CS180, Winter 2018
Homework 5
Problems: 4.25 4.28 6.4 6.6 6.12

Derek Xu

February 16, 2018

1 Problem 4.25

Let us first make the question clear:
We are given a weighted graph, $G = (P, V)$, where P is a set of points and V is a set of weights between points. We define the function $d(p_i, p_j) =$"weight of v_k such that v_k connects p_i and p_j".
As defined in the question, $d(p_i, p_i) = 0, i \neq j \Rightarrow (p_i, p_j) = d(p_j, p_i) > 0$
We must find a "hierarchal metric" of P.
A hierarchal metric is defined by the following:
Given a tree T, we map leaves of T to points in P. we define the function $\tau(p_i, p_j)$ as the lowest common ancestor of the leaf nodes corresponding to p_i and p_j, such that $\tau(p_i, p_j) \leq d(p_i, p_j)$ and there does not $\exists \tau'(p_i, p_j)$ of some other tree, T', with the some definition as τ, where $\tau'(p_i, p_j) \leq \tau(p_i, p_j)$. This $\tau(p_i, p_j)$ is the hierarchal metric we are interested in.

We notice that there is a simple solution if we create the tree bottom up with the smallest edge first. In other words, our algorithm starts from the smallest weighted edge and goes up incrementally until we cover the entire graph, G. For each iteration, we connect the two points of interest, p_i and p_j with a common ancestor v, such that the height of v is equivalent to the weight of the corresponding edge. We can do this by making a bunch of nodes above p_i and p_j before we connect it to v. Note, v always exists as the previous ancestors, or edges, all have smaller heights than v. Furthermore from the two conditions given, we can assume that the graph is connected, as $d(p_i, p_j) > 0$ for all nodes in the graph. Henceforth we can connect the highest parent node of p_i with the highest parent node of p_j instead of the nodes themselves without worry. We notice here that our algorithm is basically Kruskal's Algorithm where we are constructing branches on trees connecting the top node of each subgraph, instead of edges. The following describes our algorithm more concisely:

1 Construct and initialize Union-Find data structure.
2 Sort all the edges.
3 for each edge (u, v) going from smallest to largest weighted edge:
4 find the vertex sets of u and v
5 if these two sets are not equal
6 merge the two sets
7 create a node q that connects u and v, as described above.
8 define τ based on the resulting tree.

Our program adds line 7 and 8 to Kruskal's algorithm, but if we assume that the weights of the graph is significantly smaller than the number of nodes in the graph, we can view line 7 as a constant time operation in conjunction to line 6. Trivially, line 8 does not increase time complexity. This means our total run time is equal to that of Kruskal's algorithm, which is poly-time.

2 Problem 4.28

Let us make the question clear:
Given a graph, $G = (V, E)$, with n nodes, where each edge, $e \epsilon E$ exists in either set X or Y, we must determine if there exists a spanning tree of G such that we include k edges form X and $n - k - 1$ edges in Y, and if there exists a such a tree, what that tree is.
We realize the following phenomenon: Given a tree A and a tree B, where A has k edges from X and $n - k - 1$ edges from Y; and B has the same edges as A except replacing 1 of the X edges with an edge form Y such that it has in total $k - 1$ X edges and $n - k$ Y edges; and A and B are both spanning trees of G. We can always substitute the 1 edge that B differs from A with and replace said edge with an edge in A. The following proves this is true: Consider that the differing edge in B connects nodes (u, v). By definition, A does not contain this edge. But to connect u to v, A must contain some path between the two nodes that B does not contain (otherwise they would be the same graph). Because A and B differ in only that 1 edge (u, v), it must be that there is either some other edge in the graph connecting u in A or connecting v in A or both. We let the nodes spanned by A excluding this other edge be called A'. We let the nodes spanned by B excluding the differing edge be called B'. Notice, $B' = A'$. Therefore, we can replace the differing edge with the other edge and vice versa.
We can extend this notion to any graph A'' with k X edges and the rest Y edges, and a graph B'' with $k + m$ X edges and the rest Y edges. In other words, we can replace edges one by one in A'' with edges in B'' to construct any graph with number of X edges between k and $k + m$.
Thus, all we need to do is find the spanning tree with the minimum number of X edges and the maximum number of X edges. This can easily be done by creating a new graph with weights on the edges in X equal to 1 and the weights on the edges in Y equal to $|X| + 1$. We run the minimum spanning tree algorithm on this weighted graph and return the spanning tree with the maximum number of X edges. We then set the weight on edges in X as $|Y| + 1$ and the weight on edges in Y as 1. We run the minimum spanning tree algorithm which returns a spanning tree with the maximum number of Y edges and consequently the minimum number of X edges. We let the first graph have a "X edges" and the second graph have b "X edges". if $b \leq k \leq a$ then a spanning tree exists, and we found it through edge substitution. Otherwise, the spanning tree does not exist.

The following describes our algorithm more concisely:

1 Construct X_{min} directed graph
2 Construct X_{max} directed graph
3 Run minimum spanning tree algorithm on X_{min}
4 Run minimum spanning tree algorithm on X_{max}
5 Let a and b denote the number of X edges in X_{max} and X_{min}
6 if $b \leq k \leq a$
7 return tree with k X edges using method mentioned above
8 else
9 return no solution

Note, lines 1 and 2 takes time O(n). Lines 3 and 4 each take polynomial time, as minimum spanning tree is a polynomial time algorithm. Lines 5 and 6 is in time O(1). Line 7 takes time O(n) as there are at most $n-1$ different edges between the minimum spanning trees of X_{max} and X_{min}. Lines 8 and 9 takes O(1) time. Therefore the total algorithm runs in polynomial time.

3 Problem 6.4

(a) The following serves as a counter example:
$M = 100$
$N = \{10, 10, 1\}$
$S = \{1, 1, 10\}$
The correct answer is SSS; the algorithm returns SSN

(b) The following serves as a example:
$M = 2$
$N = \{1, 11, 1, 11\}$
$S = \{11, 1, 11, 1\}$
The optimal path jumps 3 times: $NSNS$.
Note, the total cost of this path is: $1+2+1+2+1+2+1 = 10$. Note, our path is the only path with cost 1 at every location. Therefore if another path was chosen, the path will incur cost 11, which is more than the total cost of our path, making it not th optimal path. $\therefore$ in this example, optimal path jumps 3 times.

(c) Let us make the question clear:
We are given 2 sets N, S where n_i or s_i denotes the cost of the respective set on day i.
We must choose a sequence, $x_1, x_2, ..., x_n$, where x_i is either n_i or s_i, such that $\sum_{i=1}^{n} x_i + M \cdot z$ is minimized, where z is the number of times we switch from N to S in the sequence.
We can see that our choice of jumping depends on the costs of the cities in the future. In other words the choice of jumping depends on the cost of $OPT(jump)$ and $OPT(no\ jump)$, where $OPT(no\ jump)$ is the optimum cost of the future path after not jumping and $OPT(jump)$ is the optimum cost of the future path after jumping. using this, the optimum cost starting at the i^{th} city X is:
$OPT(x_i) = x_i + min(OPT(x_{i+1}), OPT(y_{i+1}))$ where Y = other city.
From here, we simply need to write a recursive solution, and then put that recursive solution into a table form (i.e. dynamic programming).
We formalize our algorithm:

1 Let OPT be a hash table
2 Let $OPT(x_i)$ be defined as above
3 Return $OPT(1)$

*recursively fills out table by computing if x_i is not in hash table.

Because we are only filling out the OPT hash table, our total run time is proportional to the number of cities, $O(n)$.

4 Problem 6.6

Let us make the question clear:
We are given a set of words $W = \{w_1, w_2, ..., w_n\}$
We are given a set of lengths for each word $C = \{c_1, c_2, ..., c_n\}$
We partition W into sets of lines $L_1, L_2, ..., L_m$. Assume c_k is the last word in a given line L, find a way to minimize the square of the difference between the sum and L below:
$\sum_L (c_i + 1) + c_k \leq L$
Note every line must satisfy the above inequality.

We notice that the main issue is the last line, otherwise we would just use a greedy algorithm of filling each line to its fullest to minimize slack. Thus the issue is determining how long this last line should be. If we use a dynamic programming approach we can assume the optimum solution for the lines previous to the current line. This would be a function of the number of words remaining from the line above. We need to satisfy the inequality for L, thus we simply set the slack equal to inf if the value (before squaring) is negative (slack defined below). Thus, we arrive at the following solution:

Given that the slack of a line using words w_i through w_j is
$S_{ij} = (L - (\sum_{k=i}^{j-1}(c_k + 1) + c_j))^2$ *accounting for negative numbers explained above, the following describes the relationship of optimum solution:
$OPT(w_1, w_2, ..., w_n) = min_i(S_{in} + OPT(\{w_1, w_2, ..., w_{i-1}\})$

From here, we simply need to write a recursive solution, and then put that recursive solution into a table form (i.e. dynamic programming). The result is as follows...

1 Compute all values of S_{ij}
2 Let OPT be a hash table
3 Let $OPT(null - set) = 0$
4 Let $OPT(W)$ be defined as above
5 Return $OPT(w_1, w_2, ..., w_n)$
*recursively fills out table by computing if W is not in hash table.
As with the previous question, the time for lines 4 through 5 is equal to O(n). Lines 2 and 3 takes O(1) time. Because there are $\binom{n}{2}$ different values for S_{ij}, line 1 takes O(n^2) time. Our algorithm is thus O(n^2) which is polynomial.

5 Problem 6.12

Let us make the question clear:
We are given a set of servers $S = S_1, S_2, S_3, ..., S_n$
We define placement cost as a set of corresponding values $c_1, c_2, ..., c_n$
We define access cost, $k - i$, as the difference in the server with the file, S_k and the server requesting the file, S_i.
We let S_n include all possible files.
We use placement cost to copy the file. We use access cost when we look for the file. We must find an algorithm that minimizes the total cost (access+placement cost) when copying a file.

We first assume that the optimum cost of copying a file for a given server i is $OPT(i)$, from here we can conclude the optimum time of copying the file for the j^{th} server, given that the file is located in the k^{th} server is:

$OPT(j) = c_j + min_k(OPT(k) + AC_{jk})$
, where AC_{jk} is the access cost of getting to i from j, where we search each of the servers between j and i which do not contain the file. As described in the problem, $AC_{jk} = 1 + 2 + ... + (k - j - 1)$.
we know that $OPT(n) = 0$, as it contains all possible files, and $OPT(n-1) = c_{n-1}$ as we know it is in $OPT(n)$.

From here, we simply need to find OPT for all values of $1 < j < n$. The finding of value OPT would take O(n) with dynamic programming (treating OPT as a hash table or array as mentioned above), but the computation of AC will take at most: 1+2+...+n, which is O(n). Because we compute AC n times, the total running time would be bounded by O(n^2). Afterwards, finding the optimum solution for any server j can be found in O(1) by accessing the OPT array. The formal code would be the same as the previous 2 dynamic programming problems: basically, defining the recursive relationship, letting OPT be an array, and filling all of the values in OPT.