Comp 496ALG Project 3 (30 pts) Revised/Reduced 4/27/17

Due: May 11 ( 8:00 am for Moodle submission; in class for hard copy submission)

1. **GENERAL DIRECTIONS:** This is an individual project. In this project you will create a Graph class to implement several Graph algorithms for undirected weighted graphs. Neatness counts and so does indented code that is easy to read with helpful variable names. Your Graph class methods must match the specifications in the class specification below. In particular, all specified public methods must have the same signatures as specified. Otherwise the instructor test cases will not work. You may add any extra private methods and private fields as needed.
2. Create and test a general Graph class based on the specifications below. The Graph class should create an undirected weighted graph with n vertices and with m edges with nonnegative weights. Use an *adjacency list* to represent the graph. There will be two Graph constructors. One constructor will read the graph data from a file. The second constructor sets up a graph with n vertices. The users can then add edges using an addEdge method. Create a printGraph() method to print out your graph.

The Graph class should also implement the following graph algorithms from the text:

* DFS Traversal,
* Dijkstra’s algorithm for finding shortest paths
* Kruskal’s algorithm for finding a minimum spanning tree.

See further specifications below.

1. Specification of the Graph class

public class Graph

{

private ArrayList<EdgeNode> [] adjList ;

private int nVertices;

private int nEdges;

public Graph ( String inputFileName) { }//creates Graph from data in file

public Graph(int n ) { } //Creates a Graph with n vertices and 0 edges

public void addEdge(int i, int j, int weight) { } //adds an edge to the graph

public void printGraph() {

//prints nVertices, nEdges, and adjacency lists and total edge weight

// See format below in item 6.

}

public int get\_nVertices() {

public int get\_nEdges() { }

public int get\_TotalWeightOfEdges() { }

public Graph dfsTraversal ( int start) {

/\* Use recursion by calling a recursive dfs method. Visit all nodes. If graph is not connected you will need to call dfs more than once to visit all nodes .

Print the following information gleaned from the dfs traversal

* + - Print nodes in order visited
    - Connected? \_\_\_\_
    - NumberOfComponents? \_\_\_\_\_
    - Has a cycle? \_\_\_\_\_\_\_

If the graph is connected, return the spanning tree from the dfs traversal.

Otherwise, return null.

\*/

}

public void dijkstraShortestPaths (int start ) {

/\* Implement Dijkstra algorithm from text or class;

Use the Java PriorityQueue<PQNode> class. Use PQNode class below. The Java PriorityQueue class has no updateKey method. For our problem, just add a new updated node to the priority queue. This will work for Dijkstra’s algorithm since the new node has a smaller priority than the node you want to update. See Problem C-14.3 in text. An alternative is to remove the old node and add a new node.

Print shortest paths from vertex start to all other vertices reachable from start. Use format from class.

\*/}

public Graph kruskalMST() {

/\* Implement Kruskal algorithm from text or class.

You may assume that the graph is connected.

You may sort the edges or use a priority queue.

Use clusters.

Print the edges of the MST found and its total weight

Return the minimum spanning tree as a Graph

\*/

}

}

1. Specification for the EdgeNode class (Put in the same file as the Graph class)

class EdgeNode implements Comparable<EdgeNode>

{

int vertex1;

int vertex2;

int weight;

public EdgeNode ( int v1, int v2, int w)

{ vertex1 = v1, vertex2 = v2; weight = w;}

public int compareTo( EdgeNode node)

{

return weight – node.weight;

}

public String toString()

{

String s = "(";

s = s + vertex1 + "," +vertex2 + "," + weight + ")";

return s;

}

}

1. Specification for PQNode. Put in same file as the Graph class

class PQNode implements Comparable<PQNode>  
{  
 int vertex;  
 int distance;  
 public PQNode( int v, int z)  
 {  
 vertex = v;  
 distance = z;  
 }  
   
 public int compareTo(PQNode node)  
 {  
 return( distance - node.distance);  
 }  
   
 public String toString()  
 {  
 return "(" + vertex + "," + distance;  
 }  
 }

1. Notes on Graph Specification
   1. Read number of vertices and the edge pairs from an input file. The filename is passed in as a parameter to the constructor. The vertices will be numbered from 0 to n-1 where n is the number of vertices. The input file will be formatted like this (without the comments):

|  |
| --- |
| //number of vertices Vertices are labeled 0 .. nVertices-1  5  //edges with weight ; enter each edge twice in the adjacency list.  1 3 10  3 2 20  4 1 15  3 0 9  2 4 12  4 3 20 |

* 1. printGraph() prints the adjacency list, along with the number of vertices, the number of edges and the total weight of the edges in the graph.. Label. The print out for the graph above should look like

Graph: nVertices = 5 nEdges = 6 totalEdgeWeight = 86  
 Adjacency Lists   
 v= 0 [(0,3,9)]  
 v= 1 [(1,3,10), (1,4,15)]  
 v= 2 [(2,3,20), (2,4,12)]  
 v= 3 [(3,1,10), (3,2,20), (3,0,9), (3,4,20)]  
 v= 4 [(4,1,15), (4,2,12), (4,3,20)]

1. Test your Graph class thoroughly by creating your own text files with many different graphs. All output should be clearly labeled. An outline of a typical test program should look similar to this (but with more labeling) :

public static void main( String[] args)

{

Graph g = new Graph( “graph1.txt”);

g.printGraph();

int x,y,z;

//Set values of x, y and z to vertices in your graph

Graph tree = g.dfsTraversal( x);

if( tree != null)

tree.printGraph();

g.dijkstraShortestPaths(z);

Graph mstTree = g.kruskalMST();

if( mstTree != null)

mstTree.printGraph();

}

1. Your program should be nicely structured using classes and methods. No method should be longer than about 30 – 40 code lines. This will force you to break up your code into shorter, more readable methods.
2. Hand in as hard copy (stapled in this order)
   1. Cover Sheet
   2. Graph.java file with the Graph class , EdgeNode class and the PQNode class and any other helper classes you created.
   3. Labeled input and output from Instructors test case (posted May 2).
   4. Your name must be embedded in all electronic files ( not just written by hand)
3. Submit to Moodle by 8 am on May 11: Graph.java file. This file must include the Graph class , PQNode class and the EdgeNode class and any other helper classes you created.. Your name must be embedded in this file.