Assignment objectives

Our primary objective is to implement Depth First Search and Topological Sorting for graphs.

To accomplish this we must also implement a Graph class.

Introduction

In a nutshell:

- Implement the Graph class as described in detail below, including the DFS and TopoSort algorithms.
- Test your code (of course!) thoroughly, with as many graphs as you can take the trouble to enter. As usual, you can do this within your class with a main() function (which won't be graded), or by using a separate driver class (which won't be graded). The purpose of this testing is for *you* to assure yourself that your Graph implementation is fully working.

Deadline note

Since the "normal" due date for this assignment would have fallen during Spring Break, it is extended to include all of Spring Break. In other words, the deadline for submission of this lab will be midnight, *Sunday night, March 20*.

Graph class summary

Class name: Graph

Public members:

Note that none of these functions perform any console output.

1. **Constructor: Graph (ArrayList<String>, boolean)** – The ArrayList is a list of names (labels) for the vertices in the graph. The boolean is a flag that indicates whether the graph is directed or undirected. (NOTE: Throughout this handout "N" refers to the number of vertices in the graph i.e. the length of this list, but you don't actually have to store this data value or use this variable name.)
2. **void AddEdge(String, String)** – The two arguments are names of vertices. You can assume that they are valid names that are present on the list of vertices. This method updates the adjacency matrix to indicate that there is an edge between these two vertices. Note: Exactly what this means is different for directed and undirected graphs!
3. **int size()** – returns N, the number of vertices, aka the length of the list of vertex names.
4. **String getVertex(int)** – returns the name of the i^{th} vertex on the list.
5. **boolean isDirected()** – returns true iff the graph is directed.
6. **int[][] adjMatrix()** – returns an array of ints representing the adjacency matrix of the graph.
7. **ArrayList<String> getDFSOrder(String)** – The String argument is one vertex in the graph, which is a designated "start vertex" for the traversal. This method returns a list of all the vertices of the graph in "DFS Order", i.e. the order in which they are encountered in a Depth First Search of the graph, *starting from the given vertex*. You may assume that

the given vertex name is valid (i.e. present in the graph). This method works for both directed/undirected graphs. NOTE: While doing your DFS it is *not necessary* to follow the "COMP 3760 vertex choice" rule about choosing the next vertex in alphanumerical order. Any valid DFS is acceptable (however, it must start with the given vertex).

8. **ArrayList<String> getTopologicalOrder(String)** – Same idea as the DFS method, but this returns a valid "Topological Sort Order" of the vertices. NOTE: If the graph is undirected, this method returns null. Technically, TopoSort also cannot be performed on graphs with cycles; but we will not worry about testing for that condition. You may assume that the graph is acyclic.

A few more Graph class details

Graph representation: Use an adjacency matrix to represent the edges of the graph. This should be a simple 2D array of ints (1 if an edge exists, 0 if it does not).

The constructor should: 1) Store the vertex labels internally to the Graph object; 2) Initialize an adjacency matrix of size NxN (initially the graph has zero edges); 3) Set a flag in the Graph object to indicate whether it is directed or undirected.

Building graphs for testing: While you are testing your DFS and TopoSort algorithms, you will need to build graph objects using the constructor + many calls to addEdge(). This may be a bit tedious. Feel free to invent alternatives that save yourself some effort (data file? overloaded addEdge() method?), but make sure that the two required public methods still exist.

About "visited" vertices: The DFS algorithm needs to track these. One easy way is to declare an array of booleans (as a private class member), in which the items correspond 1-1 with the vertices of the graph (the list of vertex names). Your traversal function(s) can re-initialize and re-use this "visited" array as needed.

You are free to write any additional methods (public or private helpers) and create any class members that you need or want to use.

Other notes:

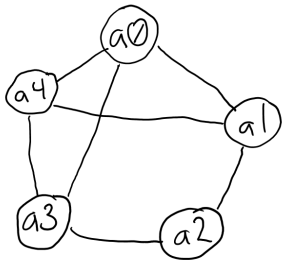
The following are some additional suggestions (*not required*, just brainstorming some possible ways to do things):

Write a toString() method that returns a representation of the graph suitable for output. What information is important? necessary? useful? helpful?

Write a private method that performs the actual DFS and stores the vertex ordering(s) in private variables. Then both of your (public, required) methods that need to do DFS can use this as a "helper function". Can you reduce the number of times DFS is performed this way? What happens if the graph itself is updated (more edges added) in between calls to DFS or TopoSort?

Some undirected graphs

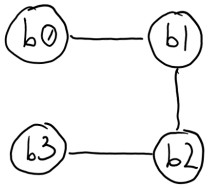
Undirected Graph A



This graph has *many* different valid DFS orders. Here are just two of them:

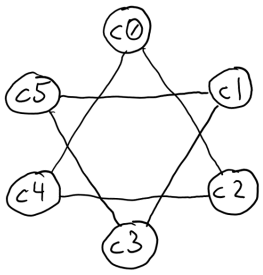
a0 a1 a2 a3 a4
a3 a2 a1 a4 a0

Undirected Graph B



I believe there are only 6 valid DFS orders of this graph. (But I have not really thought it through completely and carefully.)

Undirected Graph C



I believe this graph has 36 different valid DFS orders. (Same "but" applies.)

Undirected Graph D

Here is some Java code that should produce the example graph from the lecture notes ... except the vertices have been renamed. Instead of "a" ... "h", here they are "d0" ... "d7".

```
String[] vertices = {"d0", "d1", "d2", "d3", "d4", "d5", "d6", "d7"};
Graph G = new Graph(vertices, false);
G.addEdge("d0", "d1"); //ab in the lecture notes
G.addEdge("d0", "d4"); //ae
G.addEdge("d0", "d5"); //af
G.addEdge("d1", "d5"); //bf
G.addEdge("d1", "d6"); //bg
```

```
G.addEdge("d2", "d3"); //cd
G.addEdge("d2", "d6"); //cg
G.addEdge("d3", "d7"); //dh
G.addEdge("d4", "d5"); //ef
G.addEdge("d6", "d7"); //gh
```

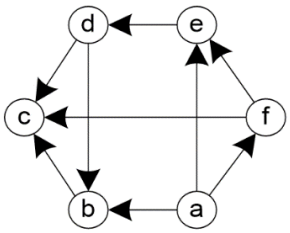
There are many valid DFS orders of this graph. The one we found in class ("abfegcdh") by following our alphanumerical convention for choosing the next vertex would be "d0 d1 d5 d4 d6 d2 d3 d7" with these vertex labels.

Some directed graphs

You can use these graphs to test BOTH DFS and TopoSort. I have not shown correct answers, but you can determine if you have produced a valid Topological Sort order by verifying that for every edge (arrow) in the original graph, the two corresponding vertices appear in the Topological Sort in that order.

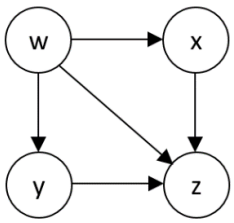
Directed Graph 1

The 6-vertex DAG example on the definitions page in the Topo Sort section of the lecture notes. Vertices are {a,b,c,d,e,f}. This also appears as Example 1 later in the slides.



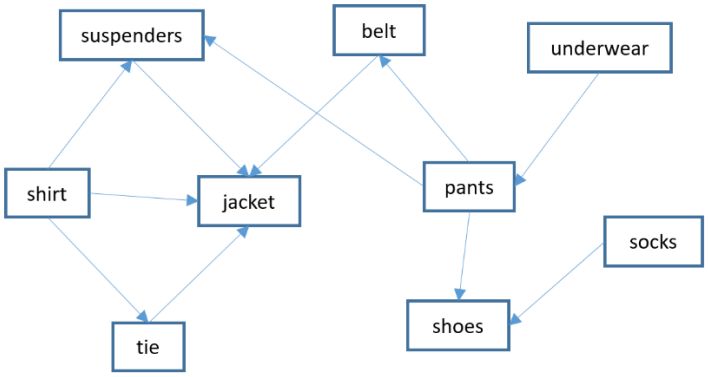
Directed Graph 2

The 4-vertex DAG example on the definitions page, vertices are {w,x,y,z}.



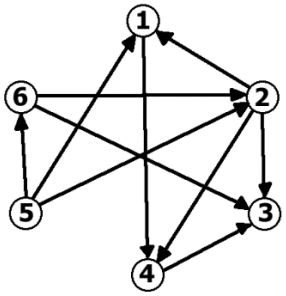
Directed Graph 3

The "how to get dressed" graph.



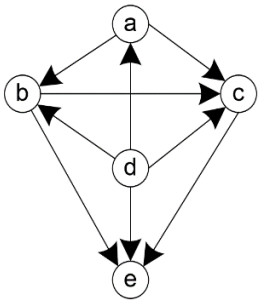
Directed Graph 4

Example 2 of the lecture notes, with vertices {1,2,3,4,5,6} (These are Strings, not numbers).



Directed Graph 5

Example 3 of the lecture notes, with vertices {a,b,c,d,e}.



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).
- File name is not important.
- Please *do not zip* or otherwise archive your code. Plain Java files only.
- Please *do not zip* or include your entire project directory.

Marking information

This lab is worth 20 points. Approximately 4-5 points of this will be allocated to conformance with the COMP 3760 coding guidelines.