Programming assignment 3

Due: May 4, 2:59 pm EST

Submit through canvas

Everything submitted to this programming assignment has to be entirely your own work. Open-source code/library is strictly not allowed unless as explicitly specified in the following description. Copy-paste or nearly copy-paste from others (students within the class, students from other/previous class, anyone else) is strictly not allowed. Please refer to the syllabus and the "Student Responsibilities and Statements of Academic Integrity & Honor Code" as you read in your reading assignment for details. Identical or near identical answers/report/code/documentation will be strictly considered as plagiarism.

In this assignment, you are asked to implement multiple graph algorithms. You are asked to submit everything (*.h and *.cpp files) that is required to fully compile and run your code.

1. Construct a graph from an input file. You will be provided with an input file which has the information of a directed graph. The file has following format: in the first line, there are two numbers m and n. The first number m is the number of vertices in the graph. The second number n is the number of edges in the graph. Then there will be n lines following. Each of the n lines has three numbers i, j and w, meaning that there is a directed edge from vertex i to vertex j with a weight w. An example of such an input file is as follows:

```
4 5
0 1 4
1 2 5
0 3 2
2 3 7
1 3 9
```

The example file represents a directed graph of 4 vertices (vertex 0, 1, 2 and 3) and 5 edges: the first edge is from vertex 0 to vertex 1 with a weight 4, the second edge is from vertex 1 to vertex 2 with a weight 5, etc.

You need to first read in the file and construct a directed graph from the file. You need to use adjacent lists to represent the graph and all the weights associated with edges internally. Implement a class Graph (submit Graph.h and Graph.cpp) which has a readGraph function that constructs a direct graph from the input file (10 points) and also an undirected version of the graph (i.e., remove all the directions along the edges. If there is a directed edge v->w and a directed edge w->v, then there should be only one undirected edge v-w between v and w in the undirected graph) (15 points), and a writeGraph function (10 points) that outputs the constructed directed graph into an output file Graph.out. The output file should have the following format, for the above example graph:

```
4 5
0: 1 4, 3 2
1: 2 5, 3 9
2: 3 7
3:
```

That is, the first line has the number of vertices and the number of edges. Then each of the following lines has the edges and their weights starting from a certain vertex. For example, the line

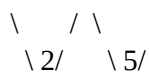
1: 2 5, 3 9

means for the vertex 1, there is a directed edge from 1 to 2 with weight 5, and a directed edge from 1 to 3 with weight 9. The output file is similar to what linked lists that you use for graph representation should look like.

You need to use your linked list class implemented in programming assignment 1 with proper modification to implement the adjacency list.

2. Implement the depth-first search algorithm on the constructed undirected graph. Your Graph class should have a function called DFS (20 points) that does a depth-first search. The DFS function should have a simple user interface (5 points) to allow the users of your program to specify which vertex to start with for the search. For example, your program prints out a prompt like
   "please specify a vertex for DFS: "
   and then gets the user input from stdio.
   Then your DFS function should output into stdio (5 points) all the vertices according to the orders they are visited during DFS. For example, for the above example graph, if vertex 1 is specified as the starting vertex, the output should be
   1 0 3 2
   That is, vertex 1 is first visited during DFS, and then vertex 0, vertex 3 and vertex 2
   (bonus credits 20 points) output the DFS search results in format of a spanning tree. For example, for the above BFS results, the output would look like
   1----0----3
   \
   \
   2
   That is, the output is a tree graphically. (You can design the art. If you consider your output is a "picture" of a certain size, then what you need to do is to figure out which pixel should be filled with which "color" and/or which number)
   You can use the standard C++ stack class.
3. Implement the breath-first search algorithm on the constructed undirected graph. Your Graph class should have a function called BFS (20 points) that does a breath-first search. The BFS function should have a simple user interface (5 points) to allow the users of your program to specify which vertex to start with for the search (similar to the interface for DFS). Then your BFS function should output into stdio (5 points) all the vertices according to the orders they are visited during BFS. For example, for the above example graph, if vertex 1 is specified as the starting vertex, the output should be
   1 0 2 3
   You can use the standard C++ queue class.
   (bonus credits 20 points) detect whether the graph is bipartite. You need to implement a function in your Graph class called isBipartite and output the detection results in the stdio (e.g., "the input graph is bipartite").
4. Implement a function in your Graph class called findArticulationPoints (30 points) that finds all articulation points in the graph. The function should output the results into stdio. For example, if there are no articulation points, then print out "there are no articulation points". If there are, then print out all the points.

(bonus credits 20 points) If there are articulations points, for each articulation point, implement a function in the Graph class called printComponents that prints out the graph components that come disconnected if the point is removed. For example, in the following graph



Vertex 3 is an articulation point, then you need to print out components 1-2-3 and 3-4-5 in the following format

3: 1 2 3, 3 4 5

If there are multiple articulation points, print all of them, one line for each point.

5. (bonus credits 30) implement a function called allPairsShortestPaths in your graph class that for each of all the vertex pairs in the undirected graph finds the length of the shortest path between the pair. The length of a path is defined as the sum of the weights with the edges along the path. A path from a vertex to itself is defined as of length 0. The allPairsShortestPaths function should output the results into an output file allPaths.txt in a format of a matrix in which the (i,j)-th element (the element in the i-th row and j-th column) saves the length of the shortest path from vertex i to vertex j. For example,

   0 3 5
   3 0 1
   5 1 0

   means that the shortest length from vertex 0 to vertex 1 is 3, from vertex 0 to vertex 2 is 5, and from vertex 1 to vertex 2 is 1.
6. What to submit
   a. Source code in a zipped fold called **Graph.zip**. You need to submit all the necessary codes, including the required files/classes and anything else that is required to correctly compile or run your code.
   b. Submit a main.cpp file which should implement the following pseudo code:

```
Main(){

readGraph();

writeGraph();

DFS(); /* ask the user to specify a starting vertex */

BFS(); /* ask the user to specify a starting vertex */

isBipartite(); /* bonus credits */

findArticulationPoints();

printComponents(); /* bonus credits*/

allPairsShortestPaths(); /* bonus credits */

}
```

   c. A README file contains a 3-sentence (5 points for each function) description of each function you have implemented. The description should include the KEY idea you have for the implementation. For example, for DFS, you description could be

i. use a stack to facilitate back-tracing;
ii. use an array visited[] to keep track whether a vertex has been visited or not;
iii. use an array orders[] to keep track the order in which a vertex is visited.

d. Your code should be developed on Linux and compile using g++. Please give a comment line you used to compile the code. For example, if you compile your code using
g++ you_code
then in the README file, please explicitly state this command. Please provide all necessary information in the README file. It is expected that the TA/graders can successfully compile and run your code just based on the README file.

7. Grading:
   a. Please strictly follow the naming scheme and use the names specified as in this description for the functions and files. If you are not using these names, 20 points will be taken off (non-negotiable!)
   b. Your code has to be able to compile on machine pegasus.cs.iupui.edu. If the compilation is not successful by copy-paste the compilation comment provided in your README file, one third of the total points will be taken off (non-negotiable!)