Algorithms
Computer Science 140 & Mathematics 168
Instructor: B. Thom
Fall 2003
Homework 8b
Due on 10/28/03 (Friday at 2pm)

1. [**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.

   Two graph definition files, `fw_data1` and `fw_data2`, can be downloaded from

   `bthom/public_html/courses/cs140/fall03/homework`

   on turing. The first file, `fw_data1`, is displayed in CLR on page 626. Output from the Floyd-Warshall program that I wrote is also available in this directory, e.g. files

   `fw_data1_results` and `fw_data2_results`.

   These output files serve to document how your program should format its answers and will also be useful when 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. 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 then simply `lpr fw_data1_out`. This is much better than using `emacs` or a diary program to record your scripts (which can record carriage returns in such a way that they don't print correctly).

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

   A few programming tips:

(a) You want to implement FW as a Dynamic Program (not as a recursive program). In other words, you need to convert the FW recursive formulation discussed in class into a DP setting.

(b) The k in FW was crafted so as to eliminate the need to loop over all vertices (recall this more expensive computation was required for our `SP_recur` formuation). The trick was to define a set of intermediate vertices, $\{v_1, v_2, \cdots, v_k\}$, such that only some optimal subset of these would contribute to a particular shortest path. It is important that your implementation have the following semantics: $k = 0$ corresponds to the empty set of vertices.

(c) What does one need to record to report an actual shortest path? If the option that includes k is best, to record k is insufficient because you'd still need to consider the path from i to k and then the path from k to j. The most natural way to print out an actual shortest path is to record which option you took at each step and then trace the solution recursively using this knowledge. While this is more efficient on the one hand (because, for each call, you know which option to use), each time the option that keeps k is recorded, two recursive backtrace calls are still needed, and this can result in an exponential runtime. The efficient $O(V)$ way to print out an actual shortest path is to, for each choice, record the next (or previous) value in the path. When retracing a solution this way, you always want to consult values that correspond to $k = V$. When printing out your paths, either a recursive or an efficient approach will be accepted.

(d) It is possible to solve this problem using a 2-d matrix (over i, j) as opposed to a 3-d matrix (over i, j, k).

(e) If two options are equal, care should be applied when handling ties. Why? A graph with a zero cost cycle should not print out that cycle in any shortest path that could contain it (this also takes care of annoying self-loops).

2. [**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 internet. "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. 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?

3. **[25 Points] Finding a Negative Cost Cycle.** In class we showed how the Bellman-Ford Algorithm can be used to detect if a negative cost cycle exists in a directed graph. In this problem, you will play with this idea some more.

   (a) Show how to modify Floyd-Warshall so that it can detect if a negative-cost cycle exists. Sketch at a high-level why your algorithm works.

   (b) Develop an algorithm that will determine how *a* negative-cost cycle can be found (that is, the vertices and edges in one such cycle can be printed out) if one exists. Your algorithm need only find *one* such negative cost cycle. This algorithm should rely on a Floyd-Warshall or Bellman-Ford detector (and if there is a particular preference for using one versus the other please specify why that is). Describe the algorithm and analyze its running time. Your algorithm should run in time polynomial in V and E, but it's not required to be the fastest possible algorithm for this problem.