CS 170 Homework 4

Due **2/22/2022, at 10:00 pm (grace period until 11:59pm)**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write "none".

2 Updating a MST

You are given a graph $G = (V, E)$ with positive edge weights, and a minimum spanning tree $T = (V, E')$ with respect to these weights; you may assume G and T are given as adjacency lists. Now suppose the weight of a particular edge $e \in E$ is modified from $w(e)$ to a new value $\hat{w}(e)$. You wish to quickly update the minimum spanning tree T to reflect this change, without recomputing the entire tree from scratch.

There are four cases. In each, give a description of an algorithm for updating T, a proof of correctness, and a runtime analysis for the algorithm. Note that for some of the cases these may be quite brief. For simplicity, you may assume that no two edges have the same weight (this applies to both w and $\hat{w}$).

(a) $e \notin E'$ and $\hat{w}(e) > w(e)$

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

(c) $e \in E'$ and $\hat{w}(e) < w(e)$

(d) $e \in E'$ and $\hat{w}(e) > w(e)$

Solution:

(a) **Main Idea:** Do nothing.

Correctness: 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: Doing nothing takes $O(1)$ time.

(b) **Main Idea:** Add e to T. Use DFS to find the cycle that now exists in T. Remove the heaviest edge in the cycle from T.

Correctness: 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: This takes $O(|V|)$ time since T has $|V|$ edges after adding e, so the DFS runs in $O(|V|)$ time.

(c) **Main Idea:** Do nothing.

Correctness: 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: Doing nothing takes $O(1)$ time.

(d) **Main Idea:** 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.

Correctness: 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: This takes $O(|V| + |E|)$ 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.

3 Twenty Questions

Your friend challenges you to a variant of the guessing game 20 questions. First, they pick some word $(w_1, w_2, ..., w_n)$ according to a known probability distribution $(p_1, p_2, ..., p_n)$, i.e. word w_i is chosen with probability p_i. Then, you ask yes/no questions until you are certain which word has been chosen. You can ask any yes/no question, meaning you can eliminate any subset S of the possible words with the question "Is the word in S?".

Define the cost of a guessing strategy as the expected number of queries it requires to determine the chosen word, and let an optimal strategy be one which minimizes cost. Design an $O(n \log n)$ algorithm to determine the cost of the optimal strategy.

Give a 3-part solution.

Note: *We are only considering deterministic guessing strategies in this question. Including randomized strategies doesn't change the answer, but it makes the proof of correctness more difficult.*

Solution:

This solution is inspired by the observation that in binary coding, each bit of a codeword we read further narrows the possible symbols being encoded, just like a question in the game above. This correspondence is made rigorous in the proof of correctness.

Main idea: Create a Huffman tree on $(w_{1\cdots n})$ with weights $(p_{1\cdots n})$ and return the expected length of a codeword under the corresponding encoding.

Proof of correctness: Note that any guessing strategy gives a prefix-free binary encoding of the words $(w_{1\cdots n})$, where each word w_i is encoded by sequences of yes/no answers which would lead you to conclude that w_i was chosen. This encoding is prefix-free because the game only ends when all words except one have been eliminated.

Additionally, any prefix-free encoding of the words can be made into a guessing strategy as follows. Let $x_n \in \{0, 1\}^n$ represent the sequence of yes/no answers received on the first n

questions, with 1 corresponding to yes and 0 corresponding to no. Then asking at step $n+1$ whether the word can be encoded by a string with the prefix $x_n \circ 1$ (i.e. the answers so far followed by a yes) will result in the final sequence of answers being a valid encoding of the chosen word.

In this correspondence, the expected code length equals the cost of a guessing strategy. Therefore finding an optimal strategy is equivalent to finding a prefix-free encoding of the words with minimum expected codelength, which is exactly what Huffman coding does. To get the final answer, we calculate the expected codelength of the optimal strategy, which can be done by DFS from the root of the Huffman tree.

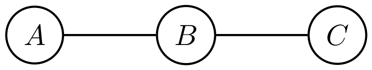
Runtime: A Huffman tree can be constructed in $n \log(n)$ time (this is dominated by the time to sort the probabilities). The average codelength (and thus cost of the associated strategy) can be calculated in $O(n)$ time by DFS. Therefore the total runtime is $O(n \log(n))$.

4 Graph Game

Given an undirected, unweighted graph G, with each node v having a value $\ell(v) \geq 0$, consider the following game.

1. All nodes are initially *unmarked* and your score is 0.
2. Choose an unmarked node u. Let $M(u)$ be the *marked* neighbours of u. Add $\sum_{v \in M(u)} \ell(v)$ to your score. Then mark u.
3. Repeat the last step for as many turns as you like, or until all the nodes are marked.

For instance, suppose we had the graph:



with $\ell(A) = 3$, $\ell(B) = 2$, $\ell(C) = 3$. Then, an optimal strategy is to mark A then C then B giving you a score of $0+0+6$. We can check that no other order will give us a better score.

(a) Is the following statement true or false? **For any graph, an optimal strategy which marks all the vertices always exists.** Briefly justify your answer.

(b) Give a greedy algorithm to find the order which gives the best score. **Please give a 3-part solution for this part.**

(c) Now suppose that $\ell(v)$ can be negative. Give an example where your algorithm fails.

(d) Your friend suggests the following modified algorithm: delete all v with $\ell(v) < 0$, then run your greedy algorithm on the resulting graph. Give an example where this algorithm fails.

Solution:

(a) Marking a node can only ever increase your score, since all values are positive.

(b) *Main idea:* Sort the nodes by value largest-first (breaking ties arbitrarily), and mark them in that order.

Proof of correctness: We show this solution is optimal by an exchange argument. Suppose that we have an ordering $v_1, \ldots, v_n$ which is not sorted by decreasing value, and let i be such that $\ell(v_i) < \ell(v_{i+1})$. Note first that if we swap the order of these nodes, only the scores that we obtain in steps i and $i+1$ could change, since in all other steps the set of marked nodes is unaffected. There are two cases.

(a) Case 1: v_i and v_{i+1} do not have an edge between them. Then the score when marking v_i is not affected by whether v_{i+1} is marked, and vice versa.

(b) Case 2: v_i and v_{i+1} do have an edge between them. Then if we swap them, v_{i+1}'s score decreases by $\ell(v_i)$, and v_i's score increases by $\ell(v_{i+1})$. Since $\ell(v_{i+1}) > \ell(v_i)$, the total score increases.

Thus, by the exchange argument, sorting the nodes will give us an optimal solution. The running time of this algorithm is $O(|V| \log |V|)$, since we just sort the vertices.

Note: this proof is missing a part. If it were the case that repeatedly choosing the first pair of out-of-order nodes and swapping them could eventually bring you back to the same order you started with, the proof would fail – it could be that there is some other solution which is even better than the sorted one, but which simply has no path to the sorted order through these swap operations. The fact that this is not the case is provable, but for this question, it is fine to assume it without proof.

Alternate Proof: An alternative proof, that does not involve an exchange argument (and so avoids the gap in the argument from above) is sketched as follows. Consider each edge (u, v) in the graph. If u is marked before v, then when v is marked, we will receive score $\ell(u)$ due to this edge. Otherwise, when u is marked, we will receive the score $\ell(v)$ due to this edge. So, looking at all the edges in the graph, we can upper-bound the total score as

$$\sum_{(u,v)\in E} \max(\ell(u), \ell(v)).$$

Now, consider the algorithm that marks nodes in descending order of value. It is clear that, for each edge, the node with a larger value is marked first, so the edge will contribute the larger value of its two endpoints. If the endpoints of the edge have equal value, then it doesn't matter how we do tie-breaking, since the edge will contribute its maximum possible value to the score in either case. So ultimately our algorithm achieves the aforementioned upper-bound on the score, so it is optimal.

Runtime: Sorting takes $O(|V| \log |V|)$ time.

(c) We can take the example graph and set $\ell(A) = 1$, $\ell(B) = -1$, $\ell(C) = -2$. Then the greedy algorithm gives A, B, C with value 0. The optimum is A, B with value 1.

(d) Again we take the example graph and set $\ell(A) = \ell(C) = 3$, $\ell(B) = -1$. The modified algorithm gives A, C which has value 0. The optimum is A, C, B with value 6.