Algorithms
Computer Science 140 & Mathematics 168
Instructor: B. Thom
Spring 2004
Homework 9b
Due on 3/30/04 (beginning of class)

1. **[10 Points] Network Reliability!**

   My-I'm-Soft has decided to buy the entire Internet and rename it My-I'm-Net. They've hired you to solve a particularly interesting routing problem in this domain. "Here's the story," says Gill Bates, propping his feet up on your desk (and again, drooling while he gnaws on a huge tootsie roll). "My-I'm-Net can be represented by an undirected graph where the vertices are the routers in the network and the edges represent network links between those routers. Each edge has a certain reliability, which is a real number between 0 and 1. The reliability is just the probability that the edge will be available for transmitting data. Since the reliabilities are statistically independent, the reliability of a path between two nodes is just the product of the reliabilities of the edges on that path. You will be given a $V \times V$ adjacency matrix such that the element at row i and column j of the matrix is the reliability of the edge between vertices i and j if it exists and 0 otherwise. (The values for all diagonal elements are one, representing the fact that, to go from some router back to itself is perfectly reliable.) Your job is to find an algorithm to compute the most reliable path between all pairs of vertices."

   (a) Explain how the Floyd-Warshall Algorithm can be modified to compute the reliability of the most reliable path between every pair of vertices and argue why your modification is correct.

   (b) What is the running time of your algorithm? Why?

2. **[35 Points] Implementing the Floyd-Warshall Algorithm!** In this problem you will implement the Floyd-Warshall (FW) shortest path algorithm in your favorite programming language.

   You can assume the graph is input as follows. The first row is a single positive integer that specifies how many vertices, n, there are. The next n rows specify the adjacency matrix for these n vertices. In particular, the first row of weights corresponds to vertex 1, the next to vertex 2, and so on up through vertex n. Within a given row, weights correspond (left to right) to vertex $1, 2, \cdots, n$ respectively, and weights are space-separated integers. The edge weight value 10000 represents ∞.

   Your program should find the shortest path between every pair of vertices and then, for each pair, print out two things:

   (a) The distance from i to j.

   (b) The *shortest path* (list of vertices) from i to j that correspond to this path.

It is a natural mapping from our discussion in class to getting the first item to work (I recommend you start by doing this part). The second item is a bit trickier—think before you code and start early (debugging can be painful). When printing paths, take care to ensure that no self-loops are printed (i.e. do not include in the output any 0-weight (i, i) edges). On a printout of our source code, briefly analyze the run-time of your path printing code (it should *not* be exponential!).

I provide two graph definition files, `fw_data1` and `fw_data2`, and corresponding correct results, `fw_data1_results` and `fw_data2_results`, are available at:

- `http://www.cs.hmc.edu/courses/2004/spring/cs140/homework/fw_data1`
- `http://www.cs.hmc.edu/courses/2004/spring/cs140/homework/fw_data1_results`
- `http://www.cs.hmc.edu/courses/2004/spring/cs140/homework/fw_data2`
- `http://www.cs.hmc.edu/courses/2004/spring/cs140/homework/fw_data2_results`

These files serve to document what your program should read as input, how it should format its answers, and provide test-cases for debugging your code.

You should turn in a printout of your source code as well as scripts that show your program's output on these two graphs (and the aforementioned run-time analysis). I recommend that you deal with input and output using standard I/O. For example, if your program was called `myFW`, run

```
myFW < fw_data1 > fw_data1_results.
```

To generate a script, you can then simply: `lpr fw_data1_out`.

You should take effort to write reasonable, comprehensible code. Badly documented or unmodular code may loose points. Extra credit may be assigned for exceptionally elegant efforts.

3. **[25 points] An Algorithmic Get-Rich-Quick Scheme!**

Your job at My-I'm-Soft was fun for awhile, but your reputation has gotten out and you've been hired back by the brokerage firm of Weil, Proffet, and Howe at a whopping salary. (We won't be specific about the actual amount, which is to say that your very happy that the company writes its payroll checks on a 64-bit computer!).

Weil, Proffet, and Howe has just entered the **arbitrage** business. Arbitrage is a money-making scheme involving anomalies in international currency exchange rates. For example, imagine that 1 U.S. dollar buys 0.8 Zambian kwachas, 1 Zambian kwacha buys 10 Mongolian tughriks, and 1 Mongolian tughrik buys 0.15 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy $0.8 \times 10 \times 0.15 = 1.2$ U.S. dollars. By capitalizing on such anomalies quickly (before they're detected and corrected by the markets), huge amounts of money can be made.

We'll assume that we're given n currencies and the exchange rate between every pair of currencies. That is, we are given currencies $c_1, \ldots, c_n$ and an $n \times n$ table R such that one unit of currency c_i buys $R[i, j]$ units of currency c_j. (The values for all the diagonal elements are one; if one were to exchange a currency with itself one gains or looses nothing. Further, negative values for $R[i, j]$ never occur.)

(a) We'll begin with the following problem. Given a list of currencies and exchange rates and a particular currency c_i—i.e. a single source—determine the maximum amount of each currency that you can obtain, beginning with 1 unit of currency c_i. *For this part of the problem assume that there exist no cycles that allow you to get arbitrarily rich via arbitrage.* Describe *two* different algorithms for doing this: one based on a modification of Floyd-Warshall and one based on a modification of Bellman-Ford. (Your algorithms can be described in English. Pseudo-code is not necessary.) In a few sentences, shed some light on why each of your algorithms works. Finally, explain the running times of each of your two algorithms.

(b) Now assume that the exchange rates are such that it may be possible to get rich via arbitrage. That is, there may exist a cycle of currency exchanges that allows you to make more of your starting currency than you had initially. Describe how you could modify each of your algorithms to determine if it's possible to profit from arbitrage with the given currencies and exchange rates. You should also prove that each of your modified algorithms is correct.

(c) Finally, describe and analyze the running time of an algorithm for finding the actual sequence of currencies to be exchanged that makes a profit using arbitrage, if such a cycle exists.