CMPT 225 - Coding Project

Due: the day before the final exam.
Submit: on Coursys.

This project contains a certain number of "mandatory" questions, which are fairly straightforward—a satisfactory grade for this project can be obtained by fulfilling these questions.

This project also contains additional questions, which are open-ended and can be used for additional credit. These questions are typically the type of tasks that you would have to undertake if you were implementing these algorithms for a specific project. These questions are purposefully vague, because what is complicated (and what we are trying to test for) is whether you would know how to understand what to do.

What the Java project contains:

- IHeap: an interface for heap-type class.
- AbstractBinHeap: an abstract class, implementing IHeap, providing all the utility protected methods required to implement a binary heap.
- MaxPQ: a maximum priority queue implementation, which uses a binary heap; this can be used to sort an array, by using the constructor which takes an existing array.
- MinPQ: a minimum priority queue.
- IDisjointSet and DisjointSet: the interface and implementation of a *Disjoint-Union* class (introduced in the textbook at p. 216-220), which may be required for certain graph algorithms; the implementation uses path compression and weighted union for maximum efficiency.
- Edge and EdgeWeightedGraph: to be completed using the textbook p. 610-611.

PRELIMINARY WORK

1) **[5 points]** Implement the Edge and EdgeWeightedGraph classes following the guidelines in pages 610 and 611 of the textbook. The class must have a constructor that takes a stream describing a graph as in the format described in p522 (this is exercise 4.3.9 p. 631).

2) **[5 points]** The *toString()* method must output the graph in the dot format which is described here:

http://www.tonyballantyne.com/graphs.html

3) **[10 points]** Create a method in your Main class

void drawGraph(String filename, Graph g)

which makes an external call to GraphViz (which you must install, it is available at graphviz.org), that generates an image file.

MINIMUM SPANNING TREE

Read pages 604 to 607 for an introduction on edge weighted graphs and minimum spanning trees. The goal here will be to implement two minimum spanning tree algorithms: Prim and Kruskal.

4) **[10 points]** Implement any version of the Prim (p. 616-623) or Kruskal (p. 624-627) algorithm. You may use any of the data structures provided in this project - MinHeap, MaxHeap, and DisjointSet (which the textbook calls UnionFind or UF).

Note: you will have to reuse one of your previous implementations of the linked-list to use as a Queue. You may additionally take the implementation from the textbook.

5) **[5 points extra]** Implement the other algorithm (if you implemented Prim, implement Kruskal, and vice-versa).

6) **[10 points extra]** Implement the reverse-delete algorithm described in exercise 4.3.24 p. 633 of the textbook.

TESTING

7) **[15 points]** Manually create a set of test graphs (at least ten different graphs), that should test various situations. For instance: what happens in a graph where all the weights are the same? what happens in a graph where all possible edges exist? what happens in a graph that is just one big cycle? what happens in a graph that is also a tree? Use this test graphs to evaluate whether your implementations are correct, and how efficient they are.

8) **[5 points extra]** Use your GraphViz function to visualize the minimum spanning tree generated by your algorithms: make it so that the edges that are picked for the minimum spanning tree are in a different style as the other ones.

9) **[10 points extra]** Create a random generator of graphs using the Erdos-Renyi model: fix the number of vertices to n, then take a probability p (between 0 and 1), and for each pair of vertices (u,v) decide that this is an edge with probability p. If you are using Prim's algorithm, the graphs need to be connected (that is, there should exist a path in the graph linking any two pairs of vertices):
- either modify your algorithm to work with non-connected graphs
- or manually add edges to the graphs that you generate to make them connected
- or pick p the probability that two vertices form an edge, large enough so that the graph is necessarily connected (this is usually true for $p > 1/2$).

10) **[5 points extra]** Use the random generator of graphs you implemented to plot the average performance of your algorithm in the following way: fix n, the number of vertices, and then show the average performance for different values of p.