CS 180 Homework 3

Name: Wang, Zheng
UID: 404855295

CS 180 Dis 1E
Time: F 10:00-11:50
TA: Orpaz Goldstein

Question 1 (Exercise 10, Page 110):

The idea is that for a BSF tree rooted at v, if the node w is located at layer L_i, then, the shortest distance between v and w must be i. Therefore, it remains to find out how many ways w can be connected v.

Now, since for all shortest path connecting w and v, it must necessarily go through some nodes in L_{i-1}. Then, without loss of generality, assume that w is directly connected with $a_1, a_2, a_3 \dots a_k \in L_{i-1}$. If for each a_j, $j = 1,2, \dots, k$, there are $N(a_j)$ shortest path from a_j to v, then there will be $\sum_{j=1}^{k} N(a_j)$ shortest path from w to v. This is because any shortest path from w to v must go through exactly one of the a's (otherwise, if you go through more than one a's, then the path length will be larger than i and therefore is not shortest. If you go through none of the a's, by the property of BFS, there is no way from w to L_{i-2} and layers above)

Define that $N(v) = 1$, we prove this algorithm works by induction. The base case says $N(b_j) = N(v) = 1$, where $b_j \in L_1$ for all j. This is true since there is only one node in L_0, namely v. Then for the inductive step, we can adapt the argument in the previous paragraph. Thus, this algorithm will work for whichever layer w is in.

Algorithm:

- Run BFS rooted at v on the graph G to obtain the BFS tree
- Find w in the tree and set I to be the layer where w is located
- Set $N(v) = 1$
- For each layer L_1 to L_{I-1} in the BFS tree
  - For each node t in this layer
    - Find all nodes t' in the previous layer that t is directly connected to
    - Set $N(t) = \sum_{\forall t'} N(t')$
  - End for
- End for
- Find all nodes u in L_{I-1} that are directly connected with w
- Set $N(w) = \sum_{\forall u} N(u)$

And the result we get for $N(w)$ is the number of shortest paths from w to v.

Complexity:

Running the BFS will take $O(m+n)$ time ($m =$ # edges; $n =$ # vertices)

Running the for loop will take $O(m)$ time since finding all of t' will take

$degree(t)$ time, and the sum of the degree of all nodes is $2m$. So, the for loop will run at $O(m)$ time.

The rest of the operation will take constant time ($O(1)$).

Therefore, overall, this algorithm runs with $O(m+n)$ time.

Question 2 (Exercise 11, Page 111):

We will start by constructing the graph representation of the triples, and then run a BFS on the directed graph.

For the construction of the graph, we are going to denote a node with the form: $C_i[t]$. Which represent the computer C_i at time t. Suppose the virus is release to C_a at time x. Denote the virus node as $C_a[x]$. Denote the target computer to be C_b.

Also, from the question, denote the time limit of our observation by y.

The Algorithm goes like this:

- Put nodes $C_i[0]$ for all $i = 1, 2, \cdots, n$, in the graph G.
- For each triple (C_j, C_k, t_l) we found in the list of triples:
  - If $C_j[t_l]$ does not exist
    - If it the first time C_j appears in the list, add a directed edge from $C_j[0]$ to $C_j[t_l]$.
    - Else add a directed edge from $C_j[t_{l-1}]$ to $C_j[t]$.
  - Do the same for C_k.
  - Add directed edges from $C_k[t_l]$ to $C_j[t_l]$ and from $C_j[t_l]$ to $C_k[t_l]$.
- End for
- For each node $C_i[t_k]$ of G:
  - Find $C_a[0]$
  - From $C_a[0]$, follow the directed edges to find the first $C_a[t^*]$ such that $t^* > x$
  - Add a directed edge from $C_a[x]$ to $C_a[t^*]$.
- End for
- Run BFS on G with root $C_a[x]$.
- Search the BFS tree to see if there is node $C_i[t_k]$with properties $i = b$ and $t_k < y$. $(*)$
  - If found, then the target computer is infected
  - If not found, then the target computer is safe.

Explaining that this algorithm work.

If a node with properties described in $(*)$ is found in the BFS tree, then, there must exist a path from the virus to the target computer. Since for each edge BFS goes through, this edge represents either a computer already infected carries the infection to the future OR a computer passes the virus to another computer, then the virus must finally reach C_b before y. Otherwise, it means there is no path virus can take to C_b before time y and thus by time y, C_b is not infected.

Algorithm Complexity:

1. Put nodes $C_i[0]$ for all $i = 1, 2, \cdots, n$, in the graph G will need $O(n)$ time
2. Go through the list of triples to build the graph will need $O(m)$ time
3. Since there will be at most $2m$ nodes (each triple will give at most 2 nodes) and at most $3m$ edges (each triple will create at maximum 3 edges). So, searching through the graph to find this $C_a[t^*]$ will take $O(m)$ times.
4. Run the BFS on G will take $O(e + v) = O(3m + 2m) = O(m)$ times
5. Similarly, search through the BFS will take $O(m)$ times.

Thus, the overall time complexity is $O(n + m)$.

Question 3 (Exercise 2, Page 189):

(a) **TRUE**.

For this problem, we will consider generating MST using Prim's algorithm.

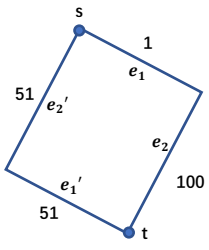
For the original graph, without loss of generality, assume that at step i of the Prim's algorithm, e_i is chosen to be included in the MST. Then e_i must be the shortest edge connecting some nodes in partition P_1 and some nodes in partition P_2. (P_1 is the partition of the nodes that have been included in the MST, P_2 is the partition of the nodes that are NOT included in MST)

Let $e_i, e_{i_1}, e_{i_2}, \dots, e_{i_n}$ be all the edges connecting P_1 and P_2 at step i of Prim's algorithm, and with out loss of generality, assume $e_i < e_{i_1} < e_{i_2} < \cdots < e_{i_n}$. As all of these edges are positive and distinct. Then, we have $e_i^2 < e_{i_1}^2 < e_{i_2}^2 < \cdots < e_{i_n}^2$. Thus, at step i, Prim's algorithm will still choose e_i in the MST.

Since this is true for all $i = 1, 2, \dots, n$. The resulting MST base on graph whose cost of each edge is squared will be the same as the MST base on the original graph.

(b) **FALSE**.

A counter example will be the graph on the right:



In this case the shortest path is from s, go through the edge e_1 with length 1 and then go through the edge e_2 with length 100 to reach t.

However, if the costs of all edges are squared, then use the path described above, the total cost is $1 + 10000 = 10001$. But if we go through the path $s \to e_1' \to e_2' \to t$. The total cost will be $51^2 * 2 = 5202$, which is smaller than the total cost of the original path.

Thus, the statement is false.

Question 4 (Exercise 4, Page 190):

We will use S to denote the full sequence with size n and the purpose is to check if a sequence S' (with size m) is a subsequence of S.

In the description of the algorithm, assume that $S[i]$ indicates the i-th element in the sequence S. Similarly, assume that $S'[i]$ indicates the i-th element in the sequence S'. Also, assume that the first element in a sequence has **index 1**.

If the algorithm returns TRUE, then it means the S' is a subsequence of S. If it returns FALSE, then it means that S' is not a subsequence of S.

The algorithm will be:

- Initialize j=1.
- While $i \leq n$
  - If $S[i]$ is the same as $S'[j]$
    - Set $j = j + 1$
  - If j is equal to $(m + 1)$
    - Return TRUE
  - Set $i = i + 1$
- End while
- Return FALSE

Proof of why this Algorithm work:

Case 1—when the algorithm returns TRUE

The algorithm returns true if and only if $j = m + 1$. Since $j = m + 1$ means so all elements in S' has a match in S. Suppose that for each element of S that matches with $S'[j]$, we will denote it $S[i_j]$.

Now since i and j will only increase when algorithm runs, we have the order $S[i_1] < S[i_2] < \cdots < S[i_m]$. By deleting everything except for $S[i_j]$'s, the sequence $S[i_1], S[i_2], \ldots, S[i_m]$ will be equivalent to the sequence S'.

Case 2—when the algorithm returns FALSE

Suppose that the algorithm returns FALSE and but S' is a subsequence of S. Then, by deleting some elements in S, the remaining event, in order, will be the same as S'. Since the remaining event will be the same as S', there are m remaining events. Call these elements in order: $S[i_1], S[i_2], S[i_3], \ldots, S[i_m]$.

Claim: the algorithm will return TRUE if there exist $S[i_1], S[i_2], \ldots, S[i_m]$ in order that matches $S'[1], S'[2], \ldots, S'[m]$.

Proof.

Base case: Suppose $m = 1$. If there exists $S[i_1] = S'[1]$, then algorithm will return TRUE.

Because before j increment if and only if when there exist some $S[i] = S'[j]$. Thus, as $S[i_1] = S'[1]$ exists, so j will be increment to $2 = 1 + 1$. Thus, the algorithm returns TRUE as claimed.

Inductive case: assume that when $m = k$, if there exist $S[i_1], S[i_2], \ldots, S[i_k]$ in order

that matches $S'[1], S'[2], \dots, S'[k]$, and the algorithm returns TRUE.
Then, the algorithm finds matches of $S[i_1], S[i_2], \dots, S[i_k]$ (in this order) corresponding to $S'[1], S'[2], \dots, S'[k]$.
We want to show: if there exist $S[i_1], S[i_2], \dots, S[i_{k+1}]$ in order that matches $S'[1], S'[2], \dots, S'[k+1]$, then the algorithm will return TRUE.
We construct our S' by adding $S'[k+1]$ to some S'_k with k elements are (in order) the same as the first k element of S'.
Then, the algorithm will still find $S[i_1], S[i_2], \dots, S[i_k]$ that matches $S'[1], S'[2], \dots, S'[k]$ in order. At the moment the algorithm finds the match of $S[i_k] = S'[k]$, $i = i_k < i_{k+1}$ and $j = k+1$.

Since i increment by 1 each loop, eventually, $i = i_{k+1}$. Since $S[i_{k+1}] = S'[k+1]$. Thus, j will increment by 1. Thus, $j = k+2 = m+1$, and the algorithm will return TRUE.

Thus, if there exist $S[i_1], S[i_2], \dots, S[i_m]$ in order that matches $S'[1], S'[2], \dots, S'[m]$, then the algorithm will return TRUE, a contradiction.

Thus, if the algorithm returns FASLE, S' is not a subsequence of the S.

Complexity:

At the worst case, the algorithm will have to go through all element in S, which takes n steps, and all elements in S', which takes m steps. Thus, overall complexity will be $O(n+m)$.

Question 5 (Exercise 7, Page 191):

The idea is that the total time needed to process all of the jobs by the super computer does not depend on the schedule G. Thus, we want to overlap the PC processing time with the super computer processing time as much as possible. The way of doing this is to ask the super computer to process the job with longest F

The algorithm will be the following:

- Sort $J_1, J_2, \dots, J_n$ by their corresponding f_i, where $i = 1,2,..,n$, in descending order
- Return the sorted list of $J_1, J_2, \dots, J_n$, which is the schedule G.

Proof that the algorithm works:

Claim: For any schedule G', if a pair of adjacent jobs J_i and J_k satisfy the condition:

1. J_i is earlier in the schedule than J_k.
2. f_i is shorter than f_k.

If we switch the order of J_i and J_k, then we will not increase the completion time.

Proof.

Let the total time of completing all preprocessing before J_i be P_{total}. Then, it will take $T_i = P_{total} + p_i + f_i$ seconds (from the very beginning) when J_i is finished. Also, it will take $T_k = P_{total} + p_i + p_k + f_k$ seconds (from the very beginning) when J_k is finished.

Now, there are two cases:

1. The job which is the last to complete is NOT J_i or J_k.
   - In this case, switching J_i and J_k will not increase the overall completion time as switching J_i and J_k does not impact the completion time of all other jobs.
2. The job with is the last to complete is one of J_i and J_k.
   - In this case, the overall completion time will be $\max\{T_i, T_k\}$.

     Now, if we switch the order of J_i and J_k in G'. i.e. process J_k before J_i. Then, it will take $T'_k = P_{total} + p_k + f_k$ seconds (from the very beginning) to complete J_k. Also, it will take $T'_i = P_{total} + p_k + p_i + f_i$ seconds (from the very beginning) to complete J_i.

     Since $f_i < f_k$, $P_{total} + p_i + p_k + f_k > P_{total} + p_k + p_i + f_i$. Also, $P_{total} + p_i + p_k + f_k > P_{total} + p_k + f_k$. Thus, $T'_i < T_k$ and $T'_k < T_k$.

     So, $\max\{T'_i, T'_k\} < \max\{T_i, T_k\}$. Thus, switching the order of J_i and J_k will not increase the overall completion time.

Thus, for any schedule $G' \neq G$, we can turn it into G by running the above switching many times without increasing the overall completion time. So, G will be the optimal schedule.

Complexity:

Since sorting the jobs will take $O(n \log n)$ time, and outputting the sorted jobs and send them to the super computer will take $O(n)$ time. Thus, the overall time complexity of this algorithm will be $O(n \log n)$.

Question 6—Part A:

I think there is no polynomial time algorithm that can solve this problem. I will use a brute force algorithm to approach this problem.

Algorithm:

- Initialize a list E to store the path with structure **vertex, edge, vertex, edge, …, vertex**.
- Initialize a list L to store the path length of the path we explored
- For each vertex v in the graph G.
  1. Put v into E

2. Choose an edge (v, w) that going out from v and mark it as explored
3. Push this edge to E, and then push w into E
4. Choose an edge (w, x) that going out from w and mark it as explored.
5. Repeat 3 and 4
    - (6) If we reach a node that either there is no path going out from it or all path going out from it is marked explored.
        - Find the length of path by following vertices in E, and push it into L.
        - Remove the last edge and last vertex from E
            - Mark the removed edge as used (different from explored) and try to find an unused & unexplored edge
            - If fails to find an unused & unexplored edge, run the chunk "Remove the last edge and last vertex from E" chunk again.
            - Else if find one such edge, run 3
                - Repeatedly running 4 and 3
                    - If reach the state of 6. Then run everything in 6's chunk.
6. If all of the edges in the graph is marked, unmark all the edges, empty E and shift to the next vertices and run everything above again.

- End for
- Find the maximum value in L

This algorithm is guaranteed to work as it will explore all possible path in G. Then the maximum we found in L will be the longest path in G.

Complexity:

Since for each iteration, we will inspect at maximum e edges for each vertex. Thus, it will take $O(e^n)$ to complete each iteration. Since there are n vertex to repeat through, we have overall time complexity of $O(n \cdot e^n)$

Question 6—Part B:

For this problem, the idea is to use a modified version of topological sort: in each iteration of the topological sort, we will remove all the sources formed. Suppose that we end up with k such set.

Algorithm:

- While there are still nodes in G
    - Go through all of the nodes in G and to find and remove all of the sources, put them in a newly created set
    - Deleting all of the outgoing edges from those sources and decrement all of the corresponding nodes' in-degree by one.
- End while
    - For G, reverse all of the edges, construct G^{rev}
    - Select arbitrary vertex v from the last set S_k and follow some reversed

edge to some vertices in set S_{k-1}.

- Repeat this process until we reach S_1, the set contains all the sources in G.
- Return the reverse of these nodes and edges

Prove that this algorithm will work:

For a point x in the set S_i, it is not in S_{i-1} since before we remove the outgoing edges of nodes in S_{i-1}, x is not a source. Thus, there exist some edge (u, x) from some $u \in S_{i-1}$ to x. Thus, when we trace backwards from S_i to S_{i-1}, there must exist some edge (x, u) in the G^{rev} such that x can go to u.

Since this is true for all i, our tracing-backwards will guarantee to find a path from a vertex in the S_k to some vertex in S_1. Note that this path will have length $k-1$.

Then we prove that this path will be the longest.

1. Since for all node v in G, whichever set it belongs to, there is no edge connecting it and any other nodes w also inside that set, otherwise, w can only become source after v is removed, so w, v cannot belong to the same set.
2. Also, for all nodes y in G, whichever set it belongs to, there is no edge connecting it and nodes x belongs to set above it. Otherwise, x will become source after y is removed, so x will not belongs to a set above the y's set.

Thus, whichever node we pick from whichever set, we can only expect to find edges going down to a set below its set. Use this idea, the maximum path length will be $k-1$, starting from S_1, and go down by only one set each time.

Complexity:

Running the topological sort will take $O(n+e)$ time.

For tracing backwards, each layer we will spend at the greatest number of out-degree tries for that vertex, and we in maximum have n sets. Thus, we totally need $O(m)$ time. Return the path takes at most $O(n)$ time as there could be at maximum n sets.

Thus, overall, the algorithm will run $O(e+n)$ time.