Jinghao Yan (jinghao)

Preda, Wed. 9-10AM

HW 7

Problem 1

Finding the maximum spanning tree of edge weights $l_1, l_2, ..., l_{|E|}$ is the equivalent of finding the minimum spanning tree of edge weights that are negative of their corresponding weights $(-l_1, -l_2, ..., -l_{|E|})$. Because Prim's algorithm is guaranteed to work even with negative edges (unlike Dijkstra's)–a fact that Professor Wagner allows us to assume without proving–we just need to call Prim's algorithm with all the lengths inverted, and the resulting edges (which would be the minimum spanning tree of the graph whose edge weights are flipped) would be those that comprise the maximum spanning tree. Since finding the negative of all $|E|$ lengths is $O(|E|)$, the entire running time stays the same at $O(|E|lg|V|)$.

Problem 2

Find the maximum spanning tree in the same way as in problem 1. We can modify the cut lemma as follows to prove that Prim's algorithm with negated edge weights still allows us to get the right answer: Not only is $X \cup \{e\}$ part of a MST, e is also the edge that connects two cuts in a way as to maximize the bandwidth. Here is why:

Suppose there is another edge f between the cuts V and S-V. The capacity of the MST cannot increase if instead of $X \cup \{e\}$ we use $X \cup \{f\}$ because all other edges in the MST besides those connecting V and S-V are the same.

Running Time: Since we just use Prim's algorithm with negated edge weights, the running time of this is exactly the same as in problem 1: $O(|E|lg|V|)$.

Problem 3

General Idea: Basically we use Kruskal's algorithm to group the most similar people first. After adding $n-k$ edges, we have k separate MSTs with separation equal to the length of the $n-k+1$st edge that would be added if we ran Kruskal's algorithm to the end. Since we connect the most similar people together first, we are guaranteed a list of edges that allows us to organize the n nodes into k clusters so as to maximize the separation.

Algorithm: We will modify Kruskal's algorithm to take an additional argument k, and stop after we have created the k MSTs we need, which is after the insertion of the $n-k$th edge. To do that, we insert an extra line at the end of the "for all edges" block to return if $|X| \geq n-k$ (if the number of edges we have inserted is greater than or equal to the number of edges we must insert, then we are done!).

Proof of Correctness: It is enough to prove that after the insertion of the ith edge, there will be $n-i$ clusters such that the separation between them is maximized, because if we set $i = n-k$, we are done. (Let this be proposition P)

When $i = 0$, there are n clusters and the separation is maximized (because there are no alternatives to a graph of no edges). Suppose we have inserted $i = j < n-k$ edges into the graph, and the assertion (that those $n-j$ clusters have maximum separation) holds. Then, upon inserting the $j+1$th edge, the separation for the new $n-j-1$ clusters is still maximized, because if we had picked any edge other than the edge with the minimum weight that doesn't create a cycle, the separation would have been that edge! Put another way, let e be the $j+1$th edge (the first one we haven't considered). Since it has the minimum weight, if we add any other edge but e, the separation would be e, which is weakly less than the separation had we chosen another edge$\neq e$. Therefore, we see from a simple induction that for $i = 0, 1, ..., n-k$ proposition $P(i)$ is true. Therefore, $P(n-k)$ is true.

Running Time: $O(|E|log|V|)$ still (the running time for the standard Kruskal's algorithm) because although we only need to add $n-k$ edges, we still need to sort $|E|$ edge weights, and we don't know which edges to disregard until the weights have been sorted. This is the tightest guaranteed upper bound. In fact, with path compression version for the union-find procedures, the minor time savings we get for not having to insert more edges is insignificant compared to the time needed to sort the edges by weight. But since we make no other modifications to Kruskal's algorithm, the running time doesn't change elsewhere.

Problem 4

General Idea: Create a DAG for which (1) each path from the root to a vertex V (that is k steps from the root) represents the portion of the message up to the k^{th} bit; (2) each vertex V (that is k steps from the root) contains the states (s_{k-1}, m_k) associated with the $\tilde{c}_k$; and (3) each edge from (s_{k-1}, m_k) to (s_k, m_{k+1}) represents the different ways of going from one realization of the k^{th} vertex to the $k+1^{th}$ vertex and has weight $-log(P(\tilde{c}_k|m))$, where $P(\tilde{c}_i|m) = \begin{cases} 0.97 & f(s_{i-1}, m_i) = \tilde{c}_i \\ .01 & otherwise \end{cases}$.

Hence, to find the path of maximum probability is to find the path with $m_1, m_2, ..., m_n$ such that $P(\tilde{c}_1|m)P(\tilde{c}_2|m)...P(\tilde{c}_n|m)$ is maximized. To maximize the product is to minimize the negative log, hence each edge has the weight attached as previously described. Therefore, we just need an algorithm that builds the DAG and we can use the standard algorithm for finding the shortest path between two nodes in a DAG to find the shortest path between the source and the sink, keeping track of the m_is along the way (not including the source and the sink which don't have any). The sequence of m_is along the optimal path is the most likely original code.

Algorithm: Building the DAG:

```
Build(State[], $\tilde{c}$):
    Let G = DAG containing just a root element
    for i = 1 ... n-1:
        foreach ith state $v_i$:
            Let $(s_{i-1}, m_i) = State[v_i]$
            Add edge $E_1 = (v, Helper(i, (s_{i-1} + m_i\,(modulo 4), 1))$
            Set l($E_1$) = $-log(P(\tilde{c}_i|(m_1, m_2, ..., m_{i-1}, 1)))$
            Add edge $E_2 = (v, Helper(i, (s_{i-1} + m_i\,(modulo 4), 0))$
            Set l($E_2$) = $-log(P(\tilde{c}_i|(m_1, m_2, ..., m_{i-1}, 0)))$
    Let Sink = new Node
    foreach nth state v:
        Add edge (v,Sink) whose weight is 1
    Return G

Helper(i, $(s,m)$):

    if there already exists a vertex with the same state $(s,m)$ that is i edges away f
```

Proof of Correctness:

First, that we correctly build the DAG: At the ith iteration of the loop, we look at all the possible states (the two numbers that determine the real character) and create a vertex for each state we can reach from the ith states if they don't exist (two states in the ith iteration may have edges to the same state in the i+th state), and create an edge between each state S in the ith iteration and each state that is one symbol from S. Therefore, $((u,v) \in E) \Leftrightarrow$ (v is one symbol from u), as desired.

Running Time: O(n).

Since there can be two edges max coming out of each vertex/state (based on whether the bit is 1 or 0), $|E| = \theta(|V|)$. The distance from the source (root) to the sinks is n edges because each edge represents one potential bit that m can have, and at each level at most every possible state is represented. There are 8 possible states, so $|V| \leq 8n$, which means $|V| \in O(n)$. Therefore, finding the shortest path between root and each of the 8 possible states at the very end takes $8 \times O(|E| + |V|) =$

$O(|V|) = O(n)$ time. To prove the linearity of the whole operation, which is one call to Build and one call to find the shortest path, I simply need to show that Build() is linear.

We start by noting that the instructions in the first foreach are constant with respect to n; we do 2 additions, 2 modulo, 2 logarithms and 2 calls to f(). Then we note that for each iteration of the for loop, we do the instructions in the foreach at most 8 times since there are at most 8 states at any given level. Then the for loop runs $O(n)$ times and the rest runs in constant time, so the whole algorithm Build is linear.