CS 180 Homework 3

Name: Wang, Zheng CS 180 Dis 1E

UID: 404855295 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 , if the node is located at layer , then, the shortest distance between and must be . Therefore, it remains to find out how many ways can be connected .

Now, since for all shortest path connecting and , it must necessarily go through some nodes in . Then, without loss of generality, assume that is directly connected with . If for each , there are shortest path from to , then there will be shortest path from to This is because any shortest path from to must go through exactly one of the ’s (otherwise, if you go through more than one ’s, then the path length will be larger than and therefore is not shortest. If you go through none of the ’s, by the property of BFS, there is no way from to and layers above)

Define that we prove this algorithm works by induction. The base case says where for all . This is true since there is only one node in , namely . Then for the inductive step, we can adapt the argument in the previous paragraph. Thus, this algorithm will work for whichever layer is in.

Algorithm:

* Run BFS rooted at on the graph to obtain the BFS tree
* Find in the tree and set to be the layer where is located
* Set
* For each layer to in the BFS tree
  + For each node in this layer
    - Find all nodes in the previous layer that is directly connected to
    - Set
  + End for
* End for
* Find all nodes in that are directly connected with
* Set

And the result we get for is the number of shortest paths from to .

Complexity:

Running the BFS will take time (edges; # vertices)

Running the for loop will take time since finding all of will take time, and the sum of the degree of all nodes is . So, the for loop will run at time.

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

Therefore, overall, this algorithm runs with 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: . Which represent the computer at time Suppose the virus is release to at time . Denote the virus node as . Denote the target computer to be .

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

The Algorithm goes like this:

* Put nodes for all , in the graph .
* For each triple we found in the list of triples:
* If does not exist
  + If it the first time appears in the list, add a directed edge from to .
  + Else add a directed edge from to .
* Do the same for .
* Add directed edges from to and from to
* End for
* For each node of :
  + Find
  + From , follow the directed edges to find the first such that
  + Add a directed edge from to .
* End for
* Run BFS on with root .
* Search the BFS tree to see if there is node with properties and . ()
  + 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 before . Otherwise, it means there is no path virus can take to before time and thus by time , is not infected.

Algorithm Complexity:

1. Put nodes for all , in the graph will need time
2. Go through the list of triples to build the graph will need time
3. Since there will be at most nodes (each triple will give at most 2 nodes) and at most edges (each triple will create at maximum 3 edges). So, searching through the graph to find this will take times.
4. Run the BFS on will take times
5. Similarly, search through the BFS will take times.

Thus, the overall time complexity is .

**Question 3 (Exercise 2, Page 189):**

1. **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 of the Prim’s algorithm, is chosen to be included in the MST. Then must be the shortest edge connecting some nodes in partition and some nodes in partition . ( is the partition of the nodes that have been included in the MST, is the partition of the nodes that are NOT included in MST)

Let be all the edges connecting and at step of Prim’s algorithm, and with out loss of generality, assume . As all of these edges are positive and distinct. Then, we have . Thus, at step , Prim’s algorithm will still choose in the MST.

Since this is true for all . 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.

1. **FALSE**.

A counter example will be the graph on the right:

**s**

**1**

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

**51**

**51**

**100**

However, if the costs of all edges are squared, then use the path described above, the total cost is . But if we go through the path . The total cost will be , which is smaller than the total cost of the original path.

**t**

Thus, the statement is false.

**Question 4 (Exercise 4, Page 190):**

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

In the description of the algorithm, assume that indicates the -th element in the sequence . Similarly, assume that indicates the -th element in the sequence . Also, assume that the first element in a sequence has **index 1**.

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

The algorithm will be:

* Initialize =1.
* While
  + If is the same as
    - Set
  + If is equal to
    - Return TRUE
  + Set
* 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 . Since means so all elements in has a match in . Suppose that for each element of that matches with , we will denote it .

Now since and will only increase when algorithm runs, we have the order . By deleting everything except for ’s, the sequence will be equivalent to the sequence .

**Case 2—when the algorithm returns FALSE**

Suppose that the algorithm returns FALSE and but is a subsequence of . Then, by deleting some elements in , the remaining event, in order, will be the same as . Since the remaining event will be the same as , there are remaining events. Call these elements in order: .

*Claim*: the algorithm will return TRUE if there exist in order that matches .

*Proof.*

Base case: Suppose . If there exists , then algorithm will return TRUE.

Because before increment if and only if when there exist some . Thus, as exists, so will be increment to . Thus, the algorithm returns TRUE as claimed.

Inductive case: assume that when , if there exist in order that matches , and the algorithm returns TRUE.

Then, the algorithm finds matches of (in this order) corresponding to .

We want to show: if there exist in order that matches , then the algorithm will return TRUE.

We construct our by adding to some with elements are (in order) the same as the first element of .

Then, the algorithm will still find that matches in order. At the moment the algorithm finds the match of , and .

Since increment by 1 each loop, eventually, Since . Thus, will increment by 1. Thus, , and the algorithm will return TRUE.

Thus, if there exist in order that matches , then the algorithm will return TRUE, a contradiction.

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

Complexity:

At the worst case, the algorithm will have to go through all element in , which takes steps, and all elements in , which takes steps. Thus, overall complexity will be .

**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 . 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

The algorithm will be the following:

* Sort by their corresponding , where , in descending order
* Return the sorted list of , which is the schedule .

Proof that the algorithm works:

*Claim:* For any schedule , if a pair of adjacent jobs and satisfy the condition:

1. is earlier in the schedule than .
2. is shorter than .

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

*Proof.*

Let the total time of completing all preprocessing before be . Then, it will take seconds (from the very beginning) when is finished. Also, it will take seconds (from the very beginning) when is finished.

Now, there are two cases:

1. The job which is the last to complete is NOT or .
   * In this case, switching and will not increase the overall completion time as switching and does not impact the completion time of all other jobs.
2. The job with is the last to complete is one of and .
   * In this case, the overall completion time will be .

Now, if we switch the order of and in . i.e. process before Then, it will take seconds (from the very beginning) to complete . Also, it will take seconds (from the very beginning) to complete .

Since , . Also, + . Thus, and .

So, . Thus, switching the order of and will not increase the overall completion time.

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

Complexity:

Since sorting the jobs will take time, and outputting the sorted jobs and send them to the super computer will take time. Thus, the overall time complexity of this algorithm will be .

**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 to store the path with structure **vertex, edge, vertex, edge, …, vertex**.
* Initialize a list to store the path length of the path we explored
* For each vertex in the graph .
  1. Put into
  2. Choose an edge that going out from and mark it as explored
  3. Push this edge to , and then push into
  4. Choose an edge that going out from 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 , and push it into.
       - Remove the last edge and last vertex from
         * 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 ” 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.

* 1. If all of the edges in the graph is marked, unmark all the edges, empty and shift to the next vertices and run everything above again.
* End for
* Find the maximum value in

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

Complexity:

Since for each iteration, we will inspect at maximum edges for each vertex. Thus, it will take to complete each iteration. Since there are vertex to repeat through, we have overall time complexity of

**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 such set.

Algorithm:

* While there are still nodes in
  + Go through all of the nodes in 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 , reverse all of the edges, construct
  + Select arbitrary vertex from the last set and follow some reversed edge to some vertices in set .
  + Repeat this process until we reach , the set contains all the sources in .
  + Return the reverse of these nodes and edges

Prove that this algorithm will work:

For a point in the set , it is not in since before we remove the outgoing edges of nodes in , is not a source. Thus, there exist some edge from some to . Thus, when we trace backwards from to , there must exist some edge in the such that can go to .

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

Then we prove that this path will be the longest.

1. Since for all node in , whichever set it belongs to, there is no edge connecting it and any other nodes also inside that set, otherwise, can only become source after is removed, so cannot belong to the same set.
2. Also, for all nodes in , whichever set it belongs to, there is no edge connecting it and nodes belongs to set above it. Otherwise, will become source after is removed, so will not belongs to a set above the ’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 , starting from , and go down by only one set each time.

Complexity:

Running the topological sort will take 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 sets. Thus, we totally need time. Return the path takes at most time as there could be at maximum sets.

Thus, overall, the algorithm will run time.