memory for the symbol table, hashmap, and metrostation objects stored inside the symbol table.

The hashmap uses a charager string (e.g., “Greenbelt”) to compute a key. It then stores a reference to a metrostation object – in other words, the metrostation objects are not part of the hashmap per se.

Part 2: Assemble a MetroSystem Graph

Source Code: MetroSystemGraph.java.

source code

```
/**
 * =====================================================================
 * MetroSystemGraph.java. Create a directed graph for the Washington DC
 *                        Metro System.
 *
 * Written by: Mark Austin                                October, 2011
 * =====================================================================
 */

package testmetro;

import java.util.List;

import org.jgrapht.alg.*;
import org.jgrapht.*;
import org.jgrapht.graph.*;

import metro.*;
import testmetro.SymbolTable;

public class MetroSystemGraph {
    String         sName;
    SymbolTable stations;
    DirectedGraph<MetroStation, DefaultEdge> metro;

    // Constructor methods ....

    public MetroSystemGraph() {
       stations = new SymbolTable();
       metro    = new DefaultDirectedGraph<MetroStation, DefaultEdge> (DefaultEdge.class);
    }

    public MetroSystemGraph( String sName ) {
       this.sName = sName;
       stations = new SymbolTable();
       metro    = new DefaultDirectedGraph<MetroStation, DefaultEdge> (DefaultEdge.class);
    }

    // Create Metro Station objects .....

    public void metroStations() {

        System.out.println("Creating metro stations..." );

        // Create metro stations. (x,y) coordinates measured in miles
        //                        from Union Station.

        MetroStation gA = new MetroStation(           "Greenbelt",   5.0,  12.0 );
        gA.setParking( true );
        MetroStation gB = new MetroStation(        "College Park",   5.0,  10.0 );
```

```
gB.setParking( true );
MetroStation gC = new MetroStation(       "Silver Spring",  -0.5,  10.0 );
gC.setParking( true );
MetroStation gD = new MetroStation(         "Fort Totten",   0.0,   2.0 );
gD.setParking( true );
MetroStation gE = new MetroStation(       "Union Station",   0.0,   0.0 );
gE.setParking( false );
MetroStation gF = new MetroStation(       "DuPont Circle",  -3.0,   1.0 );
gF.setParking( false );
MetroStation gG = new MetroStation( "Catholic University",   0.0,   1.0 );
gG.setParking( false );
MetroStation gH = new MetroStation(        "Judiciary Sq",  -1.0,   0.5 );
gH.setParking( false );
MetroStation gI = new MetroStation(    "National Airport",  -1.0,  -2.0 );
gI.setParking( true );

// Add metro stations along the "orange" line ....

MetroStation oA = new MetroStation( "New Carrollton",   5.0,   8.0 );
oA.setParking( true );
MetroStation oB = new MetroStation( "Stadium-Armory",   1.0,  -1.0 );
oB.setParking( true );
MetroStation oC = new MetroStation( "L Enfant Plaza",  -1.0,  -1.0 );
oC.setParking( false );
MetroStation oD = new MetroStation(    "Smithsonian",  -1.5,  -0.8 );
oD.setParking( false );
MetroStation oE = new MetroStation(   "Metro Center",  -1.5,   0.0 );
oE.setParking( false );
MetroStation oF = new MetroStation(        "Rosalyn",  -4.5,   0.0 );
oF.setParking( false );
MetroStation oG = new MetroStation(   "Ballston-GMU",  -6.5,   0.0 );
oG.setParking( false );
MetroStation oH = new MetroStation(         "Vienna",  -12.0,   0.0 );
oH.setParking( true );

// Add metro stations to the metro system database/symbol table.

stations.put(           "Greenbelt",  gA );
stations.put(        "College Park",  gB );
stations.put(       "Silver Spring",  gC );
stations.put(         "Fort Totten",  gD );
stations.put(       "Union Station",  gE );
stations.put(       "DuPont Circle",  gF );
stations.put( "Catholic University",  gG );
stations.put(        "Judiciary Sq",  gH );
stations.put(    "National Airport",  gI );

// Add stations along the orange line ....

stations.put(      "New Carrollton",  oA );
stations.put(      "Stadium-Armory",  oB );
stations.put(      "L Enfant Plaza",  oC );
stations.put(         "Smithsonian",  oD );
stations.put(        "Metro Center",  oE );
stations.put(             "Rosalyn",  oF );
```

```
        stations.put(        "Ballston-GMU",  oG );
        stations.put(              "Vienna",  oH );

        // Define stations along the green and red lines ....

        String redLine[] = {    "Silver Spring",    "Fort Totten",
                          "Catholic University",  "Union Station",
                                 "Metro Center",  "DuPont Circle" };
        String greenLine[] = {      "Greenbelt",   "College Park",
                                  "Fort Totten",   "Judiciary Sq",
                               "L Enfant Plaza" };
        String yellowLine[] = { "National Airport",
                                  "L Enfant Plaza",
                                    "Judiciary Sq" };
        String orangeLine[] = {   "New Carrollton",
                                  "Stadium-Armory",
                                  "L Enfant Plaza",
                                     "Smithsonian",
                                    "Metro Center",
                                         "Rosalyn",
                                    "Ballston-GMU",
                                          "Vienna" };

        // Add track assignments to metro station descriptions ....

        for ( int i = 0; i < redLine.length; i = i + 1 ) {
              MetroStation m = (MetroStation) stations.get( redLine[i] );
              m.add("Red");
        }

        for ( int j = 0; j < greenLine.length; j = j + 1 ) {
              MetroStation m = (MetroStation) stations.get( greenLine[j] );
              m.add("Green");
        }

        for ( int j = 0; j < yellowLine.length; j = j + 1 ) {
              MetroStation m = (MetroStation) stations.get( yellowLine[j] );
              m.add("Yellow");
        }

        for ( int j = 0; j < orangeLine.length; j = j + 1 ) {
              MetroStation m = (MetroStation) stations.get( orangeLine[j] );
              m.add("Orange");
        }

        // Transfer symboltable references to directed graph ....

        String names[] = stations.keys();
        for ( int j = 0; j < names.length; j = j + 1 ) {
            MetroStation m = (MetroStation) stations.get( names[j] );
            metro.addVertex( m );
        }
    }

    // Connectivity details for the DC Metrosystem Network
    public void metroNetwork() {
        MetroStation start, end;

        System.out.println("Creating metro network..." );

        // Create track/links along green line .....

        start = (MetroStation) stations.get(    "Greenbelt" );
        end   = (MetroStation) stations.get( "College Park" );
        metro.addEdge( start, end );
        metro.addEdge( end, start );

        start = (MetroStation) stations.get( "College Park" );
        end   = (MetroStation) stations.get(  "Fort Totten" );
        metro.addEdge( start, end );
        metro.addEdge( end, start );

        start = (MetroStation) stations.get(  "Fort Totten" );
        end   = (MetroStation) stations.get( "Judiciary Sq" );
        metro.addEdge( start, end );
        metro.addEdge( end, start );

        start = (MetroStation) stations.get(   "Judiciary Sq" );
        end   = (MetroStation) stations.get( "L Enfant Plaza" );
        metro.addEdge( start, end );
        metro.addEdge( end, start );

        // Create links along red line .....

        start = (MetroStation) stations.get( "Silver Spring");
        end   = (MetroStation) stations.get(  "Fort Totten" );
        metro.addEdge( start, end );
        metro.addEdge( end, start );

        start = (MetroStation) stations.get(  "Fort Totten" );
        end   = (MetroStation) stations.get(  "Catholic University");
        metro.addEdge( start, end );
        metro.addEdge( end, start );

        start = (MetroStation) stations.get(  "Catholic University");
        end   = (MetroStation) stations.get(  "Union Station");
        metro.addEdge( start, end );
        metro.addEdge( end, start );

        start = (MetroStation) stations.get(  "Union Station");
        end   = (MetroStation) stations.get(  "Metro Center");
        metro.addEdge( start, end );
        metro.addEdge( end, start );

        start = (MetroStation) stations.get(    "Metro Center");
        end   = (MetroStation) stations.get(  "DuPont Circle" );
        metro.addEdge( start, end );
        metro.addEdge( end, start );
        // Create links along yellow line .....

        start = (MetroStation) stations.get(   "Judiciary Sq" );
        end   = (MetroStation) stations.get( "L Enfant Plaza" );
        metro.addEdge( start, end );
        metro.addEdge( end, start );

        start = (MetroStation) stations.get( "L Enfant Plaza" );
        end   = (MetroStation) stations.get( "National Airport");
        metro.addEdge( start, end );
        metro.addEdge( end, start );

        // Create links along orange line .....

        start = (MetroStation) stations.get( "New Carrollton" );
        end   = (MetroStation) stations.get( "Stadium-Armory" );
        metro.addEdge( start, end );
        metro.addEdge( end, start );

        start = (MetroStation) stations.get( "Stadium-Armory" );
        end   = (MetroStation) stations.get( "L Enfant Plaza" );
        metro.addEdge( start, end );
        metro.addEdge( end, start );

        start = (MetroStation) stations.get( "L Enfant Plaza" );
        end   = (MetroStation) stations.get( "Smithsonian" );
        metro.addEdge( start, end );
        metro.addEdge( end, start );

        start = (MetroStation) stations.get( "Smithsonian" );
        end   = (MetroStation) stations.get( "Metro Center" );
        metro.addEdge( start, end );
        metro.addEdge( end, start );

        start = (MetroStation) stations.get( "Metro Center" );
        end   = (MetroStation) stations.get( "Rosalyn" );
        metro.addEdge( start, end );
        metro.addEdge( end, start );

        start = (MetroStation) stations.get( "Rosalyn" );
        end   = (MetroStation) stations.get( "Ballston-GMU" );
        metro.addEdge( start, end );
        metro.addEdge( end, start );

        start = (MetroStation) stations.get( "Ballston-GMU" );
        end   = (MetroStation) stations.get( "Vienna" );
        metro.addEdge( start, end );
        metro.addEdge( end, start );
    }

    // Print details of the Metro System

    public void print() {
        System.out.println("Washington DC Metro System");
        System.out.println("==========================");
        System.out.println( stations.toString () );
    }

    // ============================================================
    // Build metro system model and exercise graph algorithms ....
    // ============================================================

    public static void main(String args[]) {

        System.out.println("Create Washington DC Metro System Graph");
        System.out.println("=======================================");

        // Build model of metro stations and rail network ...

        MetroSystemGraph ms = new MetroSystemGraph("DC Metro");
        ms.metroStations();
        ms.metroNetwork();
        ms.print();

        // Retrieve and print details of individual metro stations ....

        System.out.println("");
        System.out.println("Print Metro Station Details");
        System.out.println("===========================");

        System.out.println( ms.stations.get(      "Greenbelt").toString() );
        System.out.println( ms.stations.get(    "Fort Totten").toString() );
        System.out.println( ms.stations.get( "L Enfant Plaza").toString() );

        // ====================================================
        // Graph Analysis ....
        // ====================================================

        // Part 1. Compute and print all the strongly connected components
        // of the directed graph

        StrongConnectivityInspector sci = new StrongConnectivityInspector( ms.metro );
        List stronglyConnectedSubgraphs = sci.stronglyConnectedSubgraphs();

        System.out.println("Strongly connected graph components:");
        System.out.println("========================================");

        for (int i = 0; i < stronglyConnectedSubgraphs.size(); i++) {
            System.out.println(stronglyConnectedSubgraphs.get(i));
        }
        System.out.println();

        // Part 2. Prints the shortest path from vertex i to vertex c. This
        // certainly exists for our particular directed graph.

        System.out.println("");
        System.out.println("Shortest path from \"Greenbelt\" to \"National Airport\"");
        System.out.println("=========================================================");

        MetroStation start01 = (MetroStation) ms.stations.get( "Greenbelt" );
        MetroStation   end01 = (MetroStation) ms.stations.get( "National Airport");
        List path01 = DijkstraShortestPath.findPathBetween( ms.metro, start01, end01 );
        System.out.println(path01 + "\n");

        System.out.println("Shortest path from \"Greenbelt\" to \"New Carrollton\"");
        System.out.println("========================================================");

        MetroStation start02 = (MetroStation) ms.stations.get( "Greenbelt" );
        MetroStation   end02 = (MetroStation) ms.stations.get( "New Carrollton");

        List path02 = DijkstraShortestPath.findPathBetween( ms.metro, start02, end02 );
        System.out.println(path02 + "\n");
    }
}
```

The abbreviated input/output is as follows:

```
Script started on Sun Apr  1 12:47:30 2012
prompt >> run11
Buildfile: /Users/austin/ence688r.d/java-code-graphs/build.xml

compile:
    [javac] /Users/austin/ence688r.d/java-code-graphs/build.xml:8:
    warning: ’includeantruntime’ was not set, defaulting to build.sysclasspath=last;
    set to false for repeatable builds
run11:
     [java] Create Washington DC Metro System Graph
     [java] =======================================
     [java] Creating metro stations...
     [java] Creating metro network...
     [java] Washington DC Metro System
     [java] ==========================
     [java] { Catholic University=MetroStation("Catholic University")
     [java]    Coordinates = (0.0,1.0)
     [java]    Parking = false
     [java]    Track   = { Red }
     [java] , Ballston-GMU=MetroStation("Ballston-GMU")
     [java]    Coordinates = (-6.5,0.0)
     [java]    Parking = false
     [java]    Track   = { Orange }

     ... metro stations removed from the output ...

     [java] , New Carrollton=MetroStation("New Carrollton")
     [java]    Coordinates = (5.0,8.0)
     [java]    Parking = true
     [java]    Track   = { Orange }
     [java] , Vienna=MetroStation("Vienna")
     [java]    Coordinates = (-12.0,0.0)
     [java]    Parking = true
     [java]    Track   = { Orange }
```

```
[java] , College Park=MetroStation("College Park")
[java]    Coordinates = (5.0,10.0)
[java]    Parking = true
[java]    Track   = { Green }
[java] }
[java]
[java] Print Metro Station Details
[java] ===========================
[java] MetroStation("Greenbelt")
[java]    Coordinates = (5.0,12.0)
[java]    Parking = true
[java]    Track   = { Green }
[java]
[java] MetroStation("Fort Totten")
[java]    Coordinates = (0.0,2.0)
[java]    Parking = true
[java]    Track   = { Red Green }
[java]
[java] MetroStation("L Enfant Plaza")
[java]    Coordinates = (-1.0,-1.0)
[java]    Parking = false
[java]    Track   = { Green Yellow Orange }
[java]
[java] Strongly connected graph components:
[java] ========================================
[java] ([MetroStation("Catholic University")
[java] , MetroStation("Ballston-GMU")
[java] , MetroStation("DuPont Circle")
[java] , MetroStation("National Airport")
[java] , MetroStation("Metro Center")
[java] , MetroStation("Smithsonian")
[java] , MetroStation("L Enfant Plaza")
[java] , MetroStation("Rosalyn")
[java] , MetroStation("Silver Spring")
[java] , MetroStation("Judiciary Sq")
[java] , MetroStation("Greenbelt")
[java] , MetroStation("Union Station")
[java] , MetroStation("Stadium-Armory")
[java] , MetroStation("Fort Totten")
[java] , MetroStation("New Carrollton")
[java] , MetroStation("Vienna")
[java] , MetroStation("College Park")
[java] ], [ ( MetroStation(   "Greenbelt"), MetroStation("College Park"),
[java]      ( MetroStation("College Park"), MetroStation("Greenbelt") ),

... details of graph edges removed ....

[java]      ( MetroStation("Vienna"), MetroStation("Ballston-GMU") ) ] )
[java]
[java] Shortest path from "Greenbelt" to "National Airport"
[java] ========================================================
[java] [ (      MetroStation("Greenbelt") : MetroStation("College Park") ),
[java]   (   MetroStation("College Park") : MetroStation("Fort Totten") ),
[java]   (    MetroStation("Fort Totten") : MetroStation("Judiciary Sq") ),
[java]   (   MetroStation("Judiciary Sq") : MetroStation("L Enfant Plaza") ),
     [java]   ( MetroStation("L Enfant Plaza") : MetroStation("National Airport") )
     [java] ]
     [java]
     [java] Shortest path from "Greenbelt" to "New Carrollton"
     [java] ========================================================
     [java] [ (      MetroStation("Greenbelt") : MetroStation("College Park") ),
     [java]   (   MetroStation("College Park") : MetroStation("Fort Totten") ),
     [java]   (    MetroStation("Fort Totten") : MetroStation("Judiciary Sq") ),
     [java]   (   MetroStation("Judiciary Sq") : MetroStation("L Enfant Plaza") ),
     [java]   ( MetroStation("L Enfant Plaza") : MetroStation("Stadium-Armory") ),
     [java]   ( MetroStation("Stadium-Armory") : MetroStation("New Carrollton") )
     [java] ]
     [java]

BUILD SUCCESSFUL
Total time: 3 seconds
prompt >> exit
prompt >>
Script done on Sun Apr  1 12:47:41 2012
```

Bibliography

[1] Butler J.A. *Designing GeoDatabases for Transportation*. ESRI Press, 2008.

[2] Chunithipaisan1 S., James P., Parker D. The Integration of Spatial Datasets for Network Analysis Operations. pages 123–132, Newcastle, UK, NE1 7 RU. DIS2004, August 2004. Department of Geomatics, University of Newcastle upon Tyne.

[3] Coelho M., Austin M.A., and Blackburn M. Distributed System Behavior Modeling of Urban Systems with Ontologies, Rules and Many-to-Many Association Relationships. *The Twelth International Conference on Systems (ICONS 2017)*, pages 10–15, April 23-27 2017.

[4] Coelho M., Austin M.A., and Blackburn M.R. Semantic Behavior Modeling and Event-Driven Reasoning for Urban System of Systems. *International Journal on Advances in Intelligent Systems*, 10(3 and 4):365–382, December 2017.

[5] Eastman C., Teicholz P., Sacks R., and Liston K. *BIM Handbook*. John Wiley and Sons, 2008.

[6] Gao J., Liu X., Li D., and Havlin S. Recent Progress on the Resilience of Complex Networks. *Energies*, 8:12187–12210, 2015.

[7] Zeiler M. *Modeling our World: The ESRI Guide to Geodatabase Design*. ESRI Press, 1999.

Index