CSI2110 - Assignment 4 (100 points, weight 7.5%)

The assignment must be uploaded to Brightspace by Monday July 19th at 10:00 pm. Late assignments are accepted for a maximum of 24 hours and they will receive a 30% penalty.

Part 1 – Programing (60 Points)

Background

The programmer that you have helped in assignment two is back, and he is as confused as ever! Unfortunately, you have to help him again.

Here is the scoop: The programmer is working on a simulator tool that requires him to implement graphs that will be later used to represent complex computer networks. He purchased the Data Structures and Algorithms book by Goodrich and Tamassia that provides an implementation of a graph. To his chagrin, the supplied implementation is based on **Adjacency Lists**. His boss wants him to implement the graph using an **Adjacency Matrix**. He has no idea how to do it, but hopefully you do, since you are in charge of helping him.

Current Implementation

The programmer has simply taken the graph implementation provided in the Structures and Algorithms book and used it (**this is the same class you have seen in Lab 9**).

The operation "***areAdjacent***" (that checks whether two vertices are connected by an edge) is the most crucial one for the programmer's application, since it will be extensively used to check whether two network nodes are connected. To implement this function more optimally, the current implementation has to be replaced by one that uses an Adjacency Matrix.

Your Task

You are in charge of finishing the programmer's work again. Therefore, you are responsible for completing the implementation of the class `AdjacencyMatrixGraph` that the programmer had started. In summary, you need to implement the following methods:

```
[1]     public Vertex<V> opposite(Vertex<V> v, Edge<E> e)
[2]     public boolean areAdjacent(Vertex<V> u, Vertex<V> v)
[3]     public Vertex<V> insertVertex(V o)
[4]     public Edge<E> insertEdge(Vertex<V> v, Vertex<V> w, E o)
[5]     public V removeVertex(Vertex<V> v)
[6]     public E removeEdge(Edge<E> e)
```

Alice, the architect on the team has provided you with the following simplified UML class diagram that explains the relationship between the different classes that come into play (see Figure 1). She has also provided you with an implementation of an Expandable Square Matrix called, obviously, `ExpandableSquareMatrix`. This class can be used to create a matrix that can be easily increased or

decreased in size. Obviously, this is necessary whenever a vertex is added or removed from the graph. Some of the methods provided in this class are described in Table 1.

NOTE

The implementation of the class `AdjacencyListGraph` is included in the code package that you will get for Assignment 4/5. This will help you in your implementation of the `AdjacencyMatrixGraph` methods as you can reuse some of the same ideas.

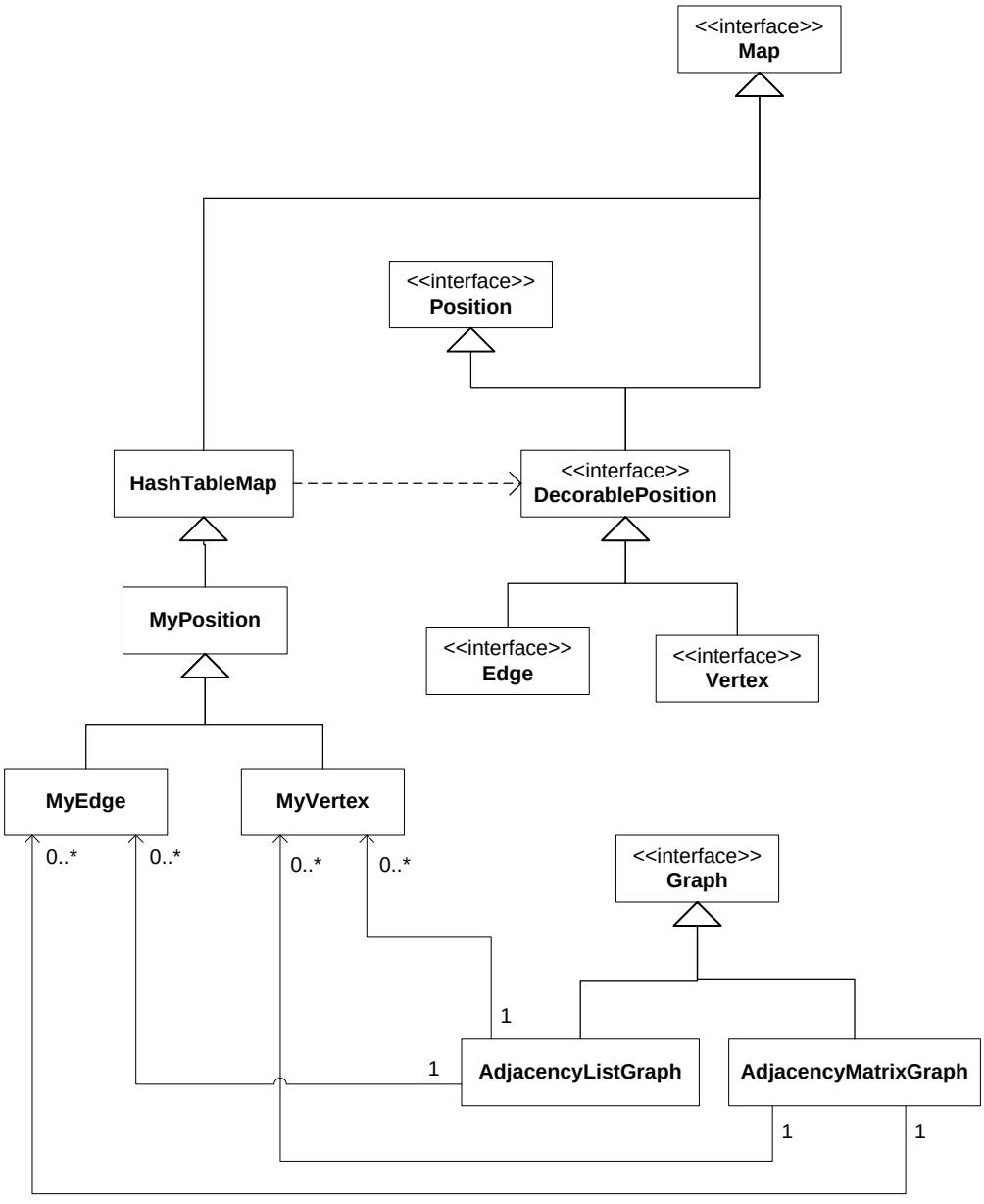


Figure 1: Simplified UML Class Diagram of the Some of the Important Classes

Method	Description
ExpandanbleSquareMatrix()	This is the Constructor. It creates an Array List of Array Lists to represent a matrix. Initially, the order of the matrix is set to 0. Therefore, the structure represents a 0×0 matrix.
increaseOrder()	Increases the dimension of the matrix from n × n to (n+1) × (n+1).
getOrder()	Returns the order of the matrix. For an n × n matrix, this method return n.
***set** (int r, int c, T t)*	Stores object "t" at position (r,c) in the matrix.
***get** (int r, int c)*	Gets the object stored at position (r,c) in the matrix.
***removeColumnRow**(int i)*	Removes column "i" and row "i" from the matrix.

Table 1 – Some of the Methods Provided in the `ExpandableSquareMatrix Class`

Testing the Implementation

Run the program from the `GraphTest` class. This program reads a list of edges from a text file and builds a graph accordingly. The file Assignment4_5.zip contains two text files that you can use as input to your program:

- graph.txt
- graph2.txt

One of these input files must be provided to the program as argument. To achieve this in eclipse, you can perform the following steps:

1. Right click on the class GraphTest
2. Go to Run As→Run Configurations
3. In the window Run Configurations, on the left panel, double click on Java Application
4. In the window Run Configurations, click on the tab arguments
5. In the text field titled Program arguments, write graph.txt (or graph2.txt)
6. Click on the button Run

This will print the vertices and edges contained in your graph. Then, using the command line interface, you can add or remove vertices and edges from the graph using the commands summarized in Table 2. Currently, the `GraphTest` class uses the `AdjacencyListGraph` implementation. This would be useful to initially familiarize yourself with the test environment. Once, your implementation of the `AdjacencyMatrixGraph` is completed, change the `GraphTest` code to use your new implementation.

Command	Description
`insert vertex v`	Insert vertex “v” into the graph where ‘”v” is any string representing the element of a vertex to be inserted into the graph.
`remove vertex v`	Remove vertex “v” into the graph where “v” is any string representing the element of a vertex to be removed from the graph.
`insert edge v w`	Insert an edge connecting vertices “v” and “w” into the graph where ‘”v” and “w” are strings representing the elements of vertices “v” and “w”.
`remove edge v w`	Remove an edge connecting vertices “v” and “w” from the graph where ‘”v” and “w” are strings representing the elements of vertices “v” and “w”.

Table 2 – Commands Summary

Figure 2 provides a set of commands you can use with the provided input class “graph.txt” along with the expected output (to use as a reference). The user inputs are emphasized in bold characters.

```
This graph has 10 vertices:

A

D

B

C

E

H

I

F

G

J

It also has 11 edges:

A to D

A to B

B to C

D to E

H to I

I to F
```

```
B to F

C to G

G to J

F to J

F to A

Enter command: insert vertex Z

This graph has 11 vertices:

A

D

B

C

E

H

I

F

G

J

Z

It also has 11 edges:

A to D

A to B

B to C

D to E

H to I

I to F

B to F

C to G

G to J

F to J

F to A

Enter command: insert edge A Z

This graph has 11 vertices:

A

D

B
```

```
C

E

H

I

F

G

J

Z

It also has 12 edges:

A to D

A to B

B to C

D to E

H to I

I to F

B to F

C to G

G to J

F to J

F to A

A to Z

Enter command: remove vertex D

This graph has 10 vertices:

A

B

C

E

H

I

F

G

J

Z

It also has 10 edges:

A to B
```

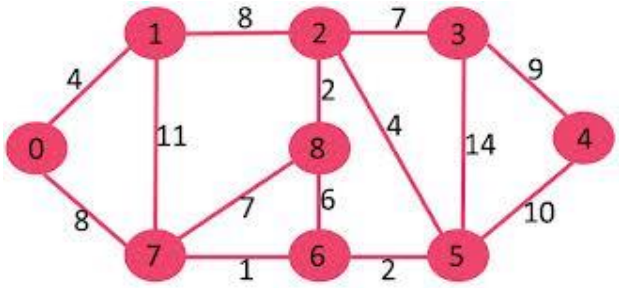
```
B to C

H to I

I to F

B to F

C to G

G to J

F to J

F to A

A to Z

Enter command: remove edge A B

This graph has 10 vertices:

A

B

C

E

H

I

F

G

J

Z

It also has 9 edges:

B to C

H to I

I to F

B to F

C to G

G to J

F to J

F to A

A to Z
```

Figure 2 – Sample of the type of Inputs that can be entered into the Test Class

Part 2 – Theory (40 Points)

2.1 (20 points) Love At First Sight (LAFS) is a dating network that connects customers to *n* coffee shops spread throughout Ottawa. Each customer register to his or her nearest coffee shop. The network of coffee stations is represented as an undirected graph where vertices represent coffee shops and two coffee shops are connected by an edge if there is a bus route between the coffee shops; the weight for each edge represents the average time to get from one coffee shop to the other by bus. Each node stores the list of customers in the dating network register to this coffee shop. You have been hired to write an application called **Love Is Near You (LINY)**: you have been asked to design an efficient algorithm (pseudocode) that, for each coffee shop, prints the information of all the customers that are 30 minutes away (on average) by bus from that coffee shop via the coffee shop network (note that the coffee shops sponsor the service, so bus routes that do not connect coffee shops are not considered). Analyse the running time of your algorithm based on *n,* the number of coffee shops, and *m,* the number of edges in the graph.

2.2 (20 points) Apply the DFS algorithm to the following graph starting from vertex 1. Assume that the incident edges are stored, for each vertex, in increasing order of their weight. List the vertices in order of visit. Clearly show the labels that will be assigned to each edge (Discovery or Back)




```
Algorithm DFS(G, v)
      setLabel(v, VISITED)
for all  e ∈ G.incidentEdges(v)  // Edges with lower weight value are considered first
      if  getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            if getLabel(w) = UNEXPLORED
                  setLabel(e, DISCOVERY)
                  DFS(G, w)
            else
                  setLabel(e, BACK)
```

Instructions

- For the Programing Part
    1. Download the `Assignment4.zip` and unzip it;
    2. Place the folder Assignment4 in your Eclipse workspace
    3. Create a new Java Project in Eclipse and call it Assignment4
    4. Run the program from the `GraphTest` class (specify either graph.txt or graph2.txt as argument)
    5. Complete the implementation of the `AdjacencyMatrixGraph` class
    6. After finishing the implementation, zip the Assignment4 project and submit it online.
- For the Theory Part, simply type or scan your solution and submit it online.

Note that in Brightspace, you can submit more than one file for a single assignment submission.