CS 170 Algorithms Fall 2013 Satish Rao HW 6

1. (10+5 pts.) Graph construction

Solution: We can construct a graph where the nodes are the states, and there is a directed edge from state a to state b if by one crossing we get state b from a. We name the the states using four-bit binary strings, where the four bits correspond to F, W, S, and C respectively, 0 means East, and 1 means West. For example, the state 0101 is the state where F, S are at the east bank, and W, C are on the west bank. There are $2^4 = 16$ states in total, among them 10 states are safe (i.e. nothing will be eaten).

$$\{0000, 1010, 0010, 1110, 1011, 0100, 0001, 1101, 0101, 1111\}$$

There are 10 edges in total

$$0000 \to 1010, 1010 \to 0010, 0010 \to 1110, 1110 \to 0100$$
$$0100 \to 1101, 1101 \to 0101, 0101 \to 1111, 0010 \to 1011$$
$$1011 \to 0001, 0001 \to 1101$$

(a) By examining the graph, it is easy to identify two paths from 0000 to 1111 involving 7 river crossings, and they are the shortest ones.

$$0000 \to 1010 \to 0010 \to 1110 \to 0100 \to 1101 \to 0101 \to 1111$$

$$0000 \to 1010 \to 0010 \to 1011 \to 0001 \to 1101 \to 0101 \to 1111$$

(b) From above, we know there are 2 such solutions.

2. (15 pts.) **Solution:** We will modify Dijkstra's algorithm to solve this problem. Dijkstra keeps a set A, and $dist(u)$ for all $u \notin A$. A contains nodes whose shortest path from s are already computed. For $u \notin A$, $dist(u)$ stores the shortest $s \to u$ path among the paths that start at s, and use only nodes in A before ending at u. At each iteration, Dijkstra adds the u with the smallest $dist(u)$ to A, and update the $dist(v)$ for all v where $(u, v) \in E$.

In particular, we will change the order of how the nodes are added to A. We first recognize the SCC of our graph treating roads as bi-directional edges. If two nodes are connected by roads, then they must be in the same SCC, and the additional constraint about planes ensures that no plane edge will be inside a SCC. Thus the edges across different SCCs are exactly the plane routes.

We first linearize the SCCs to get a topological order of them, and for each node u in a SCC, let $r(u)$ be the index of the SCC in the topological ordering. It is easy to see that if $r(u) = r(v)$, then any path from u to v must be all roads, and if $r(u) > r(v)$, there won't be any path from u to v.

The modification we make to Dijkstra is that on each iteration, we include the u with smallest $r(u)$, and among the nodes with the same $r(u)$, we include the one with the smallest $dist(u)$.

We will argue when we add u to A, $dist(u)$ is the shortest path distance from s to u. Consider any $s \to u$ path

we haven't considered (i.e. a path visits some node not in A before ending at u, let v be the first such node on the path). Since the algorithm adds u instead of v in this iteration, we must have either $r(v) > r(u)$ or $r(v) = r(u), dist(v) \geq dist(u)$. In the first case, there is no path from v to u (contradiction!), and in the second case, the part of the path from v to u must be all roads (thus positive), and $dist(v) \geq dist(u)$ guarantees the initial part of the path from s to v is already no shorter than the paths we have considered for $dist(u)$, thus the entire path won't be shorter. Either way, it is safe to say we have already computed the shortest path for u. Once we established this property, the correctness of the algorithm follows from the proof of Dijkstra.

The running time is the same as Dijkstra, since the preprocessing (i.e. finding SCCs, linearization) is linear time, and the rest is the same complexity as Dijkstra.

3. (10+5 pts.) Shortest path in currency trading
Solution:

a) Represent the currencies as the vertex set V of a complete directed graph G. To find the most advantageous ways to converts c_s into c_t, you need to find the path $c_{i_1}, c_{i_2}, \cdots, c_{i_k}$ maximizing the product $r_{i_1,i_2} r_{i_2,i_3} \cdot \cdots \cdot r_{i_{k-1},i_k}$. This is equivalent to minimizing the sum $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}})$. Hence, it is sufficient to find a shortest path in the graph G with weights $w_{ij} = -\log r_{ij}$. Because these weights can be negative, we apply the Bellman-Ford algorithm for shortest paths to the graph , taking s as origin.

b) Just iterate the updating procedure once more after $|E||V|$ rounds. If any distance is updated, a negative cycle is guaranteed to exist, i.e. a cycle with $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}}) < 0$, which implies $\prod_{j=1}^{k-1} r_{i_j,i_{j+1}} > 1$, as required.

4. (5+5+5+5 pts.) Cycle property and another MST algorithm
Solution:

(a) Consider an MST T which contains e. Removing e breaks the tree into two connected components say S and $V \backslash S$. Since all the vertices of the cycle cannot still be connected after removing e, at least one edge, say e' in the cycle must cross from S to $V \backslash S$. However, then replacing e by e' gives a tree T' such that $cost(T') \leq cost(T)$. Since T is an MST, T' is also an MST which does not contain e.

(b) If e is the heaviest edge in some cycle of G, then there is some MST T not containing e. However, then T is also an MST of $G - e$ and so we can simply search for an MST of $G - e$. At every step, the algorithm creates a new graph $(G - e)$ such that an MST of the new graph is also an MST of the old graph (G). Hence the output of the algorithm (when the new graph becomes a tree) is an MST of G.

(c) An undirected edge (u, v) is part of cycle iff u and v are in the same connected component of $G - e$. Since the components can be found by DFS (or BFS), this gives a linear time algorithm.

(d) The time for sorting is $O(|E| \log |E|)$ and checking for a cycle at every step takes $O(|E|)$ time. Finally, we remove $|E| - |V| + 1$ edges and hence the running time is $O(|E| \log |E| + (|E| - |V|) * |E|) = O(|E|^2)$.

5. (5+5+5+5 pts.) Update MST after changing one edge
Solution: To solve this problem, we shall use the characterization that T is an MST of G, if and only if for every cut of G, at least one least weight edge across the cut is contained in T.

The "only if" direction is easy since if the lightest edge across a cut is not in T, then we can include it and remove some edge in T that crosses the cut (there must be at least one) to get a better tree. To prove the "if" part, note that at each in Prim's algorithm, we include the lightest edge across some cut, which can be chosen from T. Since T is a possible output of Prim's algorithm, it must be an MST.

(a) Since the change only increases the cost of some other spanning trees (those including e) and the cost of T is unchanged, it is still an MST.

(b) We include e in the tree, thus creating a cycle. We then remove the heaviest edge e' in the cycle, which can be found in linear time, to get a new tree T'. We claim that T' contains a least weight edge across every cut of G and is hence an MST.

Note that since the only changed edge is e, $T \cup \{e\}$ already includes a least weight edge across every cut. We only removed e' from this. However, any cut crossed by e', must also be crossed by at least one more edge of the cycle, which must have weight less than or equal to e'. Since this edge is still present in T', it contains a least weight edge across every cut.

(c) The tree is still an MST if the weight of an edge in the tree is reduced. Hence, no changes are required.

(d) We remove e from the tree to obtain two components, and hence a cut. We then include the lightest edge across the cut to get a new tree T'. We can now "build up" T' using the cut property to show that it is an MST.

Let $X \subseteq T'$ be a set of edges that is part of some MST, and let $e_1 \in T' \backslash X$. Then, $T' \backslash \{e_1\}$ gives a cut which is not crossed by any edge of X and across which e_1 is the lightest edge. Hence, $X \cup \{e_1\}$ is also a part of some MST. Continuing this, we can grow X to $X = T'$, which must be then an MST.