Data Structures and Algorithms

Assignment #6

Due: Paper Assignment Wednesday May 31st in class
Due: Program Assignment Friday June 1st 11:00 am

This assignment will deal with graphs. In the programming part you will implement methods to find the shortest paths from a vertex to all other vertices (Dijkstra's algorithm) and to find the minimum spanning tree of a graph (you have the choice of either Prim or Kruskal's algorithm). You should start with the first two paper problems that you don't have to turn in. These two problems should be a good warm-up for the programming part.

Paper Assignment

1. Problem R-13.14 (Don't turn it in)

2. Problem R-13-17 (Don't turn it in but note that this is part of the output you will have to generate for the programming part).

3. Problem R-13-9

4. Give an algorithm in pseudocode that detects whether a directed graph has a cycle. Use the adjacency list representation for graphs. (Of course you can have additional field(s) for each node in addition to their name and pointers to successors.)

5. Give an algorithm that detects whether a directed graph is a forest, i.e., a set of one or more disjoint trees. Use the adjacency list representation for graphs. (Of course you can have additional field(s) for each node in addition to their name and pointers to successors.)

6. R-13-32 and R-13-33 (Recall the DFS and BFS that we did for topological sort)

Programming Part

Your input will be

- a set of vertices from the file **vertex.txt**. Each line of the file **vertex.txt** is a string of 3 characters representing an airport (e.g., SEA, ORD etc.)
- a set of edges from the file **edge.txt**. Each edge in **edge.txt** is represented in 3 consecutive lines namely, the first two lines being members of the vertex set, i.e., the names of airports, and the third line a real number representing the *distance* between the two airports.

Your first task is to build an adjacency list structure for the undirected graph. Vertices and edges are objects. The set implementations of vertices and edges are left to your choice but they should be structures on which you can apply iterators (e.g., array, linked list, Array list). Each vertex object points to its own incidence list. Each incidence list is itself a structure that should be iteratable. Each object in the incidence list points to an edge object. Each edge object has pointers to the vertex objects that it links as well as a distance component. See Figure 14.3 in your book for an illustration.

In the course of designing your program you might wish to add components to the objects such as markers etc.

Your second task is to find the shortest paths from one airport, namely SEA, to all other airports in vertex.txt. In addition, you should record the paths and their total distances. To do that you should implement *Dijkstra's algorithm with the initial vertex being SEA.* In order to test your implementation of Dijkstra's algorithm, you should provide a method *pathLengthToDest(Vertex dest)* that prints answers to queries asking for the path and the length of the path from SEA to the destination vertex (of course this method is called after you have run Dijkstra's algorithm). For example the call *pathLengthToDest(DEF)* should provide output like:

```
SEA     ABC    DEF  1386
```

where SEA, ABC, and DEF are airport names and the shortest path between SEA and DEF is a path of length 2 of distance 1386 with an intermediate "stop" in ABC.

While finding shortest paths from SEA to all other airports is convenient for Seattle inhabitants, airlines might have other criteria such as connecting all cities but with a minimum total distance. In other words, they want to see a *Minimum Spanning Tree*(MST) of the graph.

Your third task is to build an MST of the graph. You can use either Kruskal or Prim's algorithm. Your output should be a list of the edges in the order they have been added to the MST as in:

```
SEA     XYZ     1200
SEA     ABC      422
ABC    DEF       976
etc.
```

Implementation Notes
In the algorithms above, you will need to implement priority queues. Since the graph is rather sparse and the number of vertices is small, a sorted linked list would work quite well. However, for **extra credit** you can implement the priority queues using binary heaps, the best implementation for larger graphs.

You should feel free to use Java's ArrayList/Linkedlist. However you should implement your own priority queues, i.e., do not use Java's priority queue.

Files provided
In addition to **vertex.txt** and **edge.txt** you will be given a template (Java interfaces) for the methods that you have to implement as well as some testing code for the adjacency list and priority queue.