COSC 320 - Advanced Data Structures and Algorithm Analysis
Lab 10

Dr. Joe Anderson

Due: 4 December 2018

1 Objectives

In this lab you will focus on the following objectives:

1. Review basic graph representations and operations
2. Develop familiarity with the `c++` standard library tools.
3. Implement depth-first search
4. Work with directed acyclic graphs (DAG)
5. Implement method to find strongly-connected graph components

2 Tasks

1. Put your code in a folder called "Lab-10". This folder will be zipped and turned in at the end.
2. Re-use your `Graph` class from Lab 9 with the following modifications (keep all other methods, such as `print`):
   (a) Modify the class to decide *at instantiation time* whether the graph is directed or undirected. This should be specified as a parameter to the constructor, and be immutable otherwise. Include a public method to report which type of graph it is.
   (b) The `addEdge` should add an edge between two vertices (if it doesn't already exist), and take into account whether the graph needs to be directed or undirected.
   (c) Add a method to report whether the graph is both directed and acyclic (contains no cycles). Use a modified DFS to determine whether it is acyclic; during the search, if any gray nodes are encountered, then the graph contains a cycle.
   (d) Add a method to first determine if the graph is directed and acyclic. If it is, perform a topological sort on the graph, reporting a valid topological ordering. A topological ordering (as will be discussed in lecture) is an ordering of the vertices $\{v_1, v_2, \ldots, v_n\}$ so that if $i > j$, then there cannot be a path in the graph from v_i to v_j (there may or may not be a path from v_j to v_i). You can determine a valid topological sort by ordering the vertex by decreasing finish times, after performing DFS. Note that this is only possible if the graph is acyclic! If there are cycles, an error should be reported.

   (e) Write a method to report each fully connected component of the graph using DFS. The high-level approach (discussed in lecture) is to first DFS the graph to compute the finish times, then reverse all the edges (using a copy of the graph with a copy constructor will be useful here!) and perform a DFS on the reversed graph (aka the transpose graph) where the nodes are visited *in order of descending finish time*. Each DFS tree in the forest computed by the second DFS contains exactly one fully connected component of the graph. The graph does not have to be acyclic for this procedure.

3. Write a test program to demonstrate (clearly) the correctness of each of the above functions.
4. Write your `main` function to read the graph adjacency list from a file for testing purposes. Be sure to document the format your program expects, and remember to submit the files you use to test your code.
5. Include a `Makefile` to build your code.
6. Include a `README` file to document your code, any interesting design choices you made, and **answer the following questions completely and thoroughly**:
   (a) Summarize your approach to the problem, and how your code addresses the abstractions needed.
   (b) What is the theoretical time complexity of your algorithms (best and worst case), in terms of the size of the tree? Be sure to vary the parameters enough to use the observations to answer the next questions!
   (c) How could the code be improved in terms of usability, efficiency, and robustness?

3 Submission

All submitted labs must compile with your provided `Makefile` and run on the COSC Linux environment.

Upload your project files to MyClasses in a single `.zip` file.

Turn in (stapled) printouts of your source code, properly commented and formatted with your name, course number, and complete description of the code.

Also turn in printouts reflecting several different runs of your program (you can copy/past from the terminal output window). Be sure to test different situations, show how the program handles erroneous input and different edge cases.

4 Bonus

(Up to 20 pts)

Modify your `Graph` class to use `c++` templates. To accomplish this, you can either store the data directly inside the graph nodes, but pay a performance/memory penalty in the methods that require moving whole nodes around by value (instead of `int` variables). The refactoring to accommodate this would also be quite cumbersome.

A better way is to copy the data into a `vector` or `map` and use the index of each one in the vector as an integer “key” of a graph node. For instance, if the user wants a graph on the strings `"dog"`, `"cat"`, and `"computer"`, you can store them in a `std::vector<std::string>`, and alias them in the graph with keys 0,1, and 2, respectively. This will also only require minimal modification of the graph algorithms themselves, because they will still operate on `int` valued nodes!