CS 180 - Algorithms and Complexity
Homework 3

Jayant Mehra
Discussion 1D

October 25, 2018

Solution 1. The following is the solution to exercise 10 on page 110 of the textbook.

```
1.  totalSPs(graph G, node v, node w)
2.      n ← total vertices of G
3.      discovered[n]
4.      Queue Q
5.      Q.enqueue(v)
6.      discovered[v] ← True
7.      levelSize ← 1
8.
9.      while (Q ≠ ∅)
10.         count ← 0
11.         while (levelSize ≠ 0)
12.             node ← Q.dequeue()
13.             for neighbor u in all neighbors of node
14.                 if u = w
15.                     count ← count + 1
16.                     discovered[u] ← True
17.                 else
18.                     if discovered[u] = False
19.                         discovered[u] ← True
20.                         Q.enqueue(u)
21.
22.             levelSize ← levelSize - 1
23.
24.         if count ≠ 0
25.             return count
26.
27.     return 0
```

Algorithm Explanation We implement a modified breadth first search to find the total number of shortest paths between v and w in graph G. We start the BFS at v and say we

hit u for the first time when we were exploring the neighbors of nodes at level i. Therefore, the length of the shortest path is $i + 1$ and all other paths should also be of this length. Therefore, the only other possible shortest paths should be from that level. Therefore, when we check if u is equal to w in lines 14-16 we do not care if it has already been discovered or not because we need to increment the counter each time we hit u from level i. Once the counter changes from 0, it indicates that we have discovered w and we can return.

Algorithm Analysis This is a BFS with the only modification that it increments a counter each time it sees w at a particular level and return early (not always). We only process each vertex once since a vertex is never enqueued after it is dequeued. We hit all neighboring edges of a vertex and there are a total of m edges. Thus, it is still 0(m+n).

Solution 2. The following is the solution to exercise 11 on page 111 of the textbook.

```
1.  buildGraph()
2.      for each triplet:
3.      Graph G
4.          add undirected edge between C_i & C_j with weight t_k to G
5.
6.  bool isInfected(C_a, x, C_b, y)
7.      if x > y
8.          return False
9.
10.     mark C_a as discovered
11.     for all u ∈ undiscovered (uninfected) neighbours of C_a
12.         if (u = C_b) ∧ (x ≤ edge_weight) ∧ (edge_weight ≤ y)
13.             return True
14.         else if x ≤ edge_weight
15.             if isInfected(u, edge_weight, C_b, y) = True
16.                 return True
17.
18.     return False
```

Algorithm Explanation For each trace, we make an undirected edge between the two nodes with weight equal to the time at which the two nodes communicate.
After that, the algorithm is just a modified DFS wherein we visit a node only if the edge weight is more than or equal to the current time. And check if it is equal to our destination node or not.

Algorithm Analysis We hit each node once and thus also do not repeat edges if they lead to an already visited node. Thus, it is linear in the total number of edges and vertices and the run time complexity is 0(m+n).

Solution 3. The following is the solution to exercise 2 on page 189 of the textbook.

Part A) When we are building an MST, we look for the minimum weight edge immediately from the nodes which have been added to the tree. When all of these edge weights are squared, the edge which previously had the minimum weight will still have the minimum weight (since all weights are unique and positive). Thus, the new MST will be the same tree, T.
Thus, the statement is **true**.

Part B) Suppose there are two paths from s and t. The first path is made up of 15 edges wherein each has a weight of 1. The second path is made up of 2 edges wherein the first edge has weight 7 and the second edge has weight 2. There are no edges connecting these two paths except both start at s and terminate at t.

The second path is the shortest path when weights are not squared since its total weight is 9 and the other path's total weight is 15.
When the weights are squared however, the first path becomes the shortest path since its total weight is still 15, but the total weight of the second path is 53.
Thus, the statement is **false**.

Solution 4. The following is the solution to exercise 4 on page 190 of the textbook.

```
1.  isSubsequence(S, S')
2.      current ← 0
3.
4.      for each element, elem, in S:
5.          if elem = S'[current]
6.              current ← current+ 1
7.          if current = S'.length
8.              return True
9.
10.     return False
```

Algorithm Explanation Whenever we see the first occurrence of an S' element in S, we add it to our growing list (well, there is no list but we then start looking for the next element in S'). Thus, in a greedy fashion, we keep on "adding" elements of S' as we see them for the first time in S in the correct order to our "list". If that list is equal to S' we return true, otherwise we return false.

Proof of Correctness If the length of the two strings is 1, the greedy solution produces the correct result by inspection.

Let it produce the correct result for the first k where $k < n$ elements of the first string i.e. correctly found if the first l, where $l < m$, elements of the subsequence are present in the correct order in the first k elements of the first string.

When we find the first match i.e. $l + 1^{th}$ element of the subsequence in the first string,

we can safely assume it is a part of the solution for the following reasons:

1. Say there is an occurrence of the $l + 1^{th}$ element of the subsequence later on as well and the rest of the subsequence can be found from there on. But since we have already added the first occurrence of the $l+1^{th}$ element, we can add the rest of the subsequence which appears after the second (or nth) occurrence to the solution.
2. If the rest of the subsequence that is after $l + 2$ starts before the second (or nth) occurrence of the $l + 1^{th}$ element, then we need to add the first occurrence.
3. If there is no occurrence of this element again, we need to make it a part of the solution.

Thus, no matter what, the first occurrence does become a part of the solution. Thus, our greedy algorithm is optimal by strong induction.

Algorithm Analysis We loop over the string of length n once. Thus, the runtime of this algorithm is O(n) i.e. it is linear in the length of the string.

Solution 5. The following is the solution to exercise 7 on page 191 of the textbook.

```
1. order()
2.     sort the jobs in descending order with respect to their f_i times
3.
4.     for job in jobs (descending order)
5.         process in supercomputer
6.         move it to the PC
```

Algorithm Explanation Since all jobs need to be processed by a single supercomputer, no matter what order we feed in the jobs to it, it will always take the same amount of time. However, since there is a PC for each job, we can process the job which takes longest time on the PC first so that it gets completed (or close to completion) by the time a short job arrives to another PC and the total time gets reduced.
Thus, we sort the jobs in descending order of the time they require on a PC.

Proof of Correctness If there is only one job then both the greedy and the optimal solutions will have the same ordering. Thus, the base case holds.

Let us assume that the greedy and the optimal solutions produce the same ordering for the first k jobs and the $(k + 1)^{th}$ job is different.

Let us denote the greedy solution by G and the optimal solution by O. Since the $(k+1)^{th}$ job is different, it implies that in the optimal solution the $(k + 1)^{th}$ job's $f_{(k+1)}$ is less than the $f_{(k+2)}$ of the $(k+2)^{nd}$ job. Lets swap these two jobs in the optimal solution. Now, the longer job finishes earlier in this new swapped schedule since it was scheduled before the shorter job. Thus, we have improved (or remain the same) the optimal solution. We continue doing

this for the remaining pairs and we end up with our sorted, greedy solution. Thus, we do not increase the completion time when we convert the optimal solution to the greedy solution and thus our greedy solution is optimal by strong induction.

Algorithm Analysis The algorithm runs in O(nlogn) time where n is the total number of jobs. The sorting requires O(nlogn) time and after that it is just a linear scan. The runtime is polynomial since it is bounded by O(n) from the bottom.

Solution 6.a. The algorithm to find the longest path in a directed graph is the following.

```
1.  discovered[n]
2.  active[n]
3.
4.  findLongestPath(dg g, source s)
5.      Set visited
6.      maximumDistance[n] ← −∞ for all vertices
7.      discovered[n] ← False for all vertices
8.      active[n] ← False for all vertices
9.
10.     maximumDistance[s] ← 0
11.     discovered[s] ← True
12.     active[s] ← True
13.
14.     helperDFS(g, s, visited, maximumDistance)
15.
16.     for dist in maximumDistance
17.         if dist = −∞
18.             dist = ∞
19.
20. helperDFS(dg g, source s, set visited, maximumDistance)
21.     active[s] = True
22.
23.     for neighbor u in s.allNeighbors
24.         if visited does not contain edge (s, u)
25.             if maximumDistance[u] < maximumDistance[s] + edge_weight of (s, u))
26.                 maximumDistance[u] = maximumDistance[s] + edge_weight of (s, u))
27.
28.             visited.insert((s, u))
29.             if discovered[u] = False ∧ active[u] = False
30.                 helperDFS(g, u, visited, maximumDistance)
31.
32.     active[s] = False
33.     discovered[s] = True
```

Algorithm Explanation We maintain a set (hash set) of all edges that we have visited. We modify our DFS to only go to the neighbor if the edge connecting them has not already

been visited. If it hasn't been visited we update its maximum distance. If the node itself hasn't been discovered and is not currently active, we perform DFS on it. At the end, we loop through all distances if some distances are still $-\infty$, it means that they are not a part of s' component and thus I make them positive ∞.

Algorithm Analysis We visit each node exactly once (we may however update the maximum distance on a node several times but won't explore if it has already been visited). For each node, we visit all its neighboring edges and since we visit each edge exactly once, we visit a total of e edges. Thus, the run time is O(n+e).

Solution 6.b. The algorithm to find the longest path in a dag is the following:

```
1. findLongestPath(dag g, source s)
2.     dist[n] ← −∞ for all vertices
3.     dist[s] ← 0
4.
5.     create a topological ordering of the vertices from source s
6.
7.     for v in the topological order
8.         for neighbor u of v
9.             if dist[u] < dist[v] + weight(v, u)
10.                dist[u] = dist[v] + weight(v, u)
```

Algorithm Explanation Since it is a dag, there exists a topological ordering of the vertices of the graph. For each vertex, we set its distance to negative infinity and then make our source s' distance 0. We traverse in the topological ordering and update distances of its neighbours if their current distances are less than the distance of the current vertex plus the edge weight.

Algorithm Analysis After a topological sort, it is just a linear scan, thus, the topological sort contributes to the overall run time of the algorithm. The complexity is O(n+e).