Solutions to Problem Set 5

This problem set is due **at the beginning of** class on *Thursday, April 10, 2003*. Note that you have only **one week** to do it in.

Each problem is to be done on a separate sheet (or sheets) of paper. Mark the top of each sheet with your name, 6.046J/18.410J, the problem number, your recitation section, the date, and the names of any students with whom you collaborated.

Problem 5-1. Currencies

Professor Indyk has a lot of polish currency (Z zlotys). However, he cannot use them directly, and he wants to convert them to US dollars through a series of currency trades involving assorted currencies, so as to maximize the amount of dollars he ends up with. Help professor Indyk convert the zlotys.

(a) For this part you can assume you have n currencies and rates $r_{i,j}$ of exchange between currency i and currency j. Assume that going around any directed cycle of trades doesn't give you profit, i.e., there is no opportunity to make profit by arbitrage. Your algorithm should run in polynomial time.

Solution

Take the negative of the logs of the weights of each edge, and set $l_{i,j} = -\log r_{i,j}$. Since there is no cycle for of trades that give you profit, there is no negative cycle with respect to the $l_{i,j}$'s. Computing a path from Z to the dollars D that maximizes the dollars corresponds to computing a path from Z to D that minimizes the sum of the lenghts l's of the edges on the path. This can be done by using Bellman-Ford.

(b) **(Optional)** For this part you can assume that you have n clients and each client i is willing to convert up to u_i units of currency a_i into currency b_i at rate r_i. Assume that going around any directed cycle of trades doesn't give you profit, i.e., there is no opportunity to make profit by arbitrage. Your algorithm should run in polynomial time.

Problem 5-2. Euler tour

An **Euler tour** of a connected digraph $G = (V, E)$ is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once.

(a) Show that G has an Euler tour if and only if in-degree(v)=out-degree(v) for each vertex $v \in V$.

Solution

Given an Euler tour, we look at the vertex v. Since the tour is a cycle by traversing this tour, if we enter k times the vertex v then we have to exit k times. Since we have to use all the edges and we can't repeat the edges we use, we conclude that in-degree(v)=out-degree(v).

We prove the reverse direction constructively. We start at a vertex v which has non-zero out-degree(v). We start walking on the graph G and whenever we traverse an edge, we remove it from the graph. Since in-degree(w)=out-degree(w) for all $w \in V$, there is always an edge to exit a vertex $w \neq v$. We repeat this process until we come back to v. After this step we have removed a cycle from G. After removing this cycle we still have in-degree(w)=out-degree(w) for all $w \in V$. We repeat this process until there are no remaining edges in G. Since the original G is connected there is a way of gluing these cycles together into a single cycle, which is an Euler tour.

(b) Describe an $O(|E|)$-time algorithm to find an Euler tour of G if one exists.

Solution

We use the constructive proof of the previous part to produce an algorithm for computing an Euler tour. The algorithm follows exactly the steps above. The only part that remains to be proven is how to glue the cycles. This can be done in a greedy fashion: for each vertex v we can keep the cycles that visit that vertex. We take a vertex v on the first cycle and if there are other cycles that visit that vertex we glue them to the cycle. We continue walking the new cycle until we find another vertex with another cycle to glue and so on. It is easy to see that in the end, by the time we visit every edge exactly once we have traversed the whole Eulerian tour.

Problem 5-3. Bipartite testing

Given a graph $G = (V, E)$, test whether the graph is bipartite or not. You algorithm should run in time $O(|E|)$.

Solution

Just run a DFS or BFS and label the vertices either red or blue if the level of a vertex is even or odd. Check if any back edges link two vertices of the same color. If this is the case then we have a path of odd length and the graph is not bipartite. If this is not the case then we have a 2-colored graph which is bipartite by definition. To run the DFS or BFS takes $O(E)$ time. Note that we don't need to keep an array with the color of each vertex. Instead we can insert them in a hash table as we discover them. Another solution would be to virtually initialize an array which takes $O(1)$ but this trick is outside the scope of this class. We also need to run this algorithm for every connected component of G. Total running time still takes $O(E)$.

Problem 5-4. Critical Edge Problem for Shortest Paths

Let graph $G = (V, E)$. Let $w : E \to \Re$ be a cost function. Assume $w(e) \geq 0$ for all $e \in E$. Let s be a source node. We say that an edge e is *upwards critical* if by increasing $w(e)$ by any $\epsilon > 0$ we increase the distance from s to some vertex $v \in V$. We say that an edge e is *downwards critical* if by decreasing $w(e)$ by any $\epsilon > 0$ (but still having $w(e) \geq 0$, i.e., by definition if $w(e) = 0$ then e is not downwards critical) we decrease the distance from s to some vertex $v \in V$. Give an $O(|E| \log |V|)$-time algorithm to compute the upwards and downwards critical edges of G.

Solution

There is a subtlety; zero-length cycles might exist. You will still be able to get full credit if you assumed there are no zero-length cycles. The problem is still solvable with zero-length cycles but it's a lot more hairy, lots of cases to consider, etc. We assume we have no zero-length cycles.

We compute the distances d_i from s to each vertex i via Dijkstra's algorithm. This algorithm takes time $O(|E| \log |V|)$ if we use a heap as the priority queue.

For an edge (i, j) to be downwards critical, it must have $w((i, j)) > 0$ and it must be on a shortest path from s to j, which means $d_j = d_i + w((i, j))$.

For an edge (i, j) to be upwards critical, it must be on a shortest path from s to j, and there must not be another shortest path from s to j that does not contain (i, j). Such a path must end with (k, j), $k \neq i$ since we have no zero-length cycles. Therefore, first for each v, we count the number of edges (u, v) for which $d_v = d_u + w((u, v))$. Let $c(v)$ denote this count for vertex v. We conclude that an edge (i, j) is upwards critical iff $d_j = d_i + w((i, j))$ and $c(j) = 1$.