Topological Ordering and Minimum Spanning Trees

Out: 4/2
Due: 4/23 by 11:59 PM

Learning Objectives

- Checking whether a Digraph is Bipartite
- Implementing Prim's MST Algorithm Using a Binary Heap
- Implementing a Reverse Postorder DFS Topological Ordering Algorithm

This project involves the completion of a text-based menu-driven application that you began in project # 3. It involves writing three non-member functions (methods) to check whether an undirected graph is bipartite, to generate a topological ordering of the vertices of a directed acyclic graph (dag) using a reverse depth-first-search recursive algorithm or indicate that a topological ordering of the vertices is not possible because the digraph contains a cycle, and to find a minimum spanning tree of a simple connected undirected weighted graph using Prim's algorithm or a minimum spanning forest if the graph is not connected by applying Prim's algorithm to each component of the graph. One standard application of the minimum spanning tree algorithm is determining the lower bound on cost in a network. For example, a cable company may be interested in laying out cables between hubs in a city and minimizing cost. The hubs would be the vertices, the edges, the wires, and the lengths of the wires between them, the weights on the edges. A popular application of the topological sorting algorithm is in scheduling a sequence of jobs that require the observance of some precedence rules. The jobs are represented by vertices, and there is a directed edge from j_m to j_n if j_m must be performed before j_n. A notable computer science application is in instruction scheduling by CPUs. After this project, menu options 6-8 should work. The program will have the following user interface:

```
  BASIC WEIGHTED GRAPH APPLICATION
===========================================
[1] Incidence Matrix of G
[2] Floyd's Shortest Round Trip in G
[3] Postorder DFS Traversal of G Complement
[4] BFS Traversal of G of G Complement
[5] Check whether G is Bipartite
[6] Topological Ordering of V)G)
[7] Prim's Minimum Spanning Tree in G
[0] Quit
===========================================
```

Definition 1. A **bipartite** graph, also called a bigraph, is a graph that can be decomposed into two disjoint sets such that no two vertices within the same set are adjacent.

When option 5 is selected, your application will determine whether or not the input graph is bipartite. You will examine the adjacency relationships among pairs of vertices to determine whether the input graph is bipartite. This will require calls to the *isEdge* function (method) that you implemented in the previous project. A queue or stack may be used to keep track of vertices whose adjacency relationships have already been examined and additional secondary storage may be required to do bookkeeping of those relationships. Your implementation should work for connected and disconnected undirected graphs. When option 6 is selected, your application should generate a topological ordering of the vertices of the digraph if it contains no directed cycles. Your program calls a function (method) that implements the reversed post-order depth-first-search algorithm, whose pseudocode is given in the supplementary lecture notes, to generate a topological ordering of the vertices of the graph if one exists. Your implementation should explore the vertices of the graph in lexicographical order whenever the vertex being explored during the DFS traversal has several neighbors. When option 7 is selected, your program generate a minimum spanning tree or forest of an undirected weighted graph. To do this, your program calls a function (method) that uses a binary-heap as the underlying data structure for the priorty queue based implementation of Prim's MST algorithm. See files for the tasks that you need to complete as well as specifications of required functions/methods that you will implement to complete the application.

The executable file is **GraphDemo**. The input graph file will be a variation on the DIMACS network flow format file described on the project # 3 handout. For example, to run your application to find a topological ordering of a directed acyclic graph in a file called *cities3.wdg*, the file name is entered as a command line argument. The assumption is that the input file is in DIMACS format. The *readGraph* function (method) has already been implemented for you and it reads the file whose name you entered as a command line argument and creates a *Graph* instance.

Submitting Your Work

1. All source code files you submit must have a header with the following:

```
/**
* Describe the purpose of this file
* @author Programmer(s)
* @since 9999-99-99
* Course: CS3102.01
* Programming Project #: 4
* Instructor: Dr. Duncan
* @see the list files, if any, that this file references
*/
```

2. Verify that your code has no syntax error and that it is ISO C++11 or JDK 8 compliant prior to uploading it to the drop box on Moodle. Be sure to provide documentation, where applicable, for the additional methods that you have been asked to write. Also, add your name after the @author tag when you augment the starter code that I have provided. Put the last date modified in the header comments for each source code file.

3. Enclose all your source files-

   (a) for Java programmers, **GraphDemo.java**, **GraphAPI.java**, **Graph.java** and **City.java**.

   (b) for C++programmers, **GraphDemo.cpp**, **Graph.h**, **Graph.cpp** and **City.h**.

   - in a zip file. Name the zip file *YOURPAWSID_proj04.zip*, and submit your project for grading using the digital drop box on the course Moodle.