I collaborated with Aditya Prasad. I am only auditing this class, so please grade my submission after all the other ones. Thanks!

Theorem 0.1: Best Theorem

This the best theorem.

Corollary 0.2: A Corollary of the Best Theorem

This a corollary of the best theorem.

Remark This is a remark.

Lemma 0.3: A Minor Lemma

This is a lemma.

Example 0.4: An Example. This is an example.

Problem 1

For notation, let $b(T) = \sum_{e \in T} b_e$. Recall that $b(T^*) = b(T'_e) \iff b(T^* \setminus T'_e) = b(T'_e \setminus T^*)$. Furthermore, let f be the bijection between $T^* \setminus T'_e$ and $T'_e \setminus T^*$, we have

$$\sum_{e \in T^* \setminus T'_e} b_e = \sum_{e \in T'_e \setminus T^*} b_e = \sum_{e \in T^* \setminus T'_e} b_{f(e)}$$

Let us select some $i \in T^* \setminus T'_e$. By the strong exchange property, $T_i = T^* \setminus \{i\} \cup \{f(i)\}$ is a feasible solution. Since we know that T^* is a min-cost bases in b-values, $b(T^*) \leq b(T_i) \iff b_i \leq b_{f(i)}$. Note that our construction did not depend on the i that we picked, i.e. $b_i \leq b_{f(i)}$ for all $i \in T^* \setminus T'_e$.
Finally, combining this with the equation above $\sum_{e \in T^* \setminus T'_e} b_e = \sum_{e \in T^* \setminus T'_e} b_{f(e)}$, we know that $b_i = b_{f(i)}$ for all $i \in T^* \setminus T'_e$. In fact, this extends to all $i \in T^*$ as we can define $f(i) = i$ for all $i \in T^* \cap T'_e$.
Now, let us examine $p_e = c_e + c(T^*_e) - c(T^*)$ for some $e \in T^*$. Let $T_e = T^* \setminus \{e\} \cup \{f(e)\}$ for $e \in T^* \setminus T^*_e$. Note that, since T^*_e is a min-cost feasible set that does not include e, it must be that $c(T^*_e) \leq c(T_e)$. Further, we have the following inequality

$$\begin{aligned} p_e &\leq c_e + c(T_e) - c(T^*) \\ &\leq c_e + \sum_{i \in T^* \setminus \{e\} \cup \{f(e)\}} c_i - \sum_{i \in T^*} c_i \\ &\leq c_e + c_{f(e)} - c_e \\ &\leq c_{f(e)} \end{aligned}$$

so the value p_e is upper-bounded by $c_{f(e)}$. We can likewise extend this arugment to all $e \in T^*$ by defining $f(e) = e$ for $e \in T^*$. By definition, $b_{f(e)} \geq c_{f(e)}$. Thus, we have the following inequality for all $e \in T^*$

$$b_e = b_{f(e)} \geq c_{f(e)}$$

Therefore,

$$\sum_{e \in T^*} b_e \geq \sum_{e \in T^*} c_{f(e)} \geq \sum_{e \in T^*} p_e$$

Problem 2

(a) Assume that there exists $e' = (u', v') \in E$ s.t. $w'_{e'} = w_{e'} + b_{u'} - b_{v'} < 0$. This implies that

$$w_{e'} + b_{u'} < b_{v'}$$

Contradiciton because $b_{v'}$ is the shortest-path distance from s to v' and the above inequality implies that the shortest-path to from s to u' and the edge e' is a shorter path to v'.

(b) For some pair of nodes (v_0, v_k) in the original graph, the shortest-path distance d_{v_0,v_k} with respect to the new weights w'_e is given by

$$d_{v_0,v_k} = w'_{v_0,v_1} + w'_{v_1,v_2} + \cdots + w'_{v_{k-1},v_k}$$

where the shortest path from v_0 to v_k is given by $(v_0, v_1, \ldots, v_k)$. Let us expand the above sum

$$d_{v_0,v_k} = w_{v_0,v_1} + b_{v_0} - b_{v_1} + w_{v_1,v_2} + b_{v_1} - b_{v_2} + \cdots + w_{v_{k-1},v_k} + b_{v_{k-1}} - b_{v_k}$$

Note that we have a telescoping sum, which can be simplified as follows.

$$d_{v_0,v_k} = w_{v_0,v_1} + w_{v_1,v_2} + \cdots + w_{v_{k-1},v_k} + b_{v_0} - b_{v_k}$$

Subtracting $(b_{v_0} - b_{v_k})$ from the above equation we get the shorest-path distance with respect to the original edge weights.

Problem 3

(a) Assume that the set of chosen edge e_v forms a cycle C of length k, i.e. there are k nodes and k edges on the cycle. Let $e^* = (a, b)$ be the cheapest edge in C. Note that $e_a = e_b = e^*$ because e^* is the cheapest edge on the cycle. Since there are one more node than edges, by the pigeonhole principle, two edge must be picked by the same node. Contradiction — one node can only pick one edge.

(b)

The algorithm produces an acyclic graph. Assume, for contradiction, that the algorithm produces an acyclic graph at iteration i. Let E_i be the set of all edges selected by the algorithm up until iteration i, and let S be the set of edges selected on iteration i. Without loss of generality, we can assume that E_{i-1} is acyclic. Notice that E_{i-1} is also defining a set of connected components (contracted nodes). $E_i = E_{i-1} \cup S$. Since E_{i-1} is acyclic, all the connected components are acyclic. Thus, the only ways that a cycle could be created is for S to create a path from connected component A to connected component B and back to A. However, this will be a cycle in S, which we have shown to be not possible. □

The algorithm produces an connected graph. The algorithm only terminate when there is only one connected component left. This implies that the output graph is connected. □

The above two proof implies that the algorithm produces a spanning tree.

The algorithm produces an MST. The edge picked by each node satisfies the cut property. Specifically, for a connected component C defined by the contracted node v, the edge e picked by C is the minimum edge across the cut $(C, \bar{C})$. Hence, the spanning tree composed of all such edges will be a MST by the cut property. □

(c) We will make use of the Union-Find data structure. We will maintain a list of connected components.

```
CC ← each node in the graph as a connected component
while $|CC| > 1$ do
```

```
        edgeList ← []
        for $v$ in CC do
            $e$ ← the cheapest incident edge of $v$.
            add $e$ to edgeList
        end for
        for $e = (u, v)$ in edgeList do
            x = Union (u,v)        ▷ We assume that Union returns the new root of the CC.
            without loss of generality, let $u$ be the new root.
            reassign the edges s.t. all edges $(v, a)$ is now $(u, a)$.
        end for
    end while
```

Note that, in this algorithm, all edges leaving a connected component will be connected to the root of the connected component.

> ***Runtime Analysis.*** It takes $O(m)$ to find the cheapest edge for all of the connect component ($n \cdot \deg v$). It takes $O(m)$ to reassign all the edges to the new root ($n \cdot \deg v$). Each iteration of the outer-most while loop reduce the number of connect component by a factor of 2, so the while loop runs $O(\log n)$ times. Hence, the algorithm takes $O(m \log n)$. Further, since $n \leq m$, we have $O(m \log m)$. □

Problem 4

insert $O(1)$ The runtime of Insert does not change; we still just add a new tree.

delete-min $O(\max \text{rank})$ The runtime of Delete-Min does not change. We still need to check all the roots after performing `clean-up`. The only question is whether this new removal scheme ensures at most $O(\log n)$ roots for our heap.

decrease-key $O(1)$ The runtime of Decrease-Key does not change. We cut out the subtree rooted at the node that we decrease. Furthermore, as the new removal rule states, if this node is the third lost child of a parent, then we remove the parent as well (apply the same rule recursively to the grand-parants).

What we are concerned with is whether the amortized runtime of `decrease-key` remain $O(\log n)$.
Following a similar analysis in the lecture, we define S_k to be the smallest number of node in a tree of rank k. In our new scheme, $S_0 = 1$ because S_0 only consists of the root itself. $S_1 = 2$ because a tree with one child has size 2. Note that in the new removal scheme,

$$S_k = 1 + S_0 + S_0 + S_0 + S_1 + S_2 + \cdots + S_{k-4} + S_{k-3}$$

because the smallest number of nodes in the tree of rank k is given in the scenario where every subtree loses two children (if they have at least two children, otherwise lose all their children). By a similar argument, we have

$$S_{k-1} = 1 + S_0 + S_0 + S_0 + S_1 + S_2 + \cdots + S_{k-5} + S_{k-4}$$

Then, $S_k - S_{k-1} = S_{k-3}$ and $S_k = S_{k-1} + S_{k-3}$. Let us assume that S_k is exponential in k, i.e. $S_k = c^k$ for some $c > 1$. We have that $c^k = c^{k-1} + c^{k-3}$ and $c^3 = c^2 + 1$. By plugging this equation into a cubic solver, we find that the only real root is $c = 1.466$. Note that it suffice to show that $S_k \geq c^k$, so we will do so via a induction proof.

Lemma 0.5

$S_k \geq c^k$ where $c = 1.466$ is the solution to $c^3 = c^2 + 1$.

Proof.

B.C.: For $k = 0$, $S_0 = 1 \geq 1$. For $k = 1$, $S_1 = 2 \geq 1.466$.

I.H.: The lemma holds for $0 \leq n \leq k$.

I.S.: Consider $k + 1$, we know that $S_{k+1} = S_k + S_{k-2}$. By our I.H., $S_k \geq c^k$ and $S_{k-2} \geq c^{k-2}$. Hence,

$$S_{k+1} \geq c^k + c^{k-2} = c^{k-2}(c^2 + 1) = c^{k-2} \cdot c^3 = c^{k+1}$$

□

Since the max rank of our fibonacci heap remains logarithmic in terms of the number of nodes, it follows that the runtime of `decrease-key` is given by $O(\log n)$