CS 330 – Fall 2018 – Assignment 2 – Solutions

Problem 1. *(10 pts) Develop an efficient algorithm to find the length of the longest path in a directed graph* $G = (V, E)$.

Hint:

1. *First, observe that if the graph G has a directed cycle, then the maximum path length is infinite, since you can construct a path that keeps going around this cycle forever. Therefore, first devise an algorithm to test whether G contains a directed cycle (and output* ∞ *if it does). Prove the correctness of your algorithm for this step.*
2. *If G contains no cycles, it must be a DAG. Provide an efficient algorithm to output the maximum path length in a DAG. Prove the correctness of your algorithm for this step.*
3. *Analyze the overall running time of your algorithm.*

PART 1 (USING TOPOLOGICAL ORDER):

Observation: Directed cycles cannot be topologically ordered, since they require a start node (a node with no incoming edges).

Algorithm: Run the topological ordering algorithm from class on G. Recall that this algorithm repeatedly searches for a vertex with in-degree zero, and terminates when no such vertex exists. If topological ordering outputs the entire graph then there is no directed cycle. However, if it terminates before outputting the entire graph, there is a cycle, so output ∞.

Proof of correctness: As proven in class, a directed cycle cannot have a topological ordering, because a directed cycle has no root nodes. Since this algorithm terminates once there are no root nodes, it must eventually stop if the input contains a directed cycle. Conversely, when the input does not contain a directed cycle, it produces a topological ordering of the nodes. □

PART 2:

Algorithm: Set up an array $p[\,]$ that at index i contains the max path length ending at node i. We compute these from left to right along the topological order. We compute the length of the paths recursively; for each node, the longest path is exactly one-hop longer, than the longest path ending in one of its incoming neighbors. In order to consider each incoming neighbor of the nodes, we create G_{rev} the reverse graph of G, with all edges of G reversed, as covered in lecture.

`compute_Length`(G):
Initialize all $p[i]$ to zero.
for $i = 2$ to n in topological order on G **do**
  for all incoming edges to i in G of the form (u, i) (using list i in G_{rev}) **do**
    $p_i = \max_u[p_u+1]$ `// longest to i: 1 hop longer than longest path to prev. u`
  end for
end for
return $\max_i p_i$ by scanning the p_i's

Note that we can also compute the path by recording the best u for each i in an array: $\text{pred}[i] = \text{argmax}_u(p_u+1)$, and at the end travel backward along the longest path to output it. Our algorithm clearly produces a valid path. We will now prove that the value reported is the longest path length.

Proof of correctness: For each node v, the longest path ending in v is exactly one-hop longer than the longest path ending in one of its incoming neighbors. Further, because the nodes are considered in topological order, it is clear that at the time the path-length for v is considered all the nodes preceding v on the longest path have been processed already. Hence, all incoming neighbors of v will have a value assigned.

Run-time analysis: Here we analyze the running time for part 2 only. The total running time (parts 1 and 2 together) depends on which method was used to detect cycles.

Total time is $O(n + m)$, adding up several component running times. (1) Computing G_{rev}: $O(n + m)$ time, (2) Traversing the topo-sorted graph, node by node: $O(n)$ time, (3) Total number of edge comparisons: $O(n + m)$, where each comparison to compute the max is constant time, (4) Traversing the longest path in reverse: $O(n)$ time.

Problem 2. *(10 pts) Given a sequence q of n numbers, give an $O(n^2)$ algorithm to find the longest monotone increasing subsequence in q. As always, prove the correctness of your algorithm and analyze its running time.*

Hint: Find a graph representation for this problem and use a graph algorithm (from class, lab or homework) to solve it.

Our strategy is to use the longest path algorithm we just developed in Problem 1.

Graph construction and algorithm: Create n vertices numbered $1, \ldots, n$, one for each element of the sequence. For all pairs of vertices i, j where $i < j$, draw a directed edge from vertex i to vertex j if $q_i < q_j$ (satisfying the monotonicity requirement). Note that edges only go from smaller numbered vertices to larger, so the $1, \ldots, n$ ordering of the vertices is a topological ordering and the graph is thus a DAG. Now run the algorithm from Problem 1 part 2 to produce the maximum path length p on this DAG. Output $p + 1$ as the maximum subsequence length.

Running time: This algorithm has two parts (1.) construct the graph (2.) find the longest path. To construct the graph we can run a brute-force style algorithm, where we traverse the nodes in the order of the sequence they are given. For each node v we compare it against all nodes (numbers) that are later than v in the sequence. This takes $O(n)$ comparisons for the first node in the sequence, $O(n - i)$ for the ith node. Making it a total of $\frac{n(n-1)}{2} = O(n^2)$ comparisons. Note, that we cannot do better than this in terms of big-oh. The reason for this is that in total exactly half of the possible $\binom{n}{2}$ edges will be in the final graph. Simply listing all of these requires $O(n^2)$ time.

Computing the longest path takes $O(m + n)$ time. Since $m = \text{O}(n^2)$, the total running time is $\text{O}(n^2)$. [Note: you must ultimately write this algorithm as a function of n, since m is not a parameter to this problem.]

Correctness: We need to show that when the longest path algorithm outputs p then the longest subsequence is of length $p+1$. To do it, we show that there is a bijection between paths and subsequences. Take any non-empty path of length $k \geq 1$ on G: it starts at a vertex i_1, then follows a sequence of edges to successively higher numbered vertices $i_2, \ldots, i_{k+1}$. (Note: a path of length k contains k edges, but $k + 1$ vertices). This path is equivalent to a sequence $q_{i_1}, q_{i_2}, \ldots, q_{i_{k+1}}$. By the edge construction rule, each successive edge in the k-hop path, e.g., (i_5, i_6), corresponds to an increasing pair in the sequence, e.g., (q_{i_5}, q_{i_6}). So the entire subsequence of length $k + 1$ is monotonically increasing. By this bijection, the longest path in G maps to the longest subsequence in q.

Problem 3. *(10 pts) Chapter 4, Exercise 6, on. p. 191. Swim, bike, run.*

Let the time used for swimming, biking and running for contestants be $(s_1, b_1, r_1), (s_2, b_2, r_2), \ldots, (s_n, b_n, r_n)$. The algorithm is: sort the contestants by the sum of $(b_i + r_i)$ in decreasing order (and for simplicity, assume all contestants have different $b_i + r_i$ values), so that the contestant $(b + r)$ is greatest goes first and whose $(b + r)$ is the second largest to go after that, and so on. At the meet, schedule contestants with no slack, so that the pool is always in use. *Running time:* $\text{O}(n \log n)$ to sort the contestants.

Correctness: Proof by contradiction (exchange argument): Suppose there is an OPT (with no slack) which has all contestants finish earlier than my algorithm. Therefore, there must be a contestant i whose $(b_i + r_i) < (b_j + r_j)$ for another contestant j, yet i goes before j. As in the example in class, when there is an inversion of this form, there must be an inversion of consecutive contestants. Consider OPT with and without this inversion: with the inversion, if i swims before j, and the total time for i and j to both finish is $s_i + s_j + b_j + r_j$ (i finishes first and j finishes later); without the inversion, the total time for i and j to both finish is more complex, since we do not know who will finish first:

$$\max(s_j + b_j + r_j, s_j + s_i + b_i + r_i).$$

However, note that both terms in this max are

$$\leq s_i + s_j + b_j + r_j$$

(the original total finishing time), since $s_i \geq 0$ and $b_i + r_i < b_j + r_j$. Therefore, by removing the inversion, the total finishing time of these two contestants does not increase, and the finishing time of all other contestants does not change. So every time when we encounter this situation in OPT, inverting i with j will not increase the time of OPT, so if we invert all the situations like that, OPT is exactly the greedy algorithm: both are sorted by decreasing order of $(r + b)$, with no inversions.

Problem 4. *(20 pts)[programming assignment]*

Consider a different version of the job scheduling problem that minimizes the weighted sum of job completion times. In this version, each of n jobs has a weight w_i and a duration t_i, and all must be scheduled in serial order, starting at time zero (with no slack). A schedule S is therefore an ordering of the jobs, and the completion time of job i in S, $C_i(S)$, is defined to be the sum of the durations t_k of all jobs (including i) that precede i in S. Find the schedule $S*$ that minimizes over all S: $\sum_{i=1}^{n} w_i C_i(S)$.

1. *Provide a counterexample to the optimality of ordering by heaviest-weight first. [1 pt]*
2. *Provide a counterexample to the optimality of ordering by shortest-job first. [1 pt]*
3. *Design and analyze an efficient algorithm for computing an optimal schedule.$*$ Hint: use an exchange argument, similar to the one used for the minimizing lateness problem for the proof of optimality.*

($*$) For this problem you need to submit a python file, with your code and comments explaining the steps and with regard to its running time. You also need to provide answers to parts (1) and (2) as well as the proof of correctness for part (3) on your pdf with the answers to the other questions.

counterexamples:

1. Observe, that once a time t_i is part of the schedule, it will be counted in every subsequent summation term. Thus, any example where the heaviest weight job also has a long duration shows it's not optimal. For example $(10, 10), (1, 1), (1, 1) \ldots, (1, 1)$.
2. The later a job is scheduled, the larger $C(S_i)$ is that the corresponding weight is multiplied with. Thus in an example where a heavy weight is scheduled (too) late is suboptimal. $(1, 1), (1, 1), \ldots, (1, 1), (10, 10)$.

Algorithm idea: sort the jobs in descending order of the ratio, $\frac{wi}{ti}$, and process jobs in this order.

Running time: Just computing ratios, $O(n)$, then sorting, so clearly $O(nlogn)$.

Proof of correctness: To see how we came up with this greedy approach, first consider the cost of the initial ordering of the jobs: job 1 completes at t_1, job 2 completes at $t_1 + t_2$, etc. The weighted sum of completion times is therefore $W(S) = w_1 t_1 + w_2(t_1 + t_2) + ... + w_n(t_1 + t_2 + ... + t_n)$.

Let S be the schedule output by our algorithm. Let i and $i + 1$ be any two consecutive jobs in S.

Claim 1. *Let S' be the schedule where jobs i and $i+1$ are switched. Then $W(S) < W(S')$.*

We proof this claim below, for now, accept that it is true. Then the proof of correctness for the algorithm is as follows; We use an exchange argument. Recall, that if there are two schedules A and B and in A job i precedes job j while in B job j is first, then we say at i and j are in inversion. Let us assume that S is not an optimal schedule, let $S*$ be an optimal solution with *minimum number of inversions*. We have shown in class, that if there is an inversion, then there must be two consecutive jobs that are in inversion. Now apply the claim to the these consecutive jobs. It is clear that by swapping them we can only improve on the value of $W(S*)$ in contradiction to the fact that it is optimal and has minimum number of inversions. Hence, this shows that an optimal solution cannot have any inversions. But then this solution is identical to S.

Proof of claim: If we compare $W(S)$ and $W(S')$ they only differ in the last two summation terms (that correspond to jobs i and $i+1$). The last two terms look like this (?? stands for the unknown relation between the two sides):

$$w_i(t_1+t_2+\ldots+t_{i-1}+t_i)+w_{i+1}(t_1+t_2+\ldots+t_i+t_{i+1}) \;??\; w_{i+1}(t_1+t_2+\ldots+t_{i-1}+t_{i+1})+w_i(t_1+t_2+\ldots+t_{i-1}+t_i+t_{i+1})$$

If we remove all the redundant terms that appear on both sides we get

$$w_{i+1}t_i \;??\; w_it_{i+1}$$

Note that by definition of the algorithm we have $w_i/t_i > w_{i+1}/t_{i+1}$. Thus, this means that

$$w_{i+1}t_i < w_it_{i+1}$$

proving $W(S) < W(S')$ $\square$.