LABORATORY 14

Weighted Graph ADT

OBJECTIVES

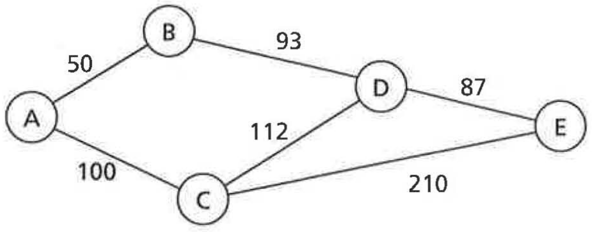
In this laboratory you

- create an implementation of the Weighted Graph ADT using a vertex list and an adjacency matrix.
- add vertex coloring and implement a method that checks whether a graph has a proper coloring.
- develop a routine that finds the least costly (or shortest) path between each pair of vertices in a graph.
- investigate the Four-Color Theorem by generating a graph for which no proper coloring can be created using less than five colors.

OVERVIEW

Many relationships cannot be expressed easily using either a linear or a hierarchical data structure. The relationship between the cities connected by a highway network is one such relationship. Although it is possible for the roads in a highway network to describe a relationship between cities that is linear (a one-way street, for example) or hierarchical (an expressway and its off-ramps, for instance), we all have driven in circles enough times to know that most highway networks are neither linear nor hierarchical. What we need is a data structure that lets us connect each city to any of the other cities in the network. This type of data structure is referred to as a **graph**.

Like a tree, a graph consists of a set of nodes (called vertices) and a set of edges. Unlike a tree, an edge in a graph can connect any pair of vertices, not simply a parent and its child. The following graph represents a simple highway network.



Each vertex in the graph has a unique **label** that denotes a particular city. Each edge has a **weight** that denotes the cost (measured in terms of distance, time, or money) of traversing the corresponding road. Note that the edges in the graph are **undirected**; that is, if there is an edge connecting a pair of vertices A and B, this edge can be used to move either from A to B, or from B to A. The resulting **weighted, undirected graph** expresses the cost of traveling between cities using the roads in the highway network. In this laboratory, the focus is on the implementation and application of weighted, undirected graphs.

Weighted Graph ADT

Elements

Each vertex in a graph has a label (of type String) that uniquely identifies it. Vertices may include additional data.

Structure

The relationship between the vertices in a graph are expressed using a set of undirected edges, where each edge connects one pair of vertices. Collectively, these edges define a symmetric relation between the vertices. Each edge in a weighted graph has a weight that denotes the cost of traversing that edge. This relationship is represented by an adjacency matrix of size $n \times n$, where n is the maximum number of vertices allowed in the graph.

Constructors and Methods

`WtGraph ( )`
Precondition:
None.
Postcondition:
Default Constructor. Calls setup, which creates an empty graph. Allocates enough memory for an adjacency matrix representation of the graph containing DEF_MAX_GRAPH_SIZE (a constant value) vertices.

`WtGraph ( int maxNumber )`
Precondition:
maxNumber > 0.
Postcondition:
Constructor. Calls setup, which creates an empty graph. Allocates enough memory for an adjacency matrix representation of the graph containing maxNumber vertices.

`void setup ( int maxNumber )`
Precondition:
maxNumber > 0. A helper method for the constructors. Is declared private since only WtGraph constructors should call this method.
Postcondition:
Creates an empty graph. Allocates enough memory for an adjacency matrix representation of the graph containing maxNumber elements.

`void insertVertex ( Vertex newVertex )`
Precondition:
Graph is not full.
Postcondition:
Inserts newVertex into a graph. If the vertex already exists in the graph, then updates it. If the vertex is new, the entire structure (both the vertex list and the adjacency matrix) is updated.

`void insertEdge ( String v1, String v2, int wt )`
Precondition:
Graph includes vertices v1 and v2.
Postcondition:
Inserts an undirected edge connecting vertices v1 and v2 into a graph. The weight of the edge is wt. If there is already an edge connecting these vertices, then updates the weight of the edge.

`Vertex retrieveVertex ( String v )`
Precondition:
None.
Postcondition:
Searches a graph for vertex v. If this vertex is found, then returns the vertex's data. Otherwise, returns null.

`int edgeWeight ( String v1, String v2 )`
Precondition:
Graph includes vertices v1 and v2.
Postcondition:
Searches a graph for the edge connecting vertices v1 and v2. If this edge exists, then returns the weight of the edge. Otherwise, returns an undefined weight.

`void removeVertex ( String v )`
Precondition:
Graph includes vertex v.
Postcondition:
Removes vertex v from a graph.

`void removeEdge ( String v1, String v2 )`
Precondition:
Graph includes vertices v1 and v2.
Postcondition:
Removes the edge connecting vertices v1 and v2 from a graph.

`void clear ( )`
Precondition:
None.
Postcondition:
Removes all the vertices and edges in a graph.

`boolean isEmpty ( )`
Precondition:
None.
Postcondition:
Returns true if a graph is empty (no vertices). Otherwise, returns false.

`boolean isFull ( )`
Precondition:
None.
Postcondition:
Returns true if a graph is full. Otherwise, returns false.

`void showStructure ( )`
Precondition:
None.
Postcondition:
Outputs a graph with the vertices in array form and the edges in adjacency matrix form (with their weights). If the graph is empty, outputs "Empty graph". Note that this operation is intended for testing/debugging purposes only.

LABORATORY 14: Prelab Exercise

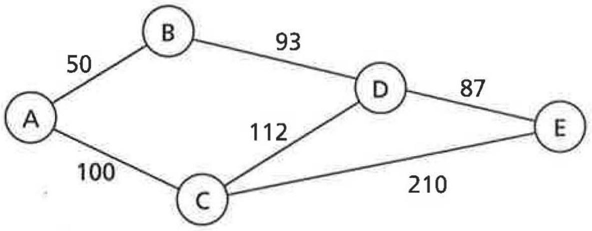
Name ____________________

Hour/Period/Section ____________________

Date ____________________

You can represent a graph in many ways. In this laboratory, you use an array to store the set of vertices and an **adjacency matrix** to store the set of edges. An entry (*j*, *k*) in an adjacency matrix contains information on the edge that goes from the vertex with index *j* to the vertex with index *k*. For a weighted graph, each matrix entry contains the weight of the corresponding edge. A specially chosen weight value is used to indicate edges that are missing from the graph.

The following graph



yields the vertex list and adjacency matrix shown below. A '–' is used to denote an edge that is missing from the graph.

Vertex List	
Index	Label
0	A
1	B
2	C
3	D
4	E

Adjacency Matrix					
From/To	0	1	2	3	4
0	—	50	100	—	—
1	50	—	—	93	—
2	100	—	—	112	210
3	—	93	112	—	87
4	—	—	210	87	—

Vertex A has an array index of 0 and vertex C has an array index of 2. The weight of the edge from vertex A to vertex C is therefore stored in entry (0, 2) in the adjacency matrix.

Step 1: Implement the operations in the Weighted Graph ADT using an array to store the vertices (`vertexList`) and an adjacency matrix to store the edges (`adjMatrix`). The number of vertices in a graph is not fixed; therefore, you need to store the actual number of vertices in the graph (`size`). Remember that in Java the size of the array is held in a constant called `length` in the array object. Therefore, in Java a separate variable (such as `maxSize`) is not necessary, since the maximum number of elements our graph can hold can be determined by referencing `length`—more specifically in our case, `vertexList.length`.

Base your implementation on the following incomplete definitions from the file *WtGraph.jshl*. The class Vertex (for the `vertexList`) is defined in the file *Vertex.java*. You are to fill in the Java code for each of the constructors and methods where only the method headers are given. Each method header appears on a line by itself and does not contain a semicolon. This is not an interface file, so a semicolon should not appear at the end of a method header. Each of these methods needs to be fully implemented by writing the body of code for implementing that particular method and enclosing the body of that method in braces.

```
public class Vertex
{
    // Data members
    private String label;                    // Vertex label

    // Constructor
    public Vertex( String name )
    {
        label = name;
    }

    // Class methods
    public String getLabel( )
    {
        return label;
    }

} // class Vertex

public class WtGraph
{
    // Default number of vertices (a constant)
    public final int DEF_MAX_GRAPH_SIZE = 10;
    // "Weight" of a missing edge (a constant) - the max int value
    public static final int INFINITE_EDGE_WT = Integer.MAX_VALUE;

    // Data members
    private int size;                        // Actual number of vertices in the graph
    private Vertex [ ] vertexList;           // Vertex list
    private int [ ][ ] adjMatrix;            // Adjacency matrix (a 2D array)
    // ------The following are Method Headers ONLY ------ //
    // each of these methods needs to be fully implemented

    // Constructors
    public WtGraph( )
    public WtGraph ( int maxNumber )

    // Class methods
    private void setUp( int maxNumber )                  // Called by constructors

    // Graph manipulation methods
    public void insertVertex ( Vertex newVertex )        // Insert vertex
    public void insertEdge ( String v1, String v2, int wt )// Insert edge
    public Vertex retrieveVertex ( String v )            // Get vertex
    public int edgeWeight ( String v1, String v2 )       // Get edge wt
    public void removeVertex ( String v )                // Remove vertex
    public void removeEdge ( String v1, String v2 )      // Remove edge
    public void clear ( )                                // Clear graph

    // Graph status methods
    public boolean isEmpty ( )                           // Is graph empty?
    public boolean isFull ( )                            // Is graph full?

    // Output the graph structure — used in testing /debugging
    public void showStructure ( )

    //  Facilitator methods
    private int index ( String v )                       // Converts vertex label to an
                                                         //  adjacency matrix index
    private int getEdge ( int row, int col )             // Get edge weight using
                                                         //  adjacency matrix indices
    private void setEdge ( int row, int col, int wt )    // Set edge wt using
                                                         //  adjacency matrix indices
} // class WtGraph
```

Your implementations of the public methods should use your `getEdge( )` and `setEdge( )` facilitator methods to access entries in the adjacency matrix. For example, the assignment statement

```
setEdge(2, 3, 100);
```

uses the `setEdge( )` method to assign a weight of 100 to the entry in the second row, third column of the adjacency matrix and the if statement

```
if ( getEdge(j, k) == WtGraph.INFINITE_EDGE_WT )
        System.out.println("Edge is missing from graph");
```

uses the `getEdge( )` method to test whether there is an edge connecting the vertex with index `j` and the vertex with index `k`.

Step 2: Save your implementation of the Weighted Graph ADT in the file *WtGraph.java*. Be sure to document your code.

LABORATORY 14: Bridge Exercise

Name ______________________________

Hour/Period/Section ______________________________

Date ______________________________

Check with your instructor as to whether you are to complete this exercise prior to your lab period or during lab.

The test program in the file *TestWtGraph.java* allows you to interactively test your implementation of the Weighted Graph ADT using the following commands.

Command	*Action*
`+v`	Insert vertex `v`.
`=v w wt`	Insert an edge connecting vertices `v` and `w`. The weight of this edge is `wt`.
`?v`	Retrieve vertex `v`.
`#v w`	Retrieve the edge connecting vertices `v` and `w` and output its weight.
`-v`	Remove vertex `v`.
`!v w`	Remove the edge connecting vertices `v` and `w`.
`E`	Report whether the graph is empty.
`F`	Report whether the graph is full.
`C`	Clear the graph.
`Q`	Quit the test program.

Note that `v` and `w` denote vertex labels (of type `String`) not individual characters (of type `char`). As a result, you must be careful to enter these commands using the exact format shown above—including spaces.

Step 1: Prepare a test plan for your implementation of the Weighted Graph ADT. Your test plan should cover graphs in which the vertices are connected in a variety of ways. Be sure to include test cases that attempt to retrieve edges that do not exist or that connect nonexistent vertices. A test plan form follows.

LABORATORY 14: In-lab Exercise 1

Name ______________________________

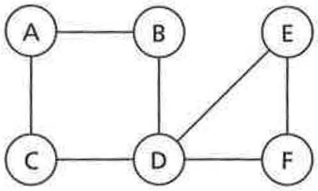
Hour/Period/Section ______________________________

Date ______________________________

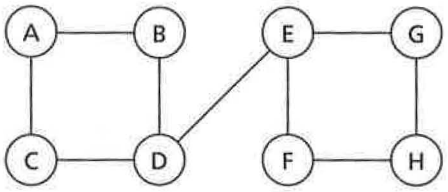
A communications network consists of a set of switching centers (vertices) and a set of communications lines (edges) that connect these centers. When designing a network, a communications company needs to know whether the resulting network will continue to support communications between *all* centers should one of these communications lines be rendered inoperative due to weather or equipment failure. That is, they need to know the answer to the following question:

> Given a graph in which there is a path from every vertex to every other vertex, will removing any edge from the graph always produce a graph in which there is *still* a path from every vertex to every other vertex?

Obviously, the answer to this question depends on the graph. The answer for the graph shown below is yes.



On the other hand, you can divide the following graph into two disconnected subgraphs by removing the edge connecting vertices D and E. Thus, for this graph the answer is no.



Although determining an answer to this question for an arbitrary graph is somewhat difficult, there are certain classes of graphs for which the answer is always yes. Given the following definitions, a rule can be derived using simple graph theory.

- A graph G is said to be **connected** if there exists a path from every vertex in G to every other vertex in G.
- The **degree** of a vertex V in a graph G is the number of edges in G that connect to V, where an edge from V to itself counts twice.

The rule states:

> If all of the vertices in a connected graph are of even degree, then removing any one edge from the graph will always produce a connected graph.

If this rule applies to a graph, then you know that the answer to the previous question is yes for that graph. Note that this rule tells you nothing about connected graphs in which the degree of one or more vertices is odd.

The following Weighted Graph ADT operation checks whether every vertex in a graph is of even degree.

```
boolean allEven ( )
```
Precondition:
The graph is connected.
Postcondition:
Returns true if every vertex in a graph is of even degree. Otherwise, returns false.

Step 1: Implement the `allEven` operation described above and add it to the file *WtGraph.java*.

Step 2: Save the file *TestWtGraph.java* as *TestWtGraph2.java*. Revise the TestWtGraph class name accordingly. Activate the '`D`' (degree) test in the test program *TestWtGraph2.java* by removing the comment delimiter (and the character '`D`') from the lines that begin with "`//D`".

Step 3: Prepare a test plan for this operation that includes graphs in which the vertices are connected in a variety of ways. A test plan form follows.

LABORATORY 14: In-lab Exercise 2

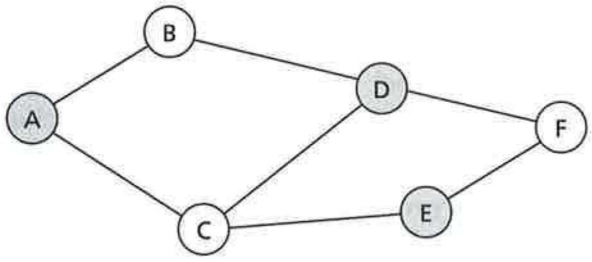
Name ______________________________

Hour/Period/Section ______________________________

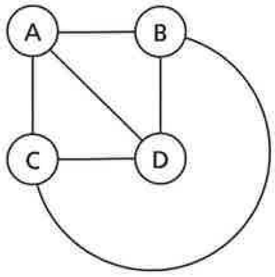
Date ______________________________

Suppose you wish to create a road map of a particular highway network. In order to avoid causing confusion among map users, you must be careful to color the cities in such a way that no cities sharing a common border also share the same color. An assignment of colors to cities that meets this criteria is called a **proper coloring** of the map.

Restating this problem in terms of a graph, we say that an assignment of colors to the vertices in a graph is a proper coloring of the graph if no vertex is assigned the same color as an adjacent vertex. The assignment of colors (gray and white) shown in the following graph is an example of a proper coloring.



Two colors are not always enough to produce a proper coloring. One of the most famous theorems in graph theory, the Four-Color Theorem, states that creating a proper coloring of any **planar graph** (that is, any graph that can be drawn on a sheet of paper without having the edges cross one another) requires using at most four colors. A planar graph that requires four colors is shown below. Note that if a graph is not planar, you may need to use more than four colors.



The following Weighted Graph ADT operation determines whether a graph has a proper coloring.

```
boolean properColoring ( )
```

Precondition:
All the vertices have been assigned a color.
Postcondition:
Returns true if no vertex in a graph has the same color as an adjacent vertex. Otherwise, returns false.

Step 1: Add the following data member to the Vertex class definition in the file *Vertex.java*.

```
private String color;    // Vertex color ("r" for red and so forth)
```

Also add the necessary methods to modify and access the vertex color.

Step 2: Implement the properColoring operation described above and add it to the file *WtGraph.java*.

Step 3: Replace the `showStructure( )` method in the file *WtGraph.java* with the `showStructure( )` method that outputs a vertex's color in addition to its label. An implementation of this `showStructure( )` method is given in the file *show14.txt*.

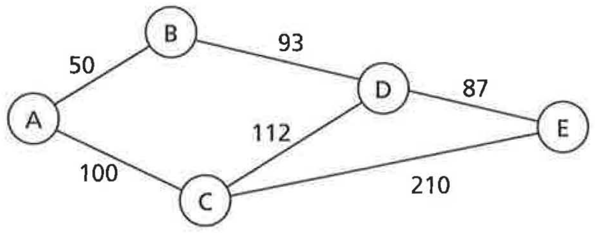
Step 4: Save the file *TestWtGraph.java* as *TestWtGraph3.java*. Revise the TestWtGraph class name accordingly. Activate the 'PC' (proper coloring) test in the test program *TestWtGraph3.java* by removing the comment delimiter (and the characters 'PC') from the lines that begin with "//PC".

Step 5: Prepare a test plan for the properColoring operation that includes a variety of graphs and vertex colorings. A test plan form follows.

LABORATORY 14: In-lab Exercise 3

Name	
Hour/Period/Section	
Date	

In many applications of weighted graphs, you need to determine not only whether there is an edge connecting a pair of vertices, but whether there is a path connecting the vertices. By extending the concept of an adjacency matrix, you can produce a **path matrix** in which an entry (j, k) contains the cost of the least costly (or **shortest**) path from the vertex with index j to the vertex with index k. The following graph



yields the path matrix shown below.

Vertex List	
Index	Label
0	A
1	B
2	C
3	D
4	E

Path Matrix					
From/To	0	1	2	3	4
0	0	50	100	143	230
1	50	0	150	93	180
2	100	150	0	112	199
3	143	93	112	0	87
4	230	180	199	87	0

This graph includes a number of paths from vertex A to vertex E. The cost of the least costly path connecting these vertices is stored in entry (0, 4) in the path matrix, where 0 is the index of vertex A and 4 is the index of vertex E. The corresponding path is ABDE.

In creating this path matrix, we have assumed that a path with cost 0 exists from a vertex to itself (entries of the form (j, j)). This assumption is based on the view that traveling from a vertex to itself is a nonevent and thus costs nothing. Depending on how you intend to apply the information in a graph, you may want to use an alternate assumption.

Given the adjacency matrix for a graph, we begin construction of the path matrix by noting that all edges are paths. These one-edge-long paths are combined to form two-edge-long paths by applying the following reasoning.

```
If there exists a path from a vertex j to a vertex m and
    there exists a path from a vertex m to a vertex k,
then there exists a path from vertex j to vertex k.
```

We can apply this same reasoning to these newly generated paths to form paths consisting of more and more edges. The key to this process is to enumerate and combine paths in a manner that is both complete and efficient. One approach to this task is described in the following algorithm, known as Warshall's algorithm. Note that variables `j`, `k`, and `m` refer to vertex indices, *not* vertex labels.

```
Initialize the path matrix so that it is the same as the edge matrix (all edges are paths).
Create a path with cost 0 from each vertex back to itself.

for ( m = 0 ; m < size ; m++ )
     for ( j = 0 ; j < size ; j++ )
          for ( k = 0 ; k < size ; k++ )
               if there exists a path from vertex j to vertex m and
                  there exists a path from vertex m to vertex k,
               then add a path from vertex j to vertex k to the path matrix.
```

This algorithm establishes the existence of paths between vertices but not their costs. Fortunately, by extending the reasoning used above, we can easily determine the costs of the least costly paths between vertices.

```
If there exists a path from a vertex j to a vertex m and
   there exists a path from a vertex m to a vertex k and
   the cost of going from j to m to k is less than entry (j,k) in the path matrix,
then replace entry (j,k) with the sum of entries (j,m) and (m,k).
```

Incorporating this reasoning into the previous algorithm yields the following algorithm, known as Floyd's algorithm.

```
Initialize the path matrix so that it is the same as the edge matrix (all edges are paths).
Create a path with cost 0 from each vertex back to itself.
for ( m = 0 ; m < size ; m++ )
     for ( j = 0 ; j < size ; j++ )
          for ( k = 0 ; k < size ; k++ )
               If there exists a path from vertex j to vertex m and
                  there exists a path from vertex m to vertex k and
                  the sum of entries (j,m) and (m,k) is less than entry (j,k) in the path
                     matrix,
              then replace entry (j,k) with the sum of entries (j,m) and (m,k).
```

The following Weighted Graph ADT operation computes a graph's path matrix.

> `void computePaths ( )`
> **Precondition:**
> None.
> **Postcondition:**
> Computes a graph's path matrix.

Step 1: Add the data member

```
private int [ ][ ] pathMatrix;    // Path matrix (a 2D array)
```

to the WtGraph class definition in the file *WtGraph.java*. Revise the WtGraph constructors as needed.

Step 2: Implement the computePaths method described above and add it to the file *WtGraph.java*. You will probably also want to implement facilitator methods for path similar to those used for edge.

Step 3: Replace the `showStructure( )` method in the file *WtGraph.java* with a `showStructure( )` method that outputs a graph's path matrix in addition to its vertex list and adjacency matrix. An implementation of this `showStructure( )` method is given in the file *show14.txt.*

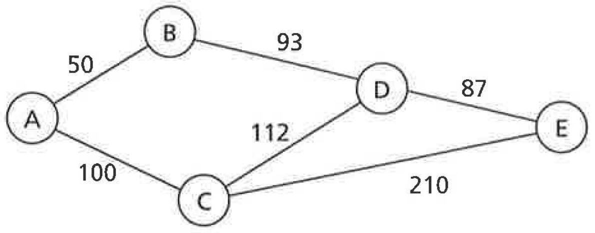
Step 4: Save the file *TestWtGraph.java* as *TestWtGraph4.java*. Revise the TestWtGraph class name accordingly. Activate the '`PM`' (path matrix) test in the test program *TestWtGraph4.java* by removing the comment delimiter (and the characters '`PM`') from the lines that begin with "`//PM`".

Step 5: Prepare a test plan for the `computePaths` operation that includes graphs in which the vertices are connected in a variety of ways with a variety of weights. Be sure to include test cases in which an edge between a pair of vertices has a higher cost than a multiedge path between these same vertices. The edge CE and the path CDE in the graph shown above have this property. A test plan form follows.

LABORATORY 14: Postlab Exercise 1

Name	
Hour/Period/Section	
Date	

Floyd's algorithm (In-lab Exercise 3) computes the shortest path between each pair of vertices in a graph. Suppose you need to know not only the cost of the shortest path between a pair of vertices, but also which vertices lie along this path. At first, it may seem that you need to store a list of vertices for every entry in the path matrix. Fortunately, you do not need to store this much information. For each entry (j, k) in the path matrix, all you need to know is the index of the vertex that follows j on the shortest path from j to k—that is, the index of the second vertex on the shortest path from j to k. The following graph, for example,



yields the augmented path matrix shown below.

Vertex List	
Index	Label
0	A
1	B
2	C
3	D
4	E

Path Matrix (Cost | Second Vertex on Shortest Path					
From/To	0	1	2	3	4
0	0|0	50|1	100|2	143|1	230|1
1	50|0	0|1	150|0	93|3	180|3
2	100|0	150|0	0|2	112|3	199|3
3	143|1	93|1	112|2	0|3	87|4
4	230|3	180|3	199|3	87|3	0|4

Entry (0, 4) in this path matrix indicates that the cost of the shortest path from vertex A to vertex E is 230. It further indicates that vertex B (the vertex with index 1) is the second vertex on the shortest path. Thus the shortest path is of the form AB...E.

Explain how you can use this augmented path matrix to list the vertices that lie along the shortest path between a given pair of vertices.