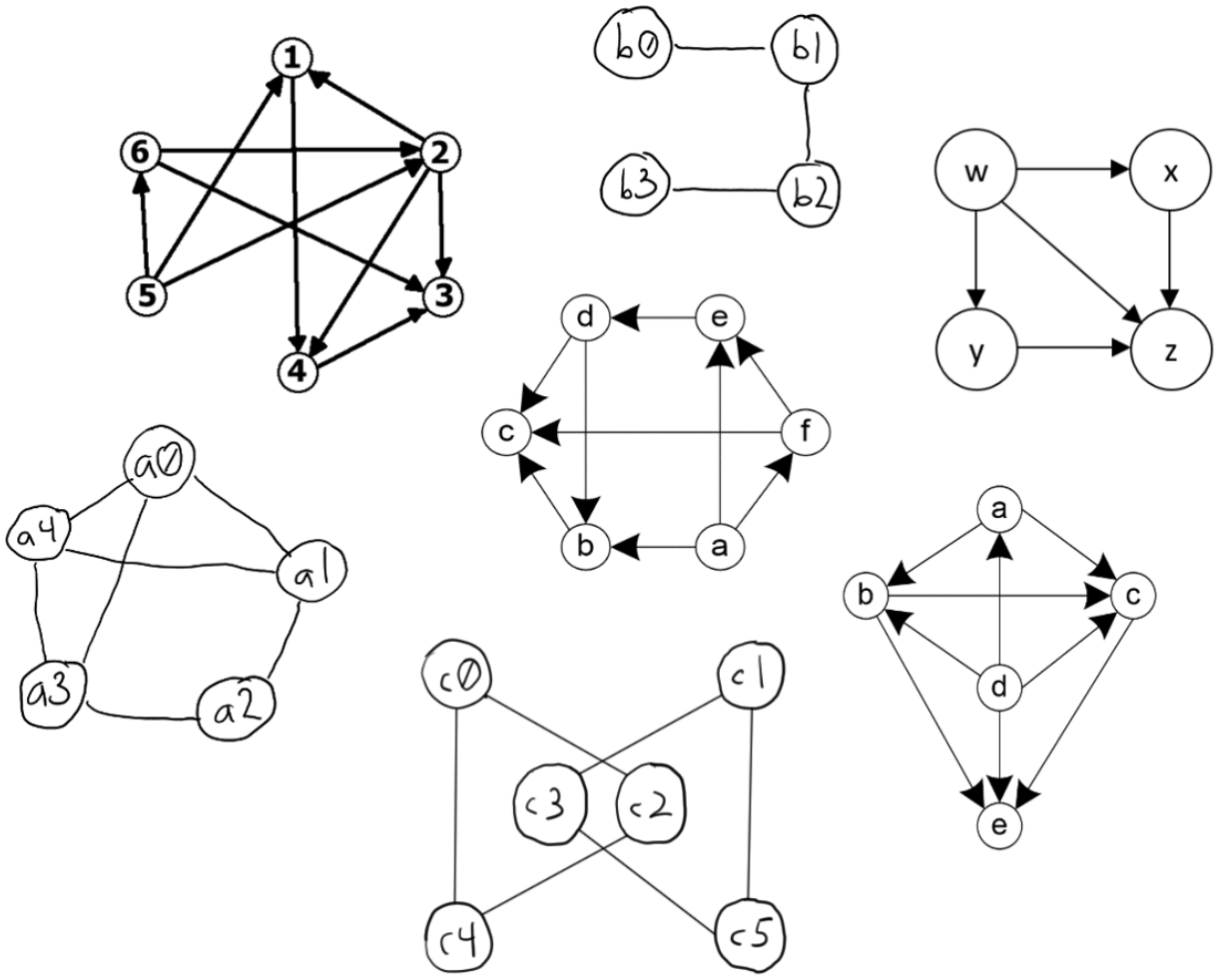
Assignment overview: GRAPHS!

This week it's time for fun with graphs! Here is a summary of the program you will write. There are full details in the sections below.

NOTE: There are lots of details in this lab. Be sure to read all the information thoroughly and carefully so that you don't overlook anything!

1. Write a class `Graph` that represents a graph.
    a. Vertices in the graph will have String names.
    b. Edges will be represented using an adjacency matrix.
    c. A graph can be either directed or undirected.
2. Include a method to perform Depth First Search on the graph.
3. Include a method to perform Breadth First Search on the graph.
4. Provide public methods both to perform the above operations and to provide a variety of information about the graph.



To-do list

Below is a summary/checklist of the *required* public methods. Details of all these methods are given further in the handout. This to-do list looks long! Do not be very alarmed—many of these methods are only a few lines of code.

- Class name: Graph
- Graph(String[ ] vertexLabels, boolean isDirected) – Constructor
- boolean isDirected() – getter
- void addEdge(String, String) – Adds an edge between two vertices
- int size() – returns the number of vertices
- String getLabel(int v) – returns the string name of the vertex numbered "v" internally
- String toString() – returns a string representation of the adjacency matrix
- void runDFS(boolean quiet) – performs Depth First Search
- void runDFS(String v, boolean quiet) – performs DFS starting at vertex v
- void runBFS(boolean quiet) – performs Breadth First Search
- void runBFS(String v, boolean quiet) – performs BFS starting at vertex v
- String getLastDFSOrder() – gets result of the most recently performed DFS
- String getLastDFSDeadEndOrder() – gets result of the most recently performed DFS
- String getLastBFSOrder() – gets result of the most recently performed BFS

You are free to use any private/helper functions that you wish.

Note: My testing program will be running some of these methods more than one time on each graph. If there are any counters or other "state-related" variables, be sure that they are re-initialized (or NOT) whenever it is necessary.

Detailed requirements

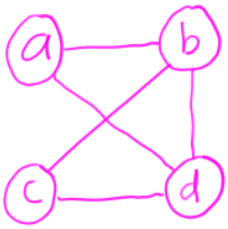
Inside your graph class you will represent a graph with the following private information:

- Array of Strings containing the names *aka* labels of the vertices. This can be an ArrayList, or even another Java collection type if you wish, but it is required that the "order" of the vertices is maintained in the same order that they were given in the Graph constructor.
- Adjacency matrix, an NxN array of ints, where N is the number of vertices.
- Boolean indicating whether graph is directed or undirected.
- Any other information that you find necessary or helpful in completing the requirements listed below.

Graph(String[ ] vertexLabels, boolean isDirected) – Constructor

vertexLabels[] is an array of Strings that represent the names of the vertices. The number of names in this array of this array will determine N, the size of the graph. This constructor will create an NxN adjacency matrix and initialize it to all zeroes. The isDirected flag indicates whether the graph is directed or undirected. All-zeroes means that for the moment the graph contains no edges. The calling program will add edges using the addEdge method later.

For example, if the following is your intended graph:



Then it can be initialized with the following code:

```
String[] vnames = { "a", "b", "c", "d" };
Graph G = new Graph(vnames, false);
```

After the graph is initialized, it will have no edges. These will be added subsequently with the addEdge() method.

void boolean isDirected() – getter

Returns true or false – true if the graph is a directed graph.

void addEdge(String A, String B) – Adds an edge between two vertices

Adds an edge from the vertex A to the vertex B. Note that if the graph is *undirected* this means setting *two* entries in the adjacency matrix to 1.

If either or both of the strings A and B are *not* present in the list of vertices, this function should quietly do nothing at all. No error messages, no exceptions thrown.

Continuing the example from above, the following sequence of calls will add the necessary edges to the graph:

```
G.addEdge("a", "b");
G.addEdge("a", "d");
G.addEdge("b", "c");
G.addEdge("b", "d");
G.addEdge("c", "d");
```

int size() – returns the number of vertices

Returns the number of vertices.

String getLabel(int v) – returns the string name of the vertex numbered "v" internally

You may assume that v is an integer in the range 0..N-1 inclusive. This method simply returns the name of the vertex numbered 'v' on your internal list of vertices (*i.e.* the vertex that was at position *v* in the array that was passed to the constructor when the graph was initialized).

String toString() – returns a string representation of the adjacency matrix

You are free to format this string any way you like. The only firm requirement is that it should include one line of "output" for each vertex, in the same order that they were listed in the graph when it was created.

Again continuing the example above, the following single String (containing some newline characters) would be an acceptable output for the graph:

```
a: 0 1 0 1
b: 1 0 1 1
c: 0 1 0 1
```

```
d: 1 1 1 0
```

Note: Vertex labels are of type String; as such, they might *not* be single letters/numbers.

void runDFS(boolean quiet) – performs Depth First Search

Runs the DFS algorithm on the graph. Use our default "first on the list" rule to decide which vertex to start/continue from. As the algorithm runs, save two lists of the vertices internally in your class. One of the lists will be the "DFS order" of the vertices – the order in which they are visited/encountered during the search. The other list will be the "finished order" (*aka* popped order or dead-end order) that occurs.

If the "quiet" flag is true, the method produces *no console output*. If the flag is false, the method will print out the sequence of vertices as they are visited in *DFS order*. For example, for the graph above, the output could be:

```
Visiting vertex a
Visiting vertex b
Visiting vertex c
Visiting vertex d
```

void runDFS(String v, boolean quiet) – performs DFS starting at vertex v

Exactly the same as the above method, except use the starting vertex given.

void runBFS(boolean quiet) – performs Breadth First Search

Runs the BFS algorithm on the graph. Use our default "first on the list" rule to decide which vertex to start/continue from. As the algorithm runs, save a list of the vertices *in BFS order* internally in your class.

If the "quiet" flag is true, the method produces *no console output*. If the flag is false, the method will print out the sequence of vertices as they are visited in *BFS order*. For example, for the graph above, the output could be:

```
Visiting vertex a
Visiting vertex b
Visiting vertex d
Visiting vertex c
```

You will need to implement or simulate a Queue in some way for this algorithm. As this is internal to your class, you are free to choose how to do this. One simple option is to use an ArrayList. It is fairly easy to add items to one end of an ArrayList (enqueue) and remove them from the other end (dequeue).

void runBFS(String v, boolean quiet) – performs BFS starting at vertex v

Exactly the same as the above method, except use the starting vertex given.

String getLastDFSOrder() – gets result of the most recently performed DFS

Returns a string containing the *DFS order* results of the most recently performed DFS (the last time that *either one* of the RunDFS() methods was called). You may format this string however you wish. If no RunDFS() has ever been called on this graph, this method should return an appropriate message instead.

String getLastDFSDeadEndOrder() – gets result of the most recently performed DFS

Returns a string containing the *dead-end order* results of the most recently performed DFS (the last time that *either one* of the RunDFS() methods was called). You may format this string however you wish. If no RunDFS() has ever been called on this graph, this method should return an appropriate message instead.

String getLastBFSOrder() – gets result of the most recently performed BFS

Returns a string containing the results of the most recently performed BFS (the last time that *either one* of the RunBFS() methods was called). You may format this string however you wish. If no RunBFS() has ever been called on this graph, this method should return an appropriate message instead.

More information, tips, etc.

Here is some data for a sample graph that you can use for testing purposes. It is the graph that appears in the lecture notes for "DFS example" and "BFS example" with vertices a..h. This is the sequence of addEdge() calls that you would make after creating the new Graph object called G.

```
G.addEdge("a", "b");
G.addEdge("a", "e");
G.addEdge("a", "f");
G.addEdge("b", "f");
G.addEdge("b", "g");
G.addEdge("c", "d");
G.addEdge("c", "g");
G.addEdge("d", "h");
G.addEdge("e", "f");
G.addEdge("g", "h");
```

Here are the DFS and BFS results for this graph:

```
DFS order traversal of graph:
Visiting vertex a
Visiting vertex b
Visiting vertex f
Visiting vertex e
Visiting vertex g
Visiting vertex c
Visiting vertex d
Visiting vertex h

BFS traversal of graph:
BFS visiting vertex a
BFS visiting vertex b
BFS visiting vertex e
BFS visiting vertex f
BFS visiting vertex g
BFS visiting vertex c
BFS visiting vertex h
BFS visiting vertex d
```

Note about directed graphs: The DFS and BFS algorithms from the lecture notes will work just as written for both directed and undirected graphs. Be sure to test your code with both kinds.

Note about connected, non-connected, and directed graphs: The DFS and BFS results for a graph should always be a list of **ALL THE VERTICES** in some order! If the input graph is not

connected, the "main part" of the DFS and BFS algorithm should (because of the main loop) make more calls to the "helper" function as needed. Similarly, if the input graph is *directed* but not all the vertices are "reachable" from the first vertex, then again, the "main part" should make additional calls to the helper function so that all the vertices are eventually visited.

Testing

I strongly urge you to test your code by using a main program that is in a separate driver class. This external main program will declare Graph objects and call the various required public methods. This is a great way to ensure that you are not inadvertently accessing private members, or calling private methods that are supposed to be public. In essence, you are testing your code the same way that I will be testing your code.

I also encourage you to test your code (and to study DFS and BFS) by creating several different graphs and running DFS and BFS on them. *My test program will be testing your code using several different graphs.*

- Some directed, some undirected
- Some connected, some not connected
- Some acyclic, some that contain cycles
- Some "extreme" cases (e.g. a graph with zero edges)

Submission information

Due date: As shown on Learning Hub.

Submit the following to the drop box on Learning Hub:

- Just your Java source code (*.java file), with the Graph class
- If you want to submit your driver class also, that's OK, but I don't need it, and might not look at it
- File name is not important.
- Please *do not zip* or otherwise archive your code. Plain Java files only.
- Please *do not submit* your entire project directory.

Marking information

This lab is worth 20 points.

Probably 4/20 of these points will be reserved for compliance with the COMP 3760 Coding Requirements.