CSE 101 Homework 5

Winter 2021

This homework is due on gradescope Friday February 26th at 11:59pm pacific time. Remember to justify your work even if the problem does not explicitly say so. Writing your solutions in LaTeXis recommend though not required.

Question 1 (Minimum Spanning Subgraph, 30 points)**.** *When introducing the Minimum Spanning Tree problem, we considered the problem of finding the least costly set of edges that connects a graph G. We showed that if the edge weights are all non-negative that these edges would necessarily form a tree. However, if negative edge weights are allowed, this no longer needs to be the case. Give an algorithm that given a weighted, undirected graph G, provides a subset S of edges so that any vertex in G can be reached from any other using only edges in S and so that subject to this, the total weight of S is as small as possible. For full credit, your algorithm should run in time $O(|V|\log(|V|) + |E|)$ or better.*

Solution 1.

First, we define a contraction of a graph $G = (V, E)$ relative to an edge $e = (u, v)$. The contraction of G relative to edge e is a new graph $G' = (V', E')$ where $V' = V - \{u, v\} \cup \{w\}$ and $E' = E - e$. For every $x \in V$, let x' be the corresponding vertex in V'(corresponding vertex for u, v is w). x' is incident on an edge $e' \in E'$ iff the corresponding edge e is incident on x. Continuing this way, we can generate the contraction of a graph relative to a set of edges as well.
The basic idea is to generate an contraction of the original graph relative to all the negative edges. This will result in a graph with only positive edges. In the original graph, along with the negative edges, a set of non-negative edges will connect the graph iff the corresponding set of non-negative edges in the contracted graph connects all the vertices in the contracted graph. Therefore, in the contracted graph, we must pick out the minimum weight edges to connect all the vertices, a.k.a. the Minimum Spanning Tree. For this, we can use the Prim's algorithm.

Pseudocode: Let S be the set of edges in the Minimum Spanning Subgraph.

1. Add all the negative edges in the original graph G to the set S.
2. Contract the original graph G relative to the negative edges. Let the contracted graph be G_c.
3. Use the Prim's algorithm to find the Minimum Spanning Tree of G_c.
4. Add the edges in the MST of G_c to S.
5. Return S

For the proof of correctness, note that the Minimum Spanning Subgraph must have all the negative edges. This can be proved by contradiction. Say, there exists a Minimum Spanning Subgraph, S', which does not include some negative edge. Then we can add this negative edge to S' to get a new spanning subgraph with lower weight, thus implying that the S' wasn't the Minimum Spanning Subgraph to start with. Therefore, we first include all the negative edges in the set S. In the resulting contraction graph, the MST will give the minimum weight Spanning Tree which will connect the connected components induced by the negative edges in the original graph. Thus, the algorithm is correct.

The time complexity of generating the contraction graph is $O(|E|)$ and the time complexity of running the Prim's algorithm is $O(|V|log(|V| + |E|))$(using Fibonacci heap), thereby resulting in the desired time complexity.

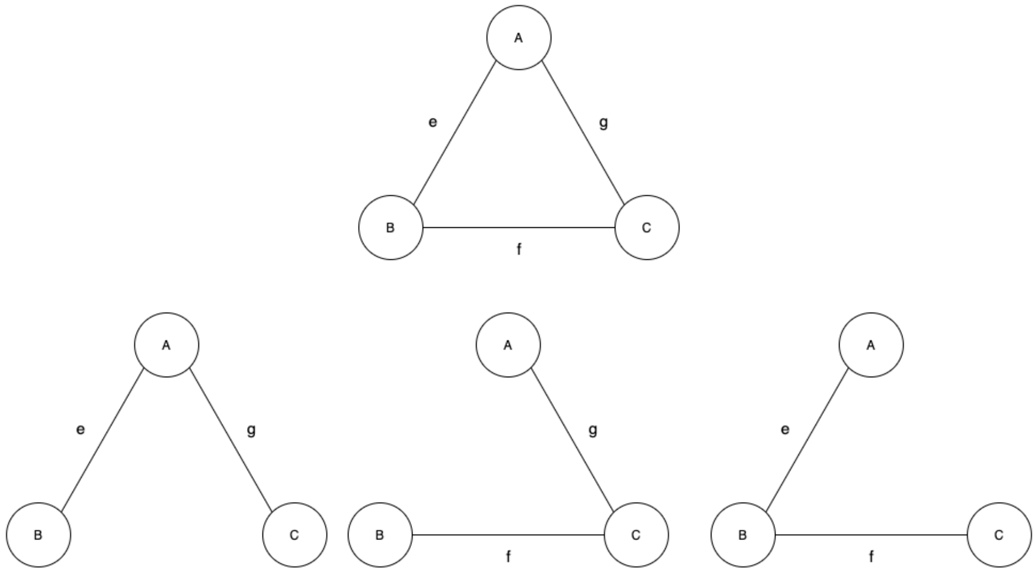

Figure 1: Example that multiple MSTs exist

Question 2 (Multiple MSTs, 40 points). *(a) Give an example of a graph that has more than one minimum spanning tree. [5 points]*

(b) Show that for any graph G with minimum spanning trees T and T' that for each weight w, T and T' contain the same number of edges of weight w. Hint: Run an exchange-like argument to slowly turn T into T' without changing the number of edges of any weight. [35 points]

Solution 2.

(a) Consider graph G shown in Figure 1. G consists of vertices A, B, C and edges e, f, g such that $e.weight = f.weight = g.weight = 1$. Then we have three MSTs T_1, T_2, T_3 where $\{e, f\} \in T_1$, $\{e, g\} \in T_2$ and $\{f, g\} \in T_3$. The sum of the weights of the edges in these three MSTs are all 2.

(b) Let T be the MST given by Kruskal's algorithm. We want to construct a sequence of MSTs $T' = T_0, T_1, ..., T_n = T$ where T_i contains at least the first i edges picked by the Kruskal's algorithm. We want to show that T_{i+1} can be obtained from T_i by swapping out an edge for another edge of the same weight for all i. For the base case, denote T_0 as the MST that contains at least the first 0 edges of T (it doesn't necessarily need to contain any edge picked by the Kruskal's algorithm). For the inductive step, let e_{i+1} be the $(i+1)^{st}$ edge picked by the Kruskal's algorithm. If it has already been included in T_i then there is nothing else to be done. Otherwise, if we add e_{i+1} to T_i to construct T_{i+1}, a cycle will be formed and we need to remove an edge from the cycle created. Here the cycle containing e_{i+1} cannot contain only edges from $e_1, e_2, ..., e_i$ because by the Kruscal's algorithm, $e_1, e_2, ..., e_{i+1}$ is part of the MST thus cannot form a cycle, so there must exist some other edges in the cycle. Suppose we remove an edge e' in the cycle that is not in $e_1, e_2, ..., e_i$. Since the Kruskal's algorithm picked e_{i+1} as the lightest edge that doesn't create a cycle with $e_1, e_2, ..., e_i$, and since the edge e' that we will remove does not form a cycle with $e_1, ..., e_i$ as well (before e_{i+1} is added), then $|e_{i+1}| \leq |e'|$. Also, if $|e_{i+1}| < |e'|$, then replacing e_{i+1} with e_i will decrease the total weight of the tree and this contradicts with the fact that T_i is a MST itself. As a result, the only possible case is $|e_{i+1}| = |e'|$. So generally, T_n and T_{n-1} should have the same number of edges of each weight and by induction, all the MSTs will have the same number of edges of each weight.

Question 3 (LCSS without Double Letters, 30 points). *Say that a string has a double letter if two consecutive letters in the string are the same. Give an algorithm that given two strings of length n, $A = a_1 a_2 \ldots a_n$ and*

$B = b_1 b_2 \ldots b_n$, *finds the longest sequence* $C = c_1 c_2 \ldots c_m$ *so that* C *is a subsequence of both* A *and* B *and has no double letters. For full credit, your algorithm should run in time* $O(n^3)$ *or better.*

Solution 3.

Let C represent the set of possible letters for this problem, and note that $C = O(n)$ (we only have to keep track of unique letters in either string). Call a subsequence with no double letters "valid." Clearly modifications to the LCSS algorithm from class are necessary here, since we need to keep track of not only the length of the longest valid common subsequence of the two prefixes $A_1 ... A_i$, $B_1 ... B_j$, but also the ending letter of that subsequence (to ensure that any future subsequences we build from are also valid). Furthermore, it is possible that for two prefixes, we have multiple valid longest common subsequences that all have maximum length but end in different letters, so we have to keep track of all of them in order to decide whether we can add the next letter or not. Thus, we need to keep track of the length of the longest subsequence for each prefix of A, prefix of B, and ending letter. We let $T[i][j][c]$ represent the length of the longest subsequence of $A_1 ... A_i$, $B_1 ... B_j$ that ends in c. Then for our base case, we simply have $T[0][j][c] = T[i][0][c] = 0$ for every i, j, c, and our recurrence for any other $T[i][j][c]$ is as follows (this follows the description of LCSS from lecture):

1. If the longest valid common subsequence of $A_1 ... A_i$, $B_1 ... B_j$ that ends in c does not use A_i, then it is actually the longest common subsequence of $A_1 ... A_{i-1}$, $B_1 ... B_j$ that ends in c. Then we have $T[i][j][c] = T[i-1][j][c]$.
2. If the longest valid common subsequence of $A_1 ... A_i$, $B_1 ... B_j$ that ends in c does not use B_j, then it is actually the longest common subsequence of $A_1 ... A_i$, $B_1 ... B_{j-1}$ that ends in c. Then we have $T[i][j][c] = T[i][j-1][c]$.
3. If the longest valid common subsequence of $A_1 ... A_i$, $B_1 ... B_j$ that ends in c uses both A_i and B_j, then we must have $A_i = B_j = c$. In this case, we have $T[i][j][c] = T[i-1][j-1][c'] + 1$ for some $c' \neq c$.

Taking the maximum over all of these for each (i, j, c) gives us the length of the best subsequence for every prefix and non-ending character combination, and at the end of our algorithm, we simply return $\max_{c \in C} T[n][n][c]$, which gives us our desired result. In terms of runtime, the first two cases of our recurrence take constant time to compute for each character, and there are $O(n)$ characters, so this sums up to $O(n)$ per (i, j) pair. The third case takes linear time to compute when $c = A[i] = B[j]$, but this only happens for one value of $c \in C$, so again, each (i, j) pair takes $O(n)$ time in total. There are $O(n^2)$ possible pairs of (i, j), so our algorithm runs in $O(n^3)$, as required.

We approach a proof of correctness by induction. Clearly the longest valid common subsequence of any prefix with an empty string is the empty string, so our base case is correct. Now, assume $T[i'][j'][c]$ has been filled out correctly for all $i' \leq i$, $j' \leq j$, $c \in C$ (except for $T[i][j][c]$). Then from our recurrence, we know that $T[i][j][c]$ is always assigned the correct value, since our recurrence relation only relies on $T[i-1][j][c], T[i][j-1][c], T[i-1][j-1][c']$, which are all assigned correctly by assumption. Thus, $T[n][n][c]$ must contain the length of the longest valid common subsequence of A, B that ends with c, and so taking the maximum over all c yields our desired result.

Alternate solution.

In our solution above, we claimed that "it is possible that for two prefixes, we have multiple valid longest common subsequences that all have maximum length but end in different letters, so we have to keep track of all of them in order to decide whether we can add the next letter or not." However, note that there is a more efficient way to store this information than to simply keep track of the LCSS for each ending character. Specifically, note that if there is only one valid LCSS with unique ending letter for prefixes $A_1 ... A_{i-1}, B_1 ... B_{j-1}$, then if $A_i = B_j$, we can use A_i in our new LCSS iff A_i is not the ending letter of the old LCSS. On the other hand, if there are multiple valid LCSS's with unique ending letters for prefixes $A_1 ... A_{i-1}, B_1 ... B_{j-1}$, then if $A_i = B_j$, we can *always* use A_i in our subsequence (at least one of the subsequences is guaranteed to not end with A_i, and we simply pick that one). Thus all we need to store at i, j is simply the ending letter of the LCSS if there is only one unique ending letter, or a filler letter (e.g. $\varnothing$) indicating that there are multiple LCSS's of $A_1 ... A_i, B_1 ... B_j$ with unique ending letters. Let $T[i][j]$ store the length of the LCSS for $A_1...A_i$, $B_1...B_j$, and let $C[i][j]$ store its ending letter (or $\varnothing$ if there are multiple unique ending letters). Then for our base case, we simply have $T[0][j] = T[i][0] = 0, C[0][j] = C[i][0] = \varnothing$ for every i, j, and our recurrence for any other $T[i][j], C[i][j]$ is as follows (explanations abbreviated for simplicity):

1. Set $T[i][j] = T[i-1][j]$ and $C[i][j] = C[i-1][j]$.
2. If $T[i][j-1] > T[i][j]$, then set $T[i][j] = T[i][j-1]$ and $C[i][j] = C[i][j-1]$. Otherwise, if $T[i][j-1] = T[i][j]$, then $A_1...A_i$ and $B_1...B_j$ could have more than one valid LCSS. Thus, if $C[i][j-1] \neq C[i][j]$, then set $C[i][j] = \varnothing$.
3. If $A_i = B_j \neq C[i-1][j-1]$, then compare $T[i-1][j-1]$ and $T[i][j]$.
   (a) If $T[i-1][j-1] + 1 < T[i][j]$, then clearly we do nothing.
   (b) If $T[i-1][j-1] + 1 = T[i][j]$, then again, $A_1...A_i$ and $B_1...B_j$ have more than one valid LCSS. Thus, if $A_i \neq C[i][j]$, then set $C[i][j] = \varnothing$.
   (c) Finally, if $T[i-1][j-1] + 1 > T[i][j]$, then we have a new unique valid LCSS, so we set $T[i][j] = T[i-1][j-1] + 1$ and $C[i][j] = A_i$.

There are $O(n^2)$ subproblems, and each subproblem takes constant time, so our algorithm is $O(n^2)$. A proof of correctness can be achieved through induction similarly to above.