CSCI 570 - HW 2

1. Minimum - Maximum

You are given a weighted graph G and two designated vertices s and t. Your goal is to find a path from s to t in which the minimum edge weight is maximized, i.e. if there are two paths with weights 10−>1−>5 and 2−>7−>3, then the second path is considered better since the minimum weight (2) is greater than the minimum weight of the first (1). Describe an efficient greedy algorithm to solve this problem and show its complexity. Prove its correctness. (Note: This problem is exactly the same problem to Pr. 8 in HW1.)

Solution: We have the below given to us,

- We have a weighted graph $W(V, E)$, source vertex s and destination vertex t

To Do: Give a greedy algorithm to find a path from $s \rightarrow t$ where the minimum edge weight is maximized.

We use Prim's algorithm below with a modified greedy step to satisfy the question's requirements.

If we look into the question we can see that the greedy step of Prim's can be modified to include those vertices having maximum edge weight and at one point we would have chosen the maximum of all minimum weights in the process of building a spanning tree T.

FIND-PATH-WITH-MAX-MIN-EDGE-WEIGHT(W, *s, t*) :

1. Start with the source vertex *s* and add it to an empty tree *T.* This would be the root of our minimum spanning tree.
2. Expand T by adding a vertex from V|T, having maximum edge weight from and having exactly one endpoint in the tree T. Basically our greedy choice will be finding the width(x) = $max_{e=(v,x),\, v \in Tree}$ (min($widthto(v)$), $width(e)$)
3. Update the distances from all vertices to T to adjacent vertices in V|T.
4. Continue to grow the tree until we reach the destination. When we reach the destination,we break out of the loop to get the first path with maximum minimal weight.

Time complexity of our algorithm:

The time complexity of the above algorithm is $O((V+E)logV)$ which is similar to Prim's. We start by building a heap of all vertices which takes $\Theta(V)$. We keep 0 as the distance for the source vertex s and ∞ for others. In a loop, we run *deleteMin()* and *decreaseKey()* to update the distance with the maximum one , V and E times respectively. Thereby, leading us to the above time complexity. Here, we assume we are using an adjacency list.

Proof Of Correctness:

We have a weighted graph, and we will prove the algorithm by *Proof by Induction* on the number of iterations. Let S(n) be the spanning tree with n < V vertices with the maximum minimum weight in it, constructed so far by our modified Prim's algorithm.

- **Base Case:** n=1, This is true, since it is a single node with no edges.
- **Inductive Hypothesis:** Assume *S(n)* is a subtree of our maximal minimal path P.
- **Inductive Synthesis:** We need to prove that *S(n+1)* is also a part of P.

We know that *S(n)* is already a subtree of our maximal minimal Tree T. Let *e* be the edge chosen by our algorithm. We need to prove that the edge that our algorithm chooses is in fact part of the same maximal minimal weight. Assume if that edge *e* was in fact a part of the maximal minimal weight path, then by Inductive hypothesis that edge is a part of the maximal minimal weight path P. If say we choose an edge that is not maximum ie say we choose the minimum weight among the two choices available, then we are essentially creating a minimum spanning tree and hence this might not necessarily give us the maximum among all minimum as in our greedy choice, Hence we choose the maximum among all minimum edge weights by using our greedy function.Hence, we can see that our greedy choice is more valid to choose the maximum minimal weight. Hence, *S(n+1)* is also a part of P.

2. Unique MST

Suppose you are given a connected graph G, with edge costs that are all distinct. Prove that G has a unique minimum spanning tree.

Solution: We have the below given to us,

$G(V, E)$ – *Connected Graph where each E has a unique cost.*

To Prove: G has a unique minimum spanning tree.

Proof: Using *Proof by Contradiction,*

Let us assume the given graph doesn't have a unique MST ie. it has say two different MST's T and T'. Then there exists an edge *e* in T' that is not present in T. Let us us now add *e* to T. This would now result in a cycle and give rise to a new tree T''. Consider the maximum edge weight in this new T'' and remove it from the tree using the Cycle property of MST's. As we know that all edges have distinct costs and hence the maximum edge cost will be one of a kind. Hence, if *e* was the most expensive edge then we could not have constructed multiple minimum spanning trees and if *e* was not the maximum edge in the first place then we could not a MST *T,* thereby contradicting our initial assumption that there are multiple spanning trees, T and T'. Hence, proving that a graph with all distinct weights has a unique spanning tree.

3. Near-Tree

Let us say that a graph G = (V,E) is a near-tree if it is connected and has at most n + 8 edges, where n = |V |. Give an algorithm with running time O(n) that takes a near-tree G with costs on its edges, and returns a minimum spanning tree of G. You may assume that all the edge costs are distinct.

Solution: We have the below given to us:

- Graph $G(V, E)$ which is a near tree ie. it is connected and has at most $n + 8$ edges, where $n = |V|$

To Do: Give a linear time algorithm to find a minimum spanning tree in G.

Algorithm to find a MST in a near tree:

FIND_MST_NEAR_TREE(*G*):

1. Start with any source node *s* and call a function BFS that that would take input this source *s* and *G* that essentially runs *BFS* on the graph starting from *s* and if we find a cycle or not(*check in the parent array if for edge(u,v), p[v]=u or p[u]=v is so there is a cycle*). We make a call to BFS running in a loop 9 times as we know it's a near tree and hence we need to check at most n+8 edges.
2. In the BFS function we essentially are checking if there is any edge that is leading to creation of a cycle and hence if so we return that edge e having consisting of vertices *(a,b)*.
3. Based on the output we get from the function BFS if we don't have any cycle then our graph is already a MST and we break from the code if not, based on the least common ancestor *c* of edge*(a,b)* we find the heaviest edge between the paths *a* - *c* and *b* - *c*.
4. We now delete the heaviest edge weight based on the cycle property and go ahead and go back in the loop to check for cycles in order to produce the MST..

The above algorithm has linear time complexity because:

- Finding the least common ancestor based on BFS takes $O(n)$ time.
- Looking for the heaviest edge takes $O(n)$ time.
- As its a near tree we run our algorithm 9 times in order to construct our MST in worst case

Total time complexity is: $O(9n) = O(n)$

4. **Subsequence**

   You are trying to find whether a sequence of characters is a subsequence of another sequence. A subsequence of a string is obtained by deleting zero or more symbols of that string. For example, RACECAR is a subsequence of QRAECCETCAURP. Give a linear algorithm that takes two sequences S′ of length m and S of length n and decides whether S′ is a subsequence of S. Prove its correctness.

Solution: We have the below given to us,

- Two string sequences S of length n and S' of length m.

To Do: Give a linear time algorithm to check if S' is a subsequence of S.

CHECK_SUBSEQUENCE(S', m, S, n):

1. Run a loop with two loop counters i and j , where they run from *0 to* n and *0 to* m respectively.
2. In the loop check traverse through both the strings either from leftmost to rightmost and if we find a n n matching character we move ahead in both our strings otherwise we only move ahead in S.
3. Finally if all characters of S′ were found in S ie. the length of m and the counter j matched, we can return "yes" it was a subsequence else "no" it was not a subsequence.

The time complexity of the above algorithm is $O(n)$ as in the worst case we need to check until the length of S to check if S' is a subsequence of S.

Proof Of Correctness:

Proof: Using Proof by Induction on the size of *S.*

- **Base Case:** For n=1, Our algorithm would perform the expected way ie. go check and compare the one character in S with S' and if it is same it would return yes it is a subsequence else no, it's not a subsequence and hence return yes/no based on the string S' and S.
- **Inductive Hypothesis:** Let us now assume that our algorithm is correct for size of S - 1 ie n-1 and label it was *S(n-1)*.

- **Inductive Synthesis:** Assuming *S(n-1)* is true, we can now see that the comparison of the two strings is working as expected for n-1 characters of S, on adding one more character we get *n* characters in total. The loop variables are changed based on the new length *n* instead of *n-1* and hence comparison will work as expected till *n-1* and further S' is checked with one more character of *S* in this case. Based on the comparison with our last character, we can output saying if S' is a subsequence of S. Hence, this means that *S(n-1)* implies *S(n)*. Thereby, by the principle of mathematical induction we can see that our algorithm is correct for *n* characters.

5. Winter in Canada

Your friends are planning an expedition to a small town deep in the Canadian north next winter break. They've researched all the travel options and have drawn up a directed graph whose nodes represent intermediate destinations and edges represent the roads between them.

In the course of this, they've also learned that extreme weather causes roads in this part of the world to become quite slow in the winter and may cause large travel delays. They've found an excellent travel Web site that can accurately predict how fast they'll be able to travel along the roads; however, the speed of travel depends on the time of year. More precisely, the Web site answers queries of the following form: given an edge e = (v,w) connecting two sites v and w, and given a proposed starting time t from location v, the site will return a value $f_e(t)$, the predicted arrival time at w. The Web site guarantees that $f_e(t) \geq t$ for all edges e and all times t (you can't travel backward in time), and that $f_e(t)$ is a monotone increasing function of t (that is, you do not arrive earlier by starting later). Other than that, the functions $f_e(t)$ may be arbitrary. For example, in areas where the travel time does not vary with the season, we would have $f_e(t) = t + l_e$, where l_e is the time needed to travel from the beginning to the end of edge e.

Your friends want to use the Web site to determine the fastest way to travel through the directed graph from their starting point to their intended destination. (You should assume that they start at time 0, and that all predictions made by the Web site are completely correct.) Give a polynomial-time algorithm to do this, where we treat a single query to the Web site (based on a specific edge e and a time t) as taking a single computational step.

Solution: We have the below given to us:

- Directed acyclic graph $G(V, E)$ with non-negative edge weights for a source *s* and destination *d*
- Website that predicts an arrival time with this function: $f_e(t) = t + l_e$
- Assumption: They start at time 0.

To Do: Polynomial time algorithm to find the fastest way to travel through the G, from starting point to intended destination.

We can use a modified Djikstra's algorithm to solve this problem as it involves graphs having non-negative edge weights. We hence keep an array pathSet *S* to store all the vertices that we encounter on that path that leads us to the fastest way to travel to *d* from *s*.

FIND_PATH_FOR_EXPEDITION(*G, s, d*):

1. We initially add the source vertex *s* to the set *S* and define a function *t* that would basically tell us the time taken to reach a vertex and hence *t(s) = 0* as we are assuming we start at time 0. We also store the path in an array - *r[]* that essentially stores the reachability of a node.
2. We loop and perform the below operations until we don't reach the destination vertex *d.* Within this loop, we select a node *b* not in set *S* and that has at least one edge from the vertices in set *S* for which the time returned by the website is minimum i.e $t'(b) = min_{e=(a,b): a \text{ in } S} f_e(t(a))$ is as small as possible, basically choosing a path based on the least estimated arrival time predicted by the website.
3. We further add *b* to our set *S* and define $t(b) = t'(b)$ to update the arrival time based on the current node reached. The *r[]* is also updated to store the path ie *r[b]=a. {Step 3 is a part of the loop defined in Step 2}*

Time complexity of the algorithm:

The time complexity of the algorithm is very similar to Djikstra's algorithm as here we are just modifying the Djikstra's greedy choice to suit the requirements of this problem. We initially build a heap of vertices that takes $\Theta(V)$. Then in the loop each time, we are calling *deleteMin()* for V nodes that gives us a time complexity of $O(V \log V)$. We also update the *t[]* in order to for it to reflect the shortest arrival time to a destination and we run *decreaseKey()* that gives us a time complexity of $O(E \log V)$. In order to output the fastest path, we traverse through *r[]* that has a complexity of $O(V)$ in the worst case. Together, we have our combined complexity as $O(V \log V + E \log V)$

6. Shortest Path

Given a directed graph G = (V,E) with nonnegative edge weights and the shortest path distances d(s,u) from a source vertex s to all other vertices in G. However, you are not given the shortest path tree. Devise a linear time algorithm, to find a shortest path from s to a given vertex t.

Solution: We have the below given to us,

- Directed Graph *G* with non-negative edge weights and shortest path distances d(s,u) where s is the source vertex to all other vertices.Assuming its given in the form of an array *d[]*.
- The edge weights that are assumed to be given along with the graph and stored in the adjacency list.

To Do: Linear time algorithm to find the shortest path from source to a given vertex t.

For this problem, we will be using a modified BFS traversal in order to find a shortest path ie we stop the further processing when we reach the destination vertex *t*.

We would be doing the below for this problem:

- For each vertex in the graph we would need a predecessor[] that basically would store the vertex that lead to the current vertex being discovered and the current shortest distance.

SHORTEST_PATH_USING_DISTANCES_FROM_SOURCE(G, *d[]*):

1. Initially mark all elements of the predecessors[] with source vertex s *(By which we are assuming we can reach a particular vertex in shortest distance through the source only)*
2. We start with *BFS* on the source vertex s, and we check if we can improve the distance from the source vertex to the descendant vertex ie. We check if the distance from origin to predecessor plus the current length of edge that we are exploring is lesser than the distance present d[] for that respective vertex to descendant vertex.
3. If we can indeed improve the current shortest distance, we update the predecessor[] in order to update the current shortest path.
4. If we cannot improve the distance then we just pick the distance present in d[] for the source-descendant pair in question and don't update the predecessor[] and continue the BFS until the destination vertex is reached.
5. In order to finally get the shortest path from source vertex *s* to *t,* we simply traverse the predecessor[] from destination to source. This would then give us the shortest path.

   Time complexity of the above algorithm:

   The algorithm produces linear time complexity because it basically involves a BFS traversal from source till we end up at the destination vertex t that has time complexity of $O(V + E)$ *ie linear in time* . And in each pass of the traversal we are just comparing the current shortest distance to the distance in shortest distance d[] which takes constant time. Further, iterating over the predecessors[] from source to destination to get the shortest path takes $O(V)$ in the worst case. Hence adding up all we get linear time complexity.

7. Worst Gas Mileage in the World

Suppose you were to drive from USC to Santa Monica along I-10. Your gas tank, when full, holds enough gas to go p miles, and you have a map that contains the information on the distances between gas stations along the route. Let $d_1 < d_2 < ... < d_n$ be the locations of all the gas stations along the route where d_i is the distance from USC to the gas station. We assume that the distance between neighboring gas stations is at most p miles. Your goal is to make as few gas stops as possible along the way. Design a greedy algorithm to determine at which gas stations you should stop and prove that your strategy yields an optimal solution. Prove its correctness.

Solution: We have the below given to us:

- When the gas tank is full it is assumed to go upto p miles.
- We have the distances from USC to every gas station d_i that comes on the way to Santa Monica. Let us assume they are stored in an array *d[]* where $d_1 < d_2 ..... < d_n$ where d_i stands for the distance from gas station G_i *to* G_{i+1}
- Map of all Gas stations on the route to Santa Monica
- Distance between neighbouring gas stations is *p* miles.
- We assume we start with a full gas tank.

To Do: Design a greedy algorithm to make the fewest gas stops along the way to Santa Monica from USC.

The greedy strategy that we follow in the below algorithm is that we go as far as possible before stopping for gas ie. in case we have enough amount of gas in our tank at the i^{th} gas station to go to $i+1^{th}$ gas station then skip filling gas at the i^{th} gas station. Or else stop and fill gas at the i^{th} gas station.

FIND_MIN_GAS_STOPS(*d*):

1. Initially set *dist = d[i] where i=1*
2. Run the below steps in a loop until we don't reach the destination, where we check if the distance to travel is lesser than or equal to p miles and if there exists another gas station G_{i+1}. This is our greedy choice. If the condition matches we update the gas station to fill petrol as G_{i+1} instead of G_i and that becomes our new source point of travel to get to Santa Monica.
3. We also update the distance covered in our *distance_travelled* variable in order to check for the next iteration if gas needs to be filled at the next station. We update the counter variable too.
4. We also update the gas stations visited in a set called *O* that has a list of all gas stations visited to fill gas.

Time Complexity of our algorithm: We iterate at most *n* times in the worst case in the outer to stop and fill gas to reach destination and within this loop we iterate *n-1* times. Hence, the total complexity of the algorithm is $O(n)$.

Proof of Correctness:

Let S be the output set containing a list of all gas stations that we stopped at by our algorithm. Let us assume there is another solution set T that gives us an optimal solution, then the total number of elements in T must be lesser than or equal to S. If solution set T includes S[1] ie the first gas station picked by our algorithm then we can assume that our greedy algorithm worked at the first step in T.

Let us now go ahead and remove the T[1] and replace it with S[1], The set T is still of the same length. We can see here that reaching Santa Monica from S[1] is at least as far as reaching Santa Monica from T[1], thereby making the replacement feasible. The problem here can be broken down to find an optimal solution from S[1]. If set T is optimal then T - S[1] should also be optimal for the subproblem. If there was a smaller solution set then that would be smaller than T, which would be impossible because we know set T is an optimal set. Hence, here we can assume that solution set S is the same as T.

Proof of Optimality:

Let $\{s_1, s_2.....s_n\}$ be the set of all gas stations that we stopped at for refueling. Let us assume $\{o_1, o_2.....o_k\}$ be the optimal solution set.

We know that we cannot get to $s_1 + 1^{th}$ gas station unless we stop, and hence any solution set must either stop at s_1 or at a gas station before . Hence, $o_1 \leq s_1$. If o_1 lesser than s_1 then we can go ahead and swap it's first refuelling stop with s_1 without changing the number of refuelling stops. Hence, our new optimal solution is $\{s_1, o_2.....o_n\}$ since before leaving s_1 we have at least the same amount of gas as we had before swapping it.

Assume that based on Inductive Hypothesis we get, $\{s_1, s_2, ...s_{c-1}, o_c, ...o_k$ is an optimal solution. From the greedy strategy taken by our algorithm, $o_c \leq s_c$. If it is lesser and not equal we swap o_c *and* s_c which would give us $s_1....s_{c,}\ o_{c+1}.....o_k$ which is in fact still a proper feasible solution. Hence, when leaving o_c we have at least as much fuel as we did before the swp and can hence still reach Santa Monica thereby proving optimality.

8. MST

You are given a minimum spanning tree T in a graph G = (V, E). Suppose we remove an edge from G creating a new graph G_1. Assuming G_1 that is still connected, devise a linear time algorithm to find a MST in G_1.

Solution: We have the below given to us:

- Minimum spanning tree T from a graph $G(V, E)$
- Connected Graph G_1 after removing edge *e* from graph G

To Do: Linear time algorithm to find MST in G_1

There could be two cases for this problem:

Case a: If we remove an edge from the Graph G and not the MST T, then we can just return the existing MST T.

Case b: If we remove an edge from the MST T, this leads to creation of connected components and below is the algorithm for the same.

FIND_MST_IN_CONNECTED(G, T, G_1):

1. After deleting the edge *e* from the MST, run BFS traversal on the two connected components that was created on deletion of an edge in the MST. This would give rise to BFS trees in the respective connected components.
2. We make use of the weights stored either in the adjacency matrix and traverse across it to find all those edges incident on the first connected component to the other one and keep all those edge weights in a heap and call heapify() on it in order to get the edge with the minimum weight.
3. We take this minimum weight edge and add it to the vertex in one component to another, which gives rise to a new MST in the new graph G_1.

Time complexity of the algorithm stated above:

- Removing an edge takes $O(1)$ time.
- Running BFS on the connected components takes $O(V + E)$ ie linear time.
- To arrange the weights in heap and heapify() again takes $\Theta(n)$ time.

The total time complexity is thereby linear ie $O(n)$

9. Recurrences

Solve the following recurrences by the master theorem.

1. $T(n) = 9T(n/3) + n + logn$

Solution: For this recurrence we have,

$a = 9,\ b = 3,\ f(n) = n + log\ n$ and as we know $c = log_b a$, we have $c = log_3 9 = 2$

We can apply **Case 1** of master's theorem since $f(n) = O(n^{log_3 9 - \varepsilon})$ as $f(n)$ is polynomially smaller than n^c for some $\varepsilon > 0$, and conclude that $T(n) = \Theta(n^2)$

2. $T(n) = 3T(n/4) + nlogn$

Solution: For this recurrence we have,

$a = 3,\ b = 4,\ f(n) = nlogn$ and $c = log_b a = log_4 3 = 0.792$

We can apply **Case 3** of master's theorem since $f(n) = \Omega(n^{c+\varepsilon})$ where we can assume $\varepsilon \approx 0.2$ and hence conclude that, $T(n) = \Theta(nlogn)$

3. $T(n) = 8T(n/2) + (n+1)^2 - 10n$

Solution: For this recurrence we have,

$a = 8,\ b = 2,\ f(n) = (n+1)^2 - 10n$ *which can be simplified to* $n^2 - 8n + 1$ and $c = log_b a = log_2 8 = 3$

We can apply **Case 1** of master's theorem as $f(n)$ is polynomially smaller than n^c for some $\varepsilon > 0$ and hence we can see that, $f(n) = O(n^{c-\varepsilon})$ and hence conclude that, $T(n) = \Theta(n^3)$

4. $T(n) = T(2n/3) + 1$

Solution: For this recurrence we have,

$a = 1,\ b = 3/2,\ f(n) = 1$ *and* $c = log_b a = log_{3/2} 1$

We can apply **Case 2** of master's theorem as n^c approximates to n^0 *which is equal to a constant ie* 1 and hence see that both $f(n)$ and n^c *are* equal concluding that $f(n) = \Theta(n^c log^k n)$ and thus the solution to the recurrence is $T(n) = \Theta(n^c log^{k+1} n)$ which is finally, $T(n) = \Theta(logn)$ where k *is assumed to be* 0.

5. $T(n) = \sqrt{7}T(n/2) + n^{\sqrt{3}}$

Solution: For this recurrence we have,

$a = \sqrt{7} = 2.645,\ b = 2,\ c = log_2 \sqrt{7} = 1.403$ *and* $f(n) = n^{\sqrt{3}} = n^{1.732}$

We can apply **Case 3** of master's theorem as $f(n) = \Omega(n^{c+\varepsilon})$ ie. n^c is polynomially smaller than $f(n)$ for any $\varepsilon > 0$, concluding that, $T(n) = \Theta(n^{\sqrt{3}})$

10. More on Recurrences

The recurrence T(n) = 7T(n/2) + n^2 describes the running time of an algorithm ALG. A competing algorithm ALG′ has a running time of T (n) = aT (n/4) + $n^2 log\, n$. What is the largest value of a such that ALG′ is asymptotically faster than ALG?

Solution:

For algorithm ALG $T(n) = 7T(n/2) + n^2$ we have,

$a = 7,\ b = 2,\ c = log_b a = log_2 7 = 2.807$ and $f(n) = n^2$

Applying master's theorem Case 1 we get,

$T(n) = \Theta(n^{log_2 7})$ which is ALG's asymptotic worst case complexity.

For algorithm ALG′ $T(n) = aT'(n/4) + n^2$ we have,

$a = a,\ b = 4,\ f(n) = n^2$ *and* $c = log_a b = log_4 a$

Applying master's theorem Case 1 we get the , asymptotic worst case complexity as

$T(n) = \Theta(n^{log_4 a})$ *if a is* > 16

For ALG′ to be asymptotically faster than A, $log_4 a < log_2 7$

Hence, the largest value of a for A′ to by asymptotically faster than A would be any value lesser than 49.

For example, the largest integer value could be 48 for which A′ is faster than A asymptotically.