Comp 3761 Lab 7 Graph Implementation and Traversal

Submit your final Java code and print screen of your results for each part to Lab7 on D2L prior to the indicated due date.

PART I. In this exercise you will create an Adjacency Matrix class.

1. [2 mark] Write a class `AdjGraph` that implements a graph structure as an adjacency matrix. Initially your class will have two methods (addEdge and toString), and a constructor. You will use primitive array types. Do not use an ArrayList! Following are a few more details ...

```
// the constructor allocates space for a graph with V vertices
//(ie: it creates (minimally) a VxV matrix).
AdjGraph(int V);

// Note: if you want to have a more general implementation you should probably
// define Vertex and Edge objects, and make your Graph a VxV matrix of Edge objects
// (as opposed to a VxV matrix of ints

// the addEdge method allows you to define an edge from vertex u to vertex v
// where the vertices are identified by integers 0..V-1
void addEdge(int u, int v);
//
// alternatively you might do: void addEdge(Vertex u, Vertex v);

// the toString method returns the adjacency matrix as a string with one line of
// output for each vertex, for example:
// 0 1 1
// 1 0 0
// 1 0 0
// your method should print the graph exactly as shown, ie, with spaces and
// new lines after each row
String toString();
```

2. [1 mark] Write a test program that uses your `AdjGraph` class to create and print the 2 sample graphs shown below. You should have two separate files:

   GraphTest.java - contains a main function that create a new AdjGraph and runs tests etc.
   AdjGraph.java - implements the AdjGraph as described above

   In your main() you can hardcode the edges (for example, you might have the line `myGraph.addEdge(1,2)` to create and store an edge from vertex 1 to vertex 2).

	0	1	2	3	4
0	0	1	0	1	1
1	1	0	1	0	1
2	0	1	0	1	0
3	1	0	1	0	1
4	1	1	0	1	0

	0	1	2	3
0	0	1	0	0
1	1	0	1	0
2	0	1	0	1
3	0	0	1	0

3. [1 mark] Modify your graph class to include a method to return the degree of a vertex. Add some tests to make sure this works.

```
// the degree method returns the degree of the specified vertex, for example:
//    int d = myGraph.degree(0);  // assume myGraph is the first graph shown above
//    System.out.println("degree[0]="+ d); // prints "degree[0]=3"
```

```
int degree(int v);   // or you could have int degree(Vertex v); if you create a Vertex
class
```

4. [2 mark] Modify your graph class to support directed graphs. To do this you will add a variable `directed` which can be set from your test program, and a method `isDirected()` that returns the current value of this variable. Add some tests for this.

```
// example:
//       myGraph.directed = true;  // make myGraph a directed graph
// boolean d = myGraph.isDirected();
//       System.out.println("directed=" + d); // prints "directed=true"
boolean isDirected();
```

Note: you are also going to have to modify `addEdge` to work correctly for both directed and undirected graphs, and you are going to have to add two new methods, `inDegree()` and `outDegree()`.

PART II: In this exercise you will implement both a BFS and a DFS, and you will use them to solve some problems.

Background:

In lecture we covered in detail the pseudocode and algorithms for DFS and BFS. Today you will implement them. Use your adjacency matrix implementations of Graph from part 1.

1. [2 mark] Let's start with DFS. Pseudocode for DFS is given in your textbook, and in the course notes. You need to code one of these descriptions of DFS. Note that there are other implementations and pseudocode descriptions online – but I don't want you to use them. You should code the algorithm presented in class.

   Your DFS will be added to your Graph class (not Vertex or Edge), and it will implement both DFS() and dfs(v), as described in the pseudocode. You will break ties by choosing the vertex that comes next in lexicographic order.

   Test your DFS by having it print the vertex labels in the order that they are visited. For example, if the input graph is the 3D hypercube shown to the left, your DFS might produce:

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	0	**1**	**1**	0	**1**	0	0	0
1	**1**	0	0	**1**	0	**1**	0	0
2	**1**	0	0	**1**	0	0	**1**	0
3	0	**1**	**1**	0	0	0	0	**1**
4	**1**	0	0	0	0	**1**	**1**	0
5	0	**1**	0	0	**1**	0	0	**1**
6	0	0	**1**	0	**1**	0	0	**1**
7	0	0	0	**1**	0	**1**	**1**	0

```
visiting vertex 0
visiting vertex 1
visiting vertex 3
visiting vertex 2
visiting vertex 6
visiting vertex 4
visiting vertex 5
visiting vertex 7
```

2. [2 mark] Now you can code BFS. Again, use the pseudocode from the textbook, and display the order that the vertices are visited. Using the same input graph from step 2 you should get:

```
visiting vertex 0
visiting vertex 1
visiting vertex 2
visiting vertex 4
visiting vertex 3
visiting vertex 5
visiting vertex 6
visiting vertex 7
```