2 Updating a MST

(a) $e \in E$ and $\hat{w}(e) < w(e)$:

Solution. *Do nothing.*

Proof. T's weight decreases by $w(e) - \hat{w}(e)$, and any other spanning tree's weight either stays the same or also decreases by this much, so T must still be an MST. □

Runtime Analysis. *Doing nothing takes $\mathcal{O}(1)$ time.*

(b) $e \notin E$ and $\hat{w}(e) < w(e)$:

Solution. *Add e to T. Use DFS to find the cycle that now exists in T. Remove the heaviest edge in the cycle from T.*

Proof. The heaviest edge in a cycle cannot be in the MST (because if it is in the MST, you can remove it from the MST and add some other edge to the MST, and the MST's cost will decrease), and any edge not in an MST is the heavest edge in some cycle (in particular, the cycle formed by adding it to the MST). For any edge not in T except for e, decreasing e's weight does not change that it is the heaviest edge in the cycle, so it is safe to exclude from the MST. By adding e to T and then removing the heaviest edge in the cycle in T, we remove an edge that is also not in the MST. Thus after this update, all edges outside of T cannot be in the MST, so T is the MST. □

Runtime Analysis. *This takes $\mathcal{O}(|V|)$ time since T has $|V|$ edges after adding e, so the DFS runs in $\mathcal{O}(|V|)$ time.*

(c) $e \in E$ and $\hat{w}(e) > w(e)$:

Solution. *Delete e from T. Now T has two components, A and B. Find the lightest edge with one endpoint in each of A and B, and add this edge to T.*

Proof. Every edge besides e in the MST is the lightest edge in some cut prior to changing e's weight, and increasing e's weight cannot affect this property. So all edges besides e are safe to keep in the MST. Then, whatever edge we add is also the lightest edge in the cut (A, B) and is thus also in the MST. □

Runtime Analysis. *This takes $\mathcal{O}(|V| + |E|)$ time, since it might be the case that almost all edges in the graph might have one endpoint in both A and B and thus almost all edges will be looked at.*

(d) $e \notin E$ and $\hat{w}(e) > w(e)$:

Solution. *Do nothing.*

Proof. T's weight does not increase, and any other spanning tree's weight either stays the same or increases, so T must still be the MST. □

Runtime Analysis. *Doing nothing takes $\mathcal{O}(1)$ time.*

3 Rigged Tournament

Solution. *Let $G = (V, E)$ be the complete graph on n vertices with edge weights $l(i, j) = f(i, j)$. The key observation is that a tournament schedule corresponds to a tree in this graph , and the weight of the tree is the number of points scored across all games in that tournament schedule . Therefore , let T be the Maximum Spanning Tree in G . The weight of T is the maximum number of points scored in the tournament . To extract the sequence of games , simply run DFS or BFS from i^* , and play the games by largest depth first , making the team with a lower depth always win . For the runtime , DFS / BFS takes $O(n)$ time , which means the MST algorithm is the bottleneck . This takes $\mathcal{O}(|E| \log |V|) = \mathcal{O}(n^2 \log n)$ since there are $\mathcal{O}(n^2)$ edges (the complete graph).*

4 Arbitrage

(a)

Solution. *We represent the currencies as the vertex set V of a complete directed graph G and the exchange rates as the edges E in the graph. Finding the best exchange rate from s to t corresponds to finding the path with the largest product of exchange rates. To turn this into a shortest path problem, we weigh the edges with the negative log of each exchange rate, i.e. set $w_{ij} = -\log r_{ij}$. Since edges can be negative, we use Bellman-Ford to help us find this shortest path.*

Proof. To find the most advantageous ways to converts cs into ct, you need to find the path $c_{i_1}, c_{i_2}, \dots, c_{i_k}$ maximizing the product $r_{i_1,i_2}, r_{i_2,i_3}, \dots, r_{i_{k1},i_k}$. This is equivalent to minimizing the sum $\sum_{j=1}^{k-1}\left(-\log r_{i_j,i_{j+1}}\right)$. Hence, it is sufficient to find a shortest path in the graph G with weights $w_{ij} = -\log r_{ij}$. The correctness of the entire algorithm follows from the proof of correctness of Bellman-Ford. ☐

Runtime Analysis. *Same as runtime of Bellman-Ford,* $O(|V|^3)$ *since the graph is complete.*

(b)

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

Proof. Same as the proof for the modification of Bellman-Ford to find negative edges. ☐

Runtime Analysis. *Same as Bellman-Ford,* $O(|V|^3)$.

5 Bounded Bellman-Ford

Solution. *The obvious instinct is to run the outer loop of Bellman-Ford for k iterations instead of* $|V| - 1$ *iterations. This fails because it's possible that we find a lowest-weight path from s to some vertex using* $|V| - 1$ *edges in the first iteration, depending on the order in which we iterate over edges (e.g., see Discussion 4, Problem 1b). Intuitively, to avoid this we want to use the "old" values of dist in each iteration. So, let* $dist_i(v)$ *be the lowest-weight of a path from s to t using at most i edges.* $dist_0(s) = 0$*, and all other* $dist_i(v)$ *are initialized to* ∞*. Instead of doing the update*

$$dist(v) := min(dist(v), dist(u) + w_u, v)$$

in iteration i of Bellman-Ford we do the update

$$dist_i(v) = min(dist_i(v), dist_{i-1}(u) + w_u, v)$$

We run for k iterations, then output the values $dist_k$.