Basic Graph Algorithms

Out: 11/2
Due: 11/21 by 11:59 PM

> *Mankind invented a system to cope with the fact that we are so intrinsically lousy at manipulating numbers. It's called the* ***graph****.*
> [Charlie Munger]

Learning Objectives

- ⊙ Augmenting a Weighted Digraph ADT
- ⊙ Using Graph Traversal Algorithms
- ⊙ Generating a -1,0,1-Incidence Matrix of a Digraph
- ⊙ Checking Strong Connectivity in a Digraph
- ⊙ Topologically Ordering Vertices of a Directed Acyclic Graph
- ⊙ Implementing Kruskal's Minimum Spanning Tree Algorithm

There are many problems in computer science that can be modeled using graphs. This project involves the augmentation of an adjacency list implementation of a weighted digraph ADT and the completion of a menu-driven application that uses the ADT. The project involves writing both ADT and non-member functions (methods) to perform various tasks. One function/method that you will implement finds a minimum spanning tree of a simple undirected connected weighted graph using Kruskal's algorithm or a minimum spanning forest if the graph is not connected by applying Kruskal'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. You will also implement a function (method) to generate the topological ordering of vertices of a graph using an out-degree-based strategy. A popular application of a 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 $j_m \rightarrow j_n$ if j_m must be performed before j_n. A notable computer science application is in instruction scheduling by CPUs. Some of the functions (methods) have already been written for you; so you will simply have to properly call them. You will implement other functions (methods).

A simplifying assumption is that the weights on the edges are non-negative real numbers. I have provided starter code for you. Augment the starter code only where indicated in the code. The program will have the following text-based user interface:

```
  BASIC WEIGHTED GRAPH APPLICATION
===========================================
[1] Generate {-1,0,1} Incidence Matrix of G
[2] BFS/DFS Traversal of G
[3] Check Strong Connectedness of G
[4] Topological Ordering of V(G)
[5] Kruskal's Minimum Spanning Tree/Forest in G
[0] Quit
===========================================
```

Definition 1. The **incidence matrix** of a digraph gives the {-1,0,1}-matrix which has a row for each vertex and column for each edge, and (u,e)=1 and (v,e)=-1 if and only if (u,v) edge $e = (u, v)$ is a directed edge whose source vertex is u and whose destination vertex is v.

Definition 2. A **strongly connected digraph** is a directed graph in which there is a directed path between any pair of vertices.

Definition 3. A topological sort is a permutation p of the vertices of a graph such that an edge $\{i, j\}$ implies that i appears before j in p. Only acyclic digraphs can be topologically sorted.

When option 1 is selected, your application will generate a $|V(G)| \times |E(G)|$ matrix such that each column represents an edge in the graph. The source vertex of an edge has the value 1, its destination has the value -1, and all other entries in the column are 0. The columns of the matrix must be arranged in lexicographical order, from left to right, based on the endpoints of the edges.

The option will require you completion of the *isEdge* and *countEdges* functions (methods). When option 2 is selected, your application should generate and display a breadth-first-search and depth-first-search traversal of the input graph. Both traversal functions (methods) have already been implemented for you. You will need to invoke them using the lambda function that you will also define. Option 3 will require your completion of the *isStronglyConnected* function (method). This function (method) calls the isEdge function (method). This function implements the Kosaraju's algorithm: Begin an out-directed DFS (or BFS) traversal of the vertices of the graph from some vertex v. If all vertices are not reached, the graph is not strongly connected. If all vertices are reached, do an in-directed DFS (or BFS) traversal of the vertices of the graph from v. If all vertices are, again, reached, the graph is strongly connected. If all vertices are not reached during the traversal, the digraph is not strongly connected. In other words, a digraph D is strongly connected if and only if a traversal of a graph from a vertex $v \in D$ and a traversal of the transpose of the graph D' from the same vertex $v \in D'$ results in all vertices being reached in either case. When option 4 is selected, the application will generate and display a topological ordering of the vertices of the input graph, if one exist, or indicate that no such listing is possible because the graph contains a directed cycle. A directed cycle serves as an obstruction for generating a topological ordering of the vertices of a digraph. This option will require your implementation of the *topSortOutDeg* function (method) that uses the out-degree-based strategy for topologically sorting the vertices. When option 5 is selected, your program generates a minimum spanning tree or forest of an undirected weighted graph. To do this, your program calls a function (method) that uses a priority-queue-based implementation of Kruskal'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 **GraphExplorer**. The input weighted digraph file will be a variation on the DIMACS network flow format described below. DIMACS is the Center for Discrete Mathematics and Theoretical Computer Science based at Rutgers University.

- **Comments.** Comment lines give human-readable information about the file and are ignored by programs. Comment lines can appear anywhere in the file. Each comment line begins with a lower-case character **c**.

```
c This is an example of a comment line.
```

- **Problem line.** There is one problem line per input file. The problem line must appear before any node or edge descriptor lines.

  ```
  p NODES EDGES
  ```

  The lower-case character `p` signifies that this is the problem line. The `NODES` field contains an integer value specifying $|V|$, the number of vertices in the graph. The `EDGES` field contains an integer value specifying $|E|$, the number of edges in the graph.

- **Node Descriptors.** All node descriptor lines must appear before all edge descriptor lines.

  ```
  n ID LABEL
  ```

  The lower-case character `n` signifies that this is a node descriptor line. The `ID` field gives a node identification number, an integer between 1 and $|V|$. The `LABEL` field gives a string which serves as an alternate label for the vertex.

- **EDGE Descriptors.** There is one edge descriptor line for each edge in the graph. The edge descriptor line will be formatted as:

  ```
  e SRC DST WEIGHT
  ```

  The lower-case character **e** signifies that this is an edge descriptor line. For a directed edge (v, w), the `SRC` field gives the identification number for the source vertex v, and the `DST` field gives the destination vertex w. For an undirected edge, (v, w) and (w, v) refer to the same edge so only one edge descriptor line appears and on that line the end points of the edge are written in lexicographical order. Identification numbers are integers between 1 and $|V|$. The `WEIGHT` field contains $cost(v, w)$.

The input file name is entered as a command line argument. For example, to run your application on a graph file called cities1.wdg, enter *cities1.wdg* as a command line argument. The assumption is that file is in DIMACS format so no validation is done on the input file. The *readGraph* function (method) has already been implemented for you and it reads the input file and creates a *Graph* instance. Several sample weighted digraph files in DIMACS format

have been provided for you. Additional digraph files in DIMACS format may be used to test the application. I have provided four weighted digraph files in DIMACS format named *cities*[**1,7,16,17**]*.wdg*. I have also provided *cities*[**1,7,16,17**]*.pdf*, portable document format files that contain visual depictions of the digraphs described in the corresponding DIMACS formatted files. Additionally, I have provided four undirected weighted graphs *cities*[**4-5,14**]*.wug* and portable document format files representing their corresponding visual depictions, *cities*[**4-5,14**]*.pdf*. To test menu option 1-4, you may use any directed graph. To test menu option 5, use any undirected graph.

Submitting Your Work

1. Most of the documentation have been provided for you. For files that you augment, *Graph.*[*cpp*|*java*] and *GraphExplorer.*[*cpp*|*java*], add your name after the @author tag and change the date to the last date that you modified the code. The documentation will include this header as well as additional details.

```
/**
* Describe the purpose of this file
* @author Programmer(s), <YOUR NAME>
* @see the list files, if any, that this file references
* <pre>
* Date: TYPE LAST DATE MODIFIED
* Course: CS3102.01
* Programming Project #: 3
* Instructor: Dr. Duncan
* </pre>
*/
```

2. Verify that your code has no syntax error and that it is ANSI/ISO C++++14 or Java™JDK 8 compliant prior to uploading it to the drop box on Moodle. Be sure to exhaustively test your program using the sample digraph files as well as additional digraph files that you create, if need be.
3. Enclose your source files that you modify -
   (a) for Java programmers, **Graph.java** and **GraphExplorer.java**.
   (b) for C++ programmers, **Graph.cpp** and **GraphExplorer.cpp**.

   - in a zip file. Name the zip file **YOURPAWSID_proj03.zip**, and submit your project for grading using the digital drop box on the course Moodle.