COMS 3134 Homework 5

Submission instructions

All programs must compile and run on CUNIX to receive credit. Submit your electronic files via `http://courseworks.columbia.edu`. We prefer electronic submission of theory, though you will not be penalized for paper submissions. (Do **not** print out your programs.) I recommend learning LaTeX for typesetting math. Include a README file that explains exactly what each file in your submission is. Place all the files you want to submit into a submission directory with the following naming scheme.

```
<your_uni>_hw<number>
```

So if my UNI is `uni1234` am submitting homework 6, my directory would be `uni1234_hw6`. Archive your submission directory using

```
tar -czvf uni1234_hw6.tar.gz uni1234_hw6
```

and upload `uni1234_hw6.tar.gz` to courseworks. (You will probably need to first download the file to a local directory using an FTP program. See CUNIX tutorial for more info.)

Multiple Submissions: You can submit multiple times, but we will only consider the latest submission based on the timestamp in courseworks. Please give at least 1-2 minutes between two submissions so we can tell which is the newest submission.

I recommend that you also keep a pristine copy of your submission folder in case there is any submission error.

Theory Problems

Make sure your solutions are clear. Diagrams and math are often insufficient to convey exactly what you mean, so supplement with some text. Either pseudocode or Java are acceptable when asked to provide algorithms. Nevertheless, clear, concise English is often preferable.

1. **Weiss 9.1** (5 points) Find a topological ordering for the graph in Figure 9.79. [Show your work as you execute the algorithm.]
2. **Weiss 9.2** (4 points) If a stack is used instead of a queue for the topological sort algorithm in Section 9.2, does a different ordering result? Why might one data structure give a "better" answer?
3. **Weiss 9.5**
   (a) (3 points) Find the shortest path from A to all other vertices for the graph in Figure 9.80. [Show your work as you execute the algorithm.]

(b) (3 points) Find the shortest unweighted path from B to all other vertices for the graph in figure 9.80. [Show your work as you execute the algorithm.]

4. **Weiss 9.15a**

   (a) (6 points; 3 points for each algorithm) Find a minimum spanning tree for the graph in Figure 9.82 using both Prim's and Kruskal's algorithms. [Show your work.]

5. **Weiss 9.31** (5 points) Give an algorithm to find in an undirected (connected) graph a path that goes through every edge exactly once in each direction.

6. **Weiss 9.38b** You are given a set of N sticks, which are lying on top of each other in some configuration. Each stick is specified by its two endpoints; each endpoint is an ordered triple giving its x, y, and z coordinates; no stick is vertical. A stick may be picked up only if there is no stick on top of it.

   (a) (**0 points**; I'm just including this for reference; don't do this, just assume you have the routine for the next part.) Explain how to write a routine that takes two sticks a and b and reports whether a is above, below, or unrelated to b. (This has nothing to do with graph theory.)

   (b) (4 points) [Assuming we have the routine from part (a), g]ive an algorithm that determines whether it is possible to pick up all the sticks, and if so, provides a sequence of stick pickups that accomplishes this.

Programming (30 points)

You are a city pigeon who wants to travel and see the world. Unfortunately, since you are addicted to city life, you cannot travel more than a certain distance without having to stop at another major city. You want to have a program that outputs the shortest paths between any two cities (with pit stops along the way). Luckily, you learned about the Floyd-Warshall All-Pairs-Shortest-Path algorithm from your Data Structures in Java class.

Given the coordinates of major cities around the world, as found in the include file `cities.txt`[1] load the cities and compute their distances under the assumption that the distance between any two cities proportional to

$$\sqrt{(\text{longitude}_1 - \text{longitude}_2)^2 + (\text{latitude}_1 - \text{latitude}_2)^2}.$$

(This is a bad assumption in real life because it essentially assumes the world is flat.)

Using this distance formula, construct a graph in which all cities within a radius of d degrees are adjacent and all cities with distance greater than d degrees are disconnected. Then compute the shortest paths (the cost and the actual path) for all pairs of cities. Your program should take as input the text file containing the city data and a number representing the number of longitude-latitude units you are able to travel without stopping. Then it should poll the user for a start city and an end city, and output the path and total distance between the two. Example usage would be as follows.

```
java ShortestPathFinder cities.txt 30
All paths computed
Enter the start city:
New York
Enter the destination city ('quit' to exit):
Paris
Shortest path:
Paris (48.86, 2.34) <- Malaga (36.72, -4.42) <- Dakar (14.72, -17.48)
<- Fortaleza (-3.78, -38.59) <- Maturin (9.75, -63.17)
<- Santo Domingo (18.48, -69.91) <- New York (40.67, -73.94)
Total path length: 129.18870704459715
```

Create a class `CityGraph.java` that stores the city graph and computes all-pairs-shortest-paths, and provides a lookup method for the shortest path between two cities. You should use a `HashMap` to look up graph nodes by their `String` names. Your main class should be called `ShortestPathFinder.java`, and it will provide the user interaction and call the functions provided in `CityGraph.java`.

[1] City data obtained from `http://gael-varoquaux.info/blog/?p=84`. The data format is tab-delimited and each row is: (city name) (longitude) (latitude).