Buffalo Hird
CS124
Homework 3

Problem 1:

Assuming exchange rates between $(0, \infty)$ (i.e. exchanges rates must be some non-zero fractional value or positive real number), we can define our graph s.t. the vertices are the individual currencies and edges are exchange rates. We note that values below 1.0 are poor options as they involve making a money-losing trade. It could be possible that a succesful path would contain such a transaction but we want to note this transactions as of negative utility. By setting the weight of edges as the log of their exchange rate, we create values such that all exchange rates below 1.0 are negative numbers, and all exchange rates above 1.0 are positive, with 1.0 being weighted as 0. We can in this process take the negative of each weight such that these relations are reversed such that all net gain trades give negative weight (or negative cost).

We can then use Bellman Ford to solve for the shortest path of the graph of currencies and exchange rates as vertices and edges respectively. We note that the shorter the path we find the more positive the exchange rate. However, this data is mostly meaningless for our purposes. We note that Bellman Ford detects negative cycles, which in this case means we have found an arbitrage loop from which we can produce risk-free positive gain. We can return this cycle found by the search.

Time complexity: We first modify each weight w by $-log(w)$ in time $O(|E|)$. We then use Bellman Ford to find cycles in time $O(|V||E|)$. Therefore our asymptotic time complexity is $O(|V||E|)$.

Proof of correctness: Assume for proof by contradiction we have a risk-free currency exchange opportunity in the graph but it is not a cycle. We note that Bellman Ford, by iterating through all edges in finding shortest paths, will detect all graphs in a cycle. Our exchange opportunity must be present in the graph so we can follow a currency's edges to find this opportunity. If we follow paths leading from a source currency (which we know is included in the path of an opportunity) we must either:

- i) either eventually find a path back to this currency
- ii) never return to this currency

In case ii) we never return to our currency. This means we have not found a risk-free opportunity because this implies we return to a currency with positive value.

In case i) we consider first the subcase where we return with a positive path (or value 0).

In this subcase we have lost money returning to our currency (or made no money in case the path has sum of weights 0). This means it is not a risk-free opportunity because there is no gain in exercising it.

In the case i) subccase where we have a negative path: This means we have found a path returning to our original currency that makes the user money. Clearly (given an unchanging graph), we can exercise this path multiple times. This means it is a negative cycle and will be reported by Bellman Ford.

As we have shown how all cases except the last subcase are contradictory, we have proven that returning negative cycles found by Bellman Ford will return and only return risk-free currency exchanges.

Problem 2:

We know that our set-making routine includes n make-set operations and m find and union operations. Usually in this process we could end up with very deep, linear trees depending on the union order. Using union by rank, however, we are guaranteed to union such that our max height of each set is the max the ranks of sets being unioned. As a result, we are guaranteed a tree-like structure s.t. find operations in union require a logarithmic amount of time to reverse, instead of a linear amount of time if the trees were non-branching. As a result, we do this logarithmic operation m times, where the logarithms value is dependent on the number of elements s.t. our max rank can never be bigger than log(|elements|). As a result we have a runtime of $m * log(n)$.

Proof that $max(rank) = logn$:

Base Case: If we have a sets with only 1 element each, then we have a max rank of 1.

Inductive Step: Equivalently to $max(rank) = logn$, we have $rank(n) \rightarrow 2^n$ elements at least. Therefore for $rank(n+1)$ we require $2 * 2^n$ elements at least. This make sense, because this means we combine two trees of rank n to produce this tree of rank $n+1$.

We have therefore inductively shown that 2^n elements or more are required for rank n, therefore equivalently max rank of a set is$logn$.

We now must consider how n make-set operations and m find and union operations run in $\Omega(mlog(n))$ time when using the heuristic path compression. We note that in the same as union by rank reduces a potentially linear problem to a problem logarithmic in the rank of a tree, path compression reduce the height of our sets such that we will not have linear sets but rather tree-like structures. In creating tall and linear sets using non-heuristic unions we can find nodes near the head of the tree and do nothing heuristically. However, as soon as we must do find on a lower node, we will collapse our tree as we iterate upwards such that we reduce the height of this branch to rank 1. As a result, we will be reducing the set to a shallower, logarithmically traversed structure.

Problem 3:

We know that a minimum spanning tree of a graph with n vertices will have $n-1$ edges. We begin by removing all edges with weights greater than or equal to the weight of our chosen edge e. We can proceed through this process

efficiently by using a DFS, where we delete this edges as we proceed through our search.

We then run a modified DFS on this new graph, keeping track of our recursive stack such that we can check for cycles. We do this by detecing back edges as a result of visited a node marked as already visited in our recursive stack. From here we can return this cycle such that we can determine if e is in this cycle. As we have removed the greater and equal edges, if e is part of one of these cycles, then e is not in an MST. If e is not in one of these cycles, then e is in fact in an MST.

Time complexity: We run a DFS to delete greater than or equal to e edges in time $O(|V| + |E|)$. We then run our new DFS likewise in time $O(|V| + |E|)$ and check discovered cycles for e. This process only adds linear time such that we have running time $O(n)$ where n is the size of the input $|V| + |E|$.

Proof of correctness: It is intuitive that our DFS's are possible and achieve what is claimed. It is therefore imperative to show how this process proves e's precense in an MST.

In the case where e is not part of a cycle and therefore in some MST we note that this edge connects two components of the graph that are otherwise unconnected. It must therefore be the minimum edge weight between these two parts of the graph as we have removed all larger edges and we have shown that no other edge connects these components. We know then by the cut property that e is in some MST.

In the case where e is part of a cycle and therefore not in some MST we note that removing e from the graph would still leave the graph connected. Since e is the largest edge in the cycle - since we have removed all larger or equal edges such that e is the largest edge left - removing e will best reduce the length of the tree such that we have the minimum spanning edges. Since we would always prefer to remove e from this cycle, e cannot be in an MST.

Problem 4:

We create an array of size n where we can track the number of edges touched by each vertex $v_i$in array index i. We iterate through our edges, checking if our edge values for the vertices this new edge connects have array value > 0. If so, we don't add the edge to our MST as one or both of these vertices are already connected. If the array value $= 0$, we add this edge to our spanning tree and incremenet the array values of the connected vertices to 1.

We proceed to run this next linear step so long as we have more than $n - 1$ edges as we know a tree with n vertices has MSTs with $n - 1$ edges and we start with at most $n + 10$ edges. We iterate through the non-added edges and add one to our spanning tree. This inevitably creates a cycle as we now have n edges. We then use a modified DFS to find this cycle and delete the largest element from this cycle. We now have a spanning tree again and repeat this

process until we done this for all additional up to 10 edges such that we have an MST.

As in problem 3, we run our modified DFS on this graph, keeping track of our recursive stack such that we can check for cycles. We do this by detecing back edges as a result of visited a node marked as already visited in our recursive stack. From here we can return this cycle such that we can delete the largest item in the cycle.

Time complexity: We determine our original spanning tree by viewing all edges and weights such that we have $O(|V|+|E|)$. We then run our modified DFS a constant ≤ 10 times where each iteration takes $O(|V|+|V|)$ time and $O(|V|)$ time to delete the largest edge from each cycle. As each of these processes are linear in the size of the graph, we have an algorithm with runtime $O(n)$ where n is $|V|+|E|$ and $|E| \leq |V|+10$.

Proof of correctness: The exchange property shows that our adding and deletion of greatest edge in a cycle creates a spanning tree no bigger than our original spanning tree. It can, in fact, create smaller trees if we replace edges with edges of lesser weight. By repeating this process, we will either have created a final MST which is no smaller than our original tree, which implies we can make no smaller tree, or we will have created a smaller MST by deleting every heavier edge in each iteration leaving the smallest $n-1$ edges by this process.