Homework 4 - Solutions

Deadline: 11:59pm, April 5, 2013
Available points: 110. Perfect score: 100.

Problem 1 (35 points): To solve this problem, we will first construct a graph using the conversion rates. The nodes, V, of this graph, represent the different commodities, while the edges E, represent possible trades, and have weights equal to the exchange rates, R_{ij}.

A. To solve this problem, we can employ Bellman-Ford on a suitable weighted, directed graph $G = (V, E)$. This graph contains one vertex for each commodity, and for each pair of commodities c_i and c_j, there are directed edges (u_i, u_j) and (u_j, u_i). To determine the weights of the edges, we start by observing that:

$$\text{maximizing } R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] \text{ implies}$$

$$\text{minimizing } \frac{1}{R[i_1, i_2]} \cdot \frac{1}{R[i_2, i_3]} \cdots \frac{1}{R[i_{k-1}, i_k]} \cdot \frac{1}{R[i_k, i_1]} \text{ which also implies}$$

$$\text{minimizing } log(\frac{1}{R[i_1, i_2]}) + log(\frac{1}{R[i_2, i_3]}) + \cdots + log(\frac{1}{R[i_{k-1}, i_k]}) + log(\frac{1}{R[i_k, i_1]})$$

Thus, if we defined the weight of an edge (u_i, u_j) so that $w(u_i, u_j) = -log(R[i, j])$, then it is possible to detect the existence of a sequence that maximizes the exchange ratio between two specific commodities by solving a shortest path problem on this graph. Given this transformation, commodities with a low conversion rate have a high edge weight, while high conversion rates will result in high negative values. Because these edges may have negative weight, we must employ Bellman-Ford rather than Dijstra's algorithm.

It takes $\Theta(|V|^2)$ to create the graph G, which has $\Theta(|V|^2)$ edges. Then it takes $O(|V||E|)$ or $O(|V|^3)$ time to run BELLMAN-FORD.

B. We will again use Bellman-Ford algorithm, this time for it's ability to detect negative weight cycles. We can add an extra vertex, v_0, with 0 weight edges to all the vertices in the graph and run Bellman-Ford on this vertex. Using the above transformation on the edge weights used in part A, a negative weight cycle will actually correspond to a cycle where the product of the exchange rates is greater than 1:

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1 \Leftrightarrow$$

$$\frac{1}{R[i_1, i_2]} \cdot \frac{1}{R[i_2, i_3]} \cdots \frac{1}{R[i_{k-1}, i_k]} \cdot \frac{1}{R[i_k, i_1]} < 1 \Leftrightarrow$$

$$log(\frac{1}{R[i_1, i_2]}) + log(\frac{1}{R[i_2, i_3]}) + \cdots + log(\frac{1}{R[i_{k-1}, i_k]}) + log(\frac{1}{R[i_k, i_1]}) < 0$$

Therefore, to report whether such an undesirable trade exists, we report the inverse of the negative weight cycle result from Bellman-Ford. The running time is no different than the original Bellman-Ford, resulting in a running time of $O(|V||E|)$ or $O(|V|^3)$.

Another way to determine whether a negative-weight cycle exists is to create G and, without adding v_0 and its incident edges, run an all-pairs shortest-paths algorithms. If the resulting shortest-path distance matrix has any negative values on the diagonal, then there is a negative-weight cycle.

Problem 2 (25 points): A relatively straight-forward approach to this problem is to build a specialized graph using the flight information, call this graph S. For each city in the original graph, G, there will exist some set of incoming flights, which will be represented by in-edges to this node, v, and let us call this set of in-edges I_v. For each of these vertices, we define a new set of vertices in S, where each of these vertices represent the city of v, but after different incoming flights. Each of these new vertices will contain exactly one in-edge, which is the incoming flight, and retain all out-edges which are feasible to use still; that is, the incoming flight arrives at least an hour before the departure time. Once this step has been performed over all the vertices, the new graph S will contain all feasible flight plans. To get the actual shortest flight information, we need to augment the weights on the edges with the expected wait time before departure as well. From this, we can run Dijkstra's Algorithm to return the shortest sequence of flights from one city to the next, though there will be slight modifications due to multiple nodes representing a single city.

The running time of this solution depends both on the construction time of the new graph, S, as well as the cost of performing Dijkstra's algorithm over this new graph. The construction of the graph will take $O(m^3)$ time, as we must simply create new destination nodes, based on the number of edges in the graph. From there, these new nodes need only have some of their out edges pruned, which will take on the order of the number of edges in the new graph, S, which is $O(m^2)$, giving the total runtime of $O(m^3)$. We must consider the cost of Dijkstra's algorithm run on S. Because the number of nodes in S is $O(m)$ and the number of edges in S is $O(m^2)$, the total runtime of Dijkstra's will be $O(m^2 + mlogm)$. This means the runtime will be dominated by the construction time, $O(m^3)$.

To see that this method is correct, we observe that all the paths through the graph represent feasible sequences of flights from city to city. Assuming that wait times were appropriately added to the out-edge weights from each of the vertices, then a path would accurately accumulate the total time to get from one city to the next. Then, the correctness falls to the correctness of Dijkstra's algorithm, which is already known.

Problem 3 (50 points):
A. (30 points) We will take a Dynamic Programming approach to this problem, very similar to the Floyd-Warshall algorithm. We will store the strengths of the preference sequences of varying length for each pair of candidates. We will use a 3 dimensional array, where the first two coordinates correspond to the candidates. This means each cell corresponds to a particular pair of candidates, A and B, and will be filled with values of $s[A, B]$. Along the third dimension will be the maximum length of the sequence of candidates. For example, the first layer of cells will contain the preference values between the pairs of candidates, where $s[A, A] = 0, \forall\ A$ (i.e., no intermediate nodes allowed). The next layer will represent sequences of length at most 3 where one intermediate node is allowed and so on.

The algorithm then proceeds as follows. Begin by initializing the first layer with the initial preference values for all candidtates, $d[A, B]$. Then, for each layer, we use the strengths computed on the previous layer. For example, to compute $t(s[Bob, Dana])$, we consider the strength of all sequences, $\{Bob, C, Dana\}$ by using the previously computed values for $\{C, Dana\}$ (for example, compare with all of $\{Katrina, Dana\}$, $\{Miranda, Dana\}$, and $\{Peter, Dana\}$). We compare this with the previously computed strength between these two, $s[Bob, Dana]$, and retain the strongest of all the paths. We stop iterating when the maximum path length has been reached.

As this is just a modified Floyd-Warshall algorithm, the running time is just $O(n^3)$. Pseudocode for the method is provided.

Algorithm 1: `Election`(d - Matrix of preference values)

```
Initialize $p[i,j] = 0\ \forall i,j$
for $i := 1$ to $n$ do
    for $j := 1$ to $n$ do
        if $i \neq j$ then
            if $d[i,j] > d[j,i]$ then
                $p[i,j] := d[i,j]$
for $i := 1$ to $n$ do
    for $j := 1$ to $n$ do
        if $i \neq j$ then
            for $k := 1$ to $n$ do
                if $i \neq k$ and $j \neq k$ then
                    $p[j,k] :=$ max$(p[j,k],$ min$(p[j,i], p[i,k]))$
return $p[i,j]$
```

B (20 points). We begin by initializing the first array of values.

1	Bob	Dana	Katrina	Miranda	Peter
Bob	0	12	28	14	15
Dana	33	0	16	18	25
Katrina	17	29	0	24	19
Miranda	31	27	21	0	23
Peter	30	20	26	22	0

At this point, there is no clear, undisputed winner, as it must be that the row of a candidate must have values all higher than the corresponding values in that candidate's column (which means that candidate is preferred over every other candidate). Because of this, we must iterate until we reach sequences of length 5, resulting in:

4	Bob	Dana	Katrina	Miranda	Peter
Bob	0	28	28	**24**	25
Dana	33	0	28	**24**	25
Katrina	29	29	0	**24**	25
Miranda	**31**	**28**	**28**	0	**25**
Peter	30	28	28	**24**	0

Using this table, we can now compute the winner of the election. Consider the values computed for Miranda. We see according to the above table that $p[Miranda, X] > P[X, Miranda]$ for every other candidate X. Clearly this implies that Miranda is the most preferred, and thus wins the election. To get the next candidate, simply repeat this check omitting all values involving Miranda, and so on. In so doing, we get the final total ordering of Miranda > Peter > Katrina > Dana > Bob.