Solutions for Homework 3

Assigned Problems

Following are the problems to be handed in, 25 points each. Maximum score for this homework is 100 points. We will take your best four attempted problems.

1. (**Graphs**, 2-page limit – your solutions should fit on two sides of 1 page).
   You are managing the creation of a large software product. You have the following information:

   (a) A set V of n small projects that are part of your software product.

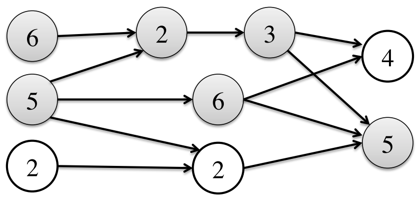
   (b) A set E of pairs of projects in V. A pair (u, v) is in E if project u must be completed before project v is started. There are no cycles in E.

   (c) For each project $u \in V$, a duration $t_u \in \mathbb{N}$. This is the amount of time the project will take from start to finish.

   You can do any number of projects in parallel, but you can't start a project before all of its predecessors (according to the list E) are completed.

   Given this information, you want to know two things: First, how soon can each of the projects be completed? Second, which projects are *critical*? We say a project is critical if increasing its duration by 1 would delay the completion of the entire product.

   For example, in the following picture, nodes represent projects, and numbers inside nodes are durations. Critical nodes are colored gray.



   Give an algorithm that takes the inputs above and returns:

   (a) For each project $u \in V$, the earliest possible completion time $c(v)$.

   (b) A list of the critical projects.

   Give the running time and space complexity for your algorithm. Prove that your algorithm is correct. (*Hint:* You may want to compute the completion times $c(v)$ first, then look for critical projects.)

Algorithm 1: Problem 1 Solution

```
1  DAG $G = (V, E)$ in adjacency list representation.;
2  $G \leftarrow TopologicalSort(G)$;
3  Compute list of incoming edges for each vertex;
4  Compute completion times;
5  for $v = 1$ to $|V|$ do
6      $latest \leftarrow 0$;
7      for each $u$ with an edge to $v$ do
8          $latest \leftarrow \max(latest, c(u))$
9      $c(v) \leftarrow t_v + latest$;
10 Compute critical vertices, using reverse chronological sort.;
11 $last \leftarrow \max_{v \in V} c(v)$;
12 Initialize $critical(u) = false$ for all $u \in V$;
13 for $u = |V|$ down to 1 do
14     if $c(u) = last$ then
15         $critical(u) = true$;
16     for each $v$ such that $(u, v) \in E$ do
17         if $critical(v) = true$ and $c(v) = c(u) + t_v$ then
18             $critical(u) \leftarrow true$;
```

Solution: The biggest hint in the statement of this problem is that input is a directed acyclic graph. The first step, therefore, should be to sort the graph topologically. The next hint is that it makes sense to *first* compute the completion times, and *then* figure out which projects are critical.

To compute the completion times, we process the graph in topological order and compute $c(v) = \max_{(u,v) \in E} c(u) + t_v$.

- State the running time and space complexity of your algorithm.

  The algorithm takes time and space $O(m + n)$ where $m = |E|$ and $n = |V|$. Topological sort takes time $O(m + n)$, and the remaining operations process each edge and vertex a constant number of times.

- Prove that your algorithm is correct.

  Completion times: First, we note that the formula $c(v) = \max_{(u,v) \in E} c(u) + t_v$ is correct because, by definition, a project can start as soon as all the projects it depends on are completed, and no sooner. Second, our code will compute $c(v)$ correctly as long as all the numbers $c(u)$ were computed *before* we try to compute $c(v)$. The topological sorting step guarantees this. So the completion times are computed correctly.

  Critical vertices: A project u_1 is critical iff there is a path of vertices $u_1, u_2, ..., u_k$ such that u_{i+1} can start immediately after u_i (for $i = 1$ to $k - 1$) and u_k is one of the last projects completed. In other words, u is critical if and only if there is a vertex v such that (a) there is an edge (u, v) and (b) v is critical and (c) v starts immediately after u is done. We can check condition (c) by checking if $c(v) = c_u + t_v$. Our algorithm checks (a), (b) and (c) for each vertex. Moreover, it is guaranteed that $critical(v)$ was correctly computed before v is processed since it considers the vertices in *reverse* topological order.

2. (**Dynamic Programming**, 2-page limit – your solutions should fit on two sides of 1 page).

You are managing the construction of power plants along a river. As opposed to the last problem that involved construction along a river, this time around the possible sites for the power plants are given by real numbers $x_1, ..., x_n$, each of which specifies a position along the river measured in miles from its spring. Assume that the river flows in a completely straight line. Due to differences in the water flow, the position of each power plant influences its capacity for energy production. In other words, if you place a power plant at location x_i, you will produce a quantity of electricity of $r_i > 0$ MW[1].

Environmental regulations require that every pair of power plants be at least 5 miles apart. You'd like to place power plants at a subset of the sites so as to maximize total energy production, subject to this restriction. The input is given as a list of n pairs $(x_1, r_1), ..., (x_n, r_n)$ where the x_i's are sorted in increasing order.

(a) Your foreman suggests some very simple approaches to the problem, which you realize will not work well. For each of the following approaches, give an example of an input on which the approach does not find the optimal solution:

i. *Next available location:* put a power plant at $i = 1$. From then on, put a power plant at the smallest index i which is more than five miles from your most recently placed power plant.

ii. *Most profitable first*: Put a power plant at the most profitable location. From then on, place a power plant at the most profitable location not ruled out by your current power plant.

(b) Give a dynamic programming algorithm for this problem. Analyze the space and time complexity of your algorithm.

To make it easier to present your answer clearly, try to follow the steps below (as with any design process, you may have to go back and forth a bit between these steps as you work on the problem):

i. Clearly define the subproblems that you will solve recursively (note: weighted interval scheduling should be a good source of inspiration here).

ii. Give a recursive formula for the solution to a given subproblem in terms of smaller subproblems. Explain why the formula is correct.

iii. Give pseudocode for an algorithm that calculates the profit of the optimal solution. Analyze time/space and explain why the algorithm is correct (based on previous parts).

iv. Give pseudocode for an algorithm that uses the information computed in the previous part to output an optimal solution. Analyze time/space and explain why the algorithm is correct.

Solution: **Note**: *This problem is actually a special case of weighted interval scheduling. That is, one could solve the problem by setting up a set of intervals, each five miles long, centered at the power plant locations, where the weight of an interval is the profit associated with the corresponding bollboard.*

(a) Your foreman suggests some very simple approaches to the problem, which you realize will not work well. For each of the following approaches, give an example of an input on which the approach does not find the optimal solution:

[1]this stands for megawatts, but don't concern yourself with the unit of measurement

i. *Next available location:* put a power plant at $i = 1$. From then on, put a power plant at the smallest index i which is more than five miles from your most recently placed power plant.

Power plant #	1	2
Distance	1	2
Profit	1	100

ii. *Most profitable first*: Put a power plant at the most profitable location. From then on, place a power plant at the most profitable location not ruled out by your current power plants.

Power plant #	1	2	3
Distance	2	5	8
Profit	2	3	2

(b) Give a dynamic programming algorithm for this problem. Analyze the space and time complexity of your algorithm.

To make it easier to present your answer clearly, try to follow the steps below (as with any design process, you may have to go back and forth a bit between these steps as you work on the problem):

i. Clearly define the subproblems that you will solve recursively (note: weighted interval scheduling should be a good source of inspiration here).

Assume that power plants are sorted in increasing order of distance from the start of the river. The subproblems are then defined as OPT(j) = maximum profit you can obtain using only power plants $1, 2, ..., j$. For simplicity, define $OPT(0) = 0$.

ii. Give a recursive formula for the solution to a given subproblem in terms of smaller subproblems. Explain why the formula is correct.

For every index j, let $p(j)$ be the highest index of a power plant that is five or more miles closer to the beginning of the river than power plant j. That is,

$$p(j) = \max\{i : x_i \leq x_j - 5\}$$

When there are no such power plants, define $p(j)$ to be 0.

Then we can define $OPT(j)$ recursively:

$$OPT(j) = \max(OPT(j-1), r_j + OPT(p(j)))$$

The formula holds because one of two cases always holds: either j is not in the optimal solution, in which case we are free to choose the best solution among bilboards $\{1, ..., j-1\}$, or j is in the optimal solution, in which case all the power plants between $p(j)+1$ and $j-1$ are ruled out by the presence of power plant j, but we are free to choose the best solution among power plants $\{1, ..., p(j)\}$ (and we add to that optimum the profit obtained from power plant j).

iii. Give pseudocode for an algorithm that calculates the profit of the optimal solution. Analyze time/space and explain why the algorithm is correct (based on previous parts).

This algorithm computes the recursive formula above, bottom up. Note that we store the optimum as well as whether or not the optimum was obtained by including j.

The algorithm is correct because of the recursive formula above: the important point is that all the information needed to compute $OPT(j)$ is prepared before execution j of the for loop.

The algorithm takes $\Theta(n)$ time when the input is sorted by x_i. On an unsorted input, the sorting makes the total time be $\Theta(n \log n)$.

iv. Give pseudocode for an algorithm that uses the information computed in the previous part to output an optimal solution. Analyze time/space and explain why the algorithm is correct.

```
Algorithm 2: POWER PLANT-PROFIT($(x_1, r_1), ..., (x_n, r_n)$)
▷ $OPT$ is an array of integers, $Include$ is an array of booleans.;
▷ $Include[j]$ indicates whether power plant $j$ was used in the solution with value $OPT[j]$.;
Compute $p[1], ..., p[n]$. ;
for $j \leftarrow 1$ to $n$ do
    if $OPT[j-1] > r_j + OPT[p[j]]$ then
        $OPT[j] \leftarrow OPT[j-1]$;
        $Include[j] \leftarrow$ FALSE;
    else
        $OPT[j] \leftarrow r_j + OPT[p[j]]$;
        $Include[j] \leftarrow$ TRUE ;
return $OPT[n]$;
```

```
Algorithm 3: POWER PLANT($(x_1, r_1), ..., (x_n, r_n)$)
Compute $p, OPT, Include$ as in previous algorithm.;
$final \leftarrow$ empty list;
$j \leftarrow n$;
while $j > 0$ do
    if $Include[j] ==$ TRUE then
        Add $j$ to $final$ ▷ Include $j$ and skip to next available power plant;
        $j \leftarrow p[j]$;
    else
        $j \leftarrow j - 1$ ▷ Don't include $j$;
return $final$;
```

The algorithm runs in time $\Theta(n)$. The correctness is already proven in the previous part.

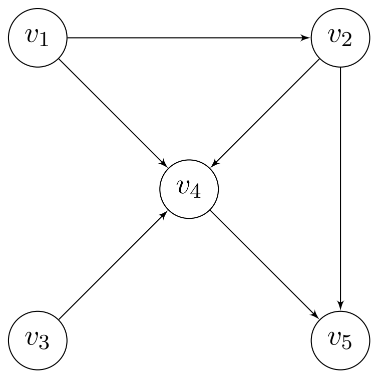
3. (**Dynamic Programming**, 2-page limit – your solutions should fit on two sides of 1 page).

Let $G = (V, E)$ be a directed graph with nodes v_1,...,v_n. We say that G is an *ordered graph* if it has the following properties.

(i) Edges go from a node with a lower index to a node with a higher index. In other words, every directed edge has the form (v_i, v_j) with $i < j$.

(ii) Each node with the exception of v_n has at least one edge leaving it. In other words, for every node $v_i, i = 1, 2, ..., n-1$, there is at least one edge of the form (v_i, v_j).

The length of a path is the number of edges in it. See the following example.



The correct answer for the above graph is 3: The longest bath from v_1 to v_n uses the three edges (v_1, v_2), (v_2, v_4), and (v_4, v_5). The goal in this question is to solve the following problem.

Given an ordered graph G, find the length of the longest path that begins at v_1 and ends at v_n.

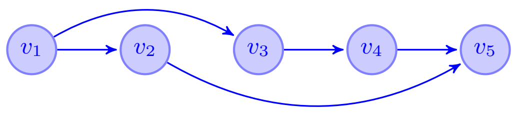
(a) Show that the following algorithm does not correctly solve this problem, by giving an example of an ordered graph on which it does not return the correct answer.

```
Set $a = v_1$
Set $L = 0$
While there is an edge out of node $a$
    Choose the edge $(a, v_j)$ for the smallest possible $j$
    Set $a = v_j$
    Increase $L$ by 1
Return L as the length of the longest path
```

In your example, say both what the correct answer is and what the algorithm above finds.

(b) Give an efficient algorithm that takes an ordered graph G and returns the *length* of the longest path that begins at v_1 and ends at v_n.

Solution:



(a) However, the right answer is 3, corresponding to the path $v_1 \to v_3 \to v_4 \to v_5$.

(b) The idea is to use dynamic programming. Let $OPT[i]$ denote the length of the longest path from v_i to v_n. The base case is $OPT(n) = 0$, since the longest path from v_n to v_n has 0 edges. It is easy to show by induction that $OPT(i) = 1 + \max_{j:(v_i,v_j)\in E} OPT[j]$.

Algorithm 4: LongestPathLength(G, n)

```
1 ▷ $G$ is a graph with $n$ vertices and $m$ edges in adjacency list representation;
2 Let $M[1..n]$ be an array ;
3 $M[n] \leftarrow 0$;
4 for  $i \leftarrow n-1$ to $n$ do
5     $M[i] \leftarrow 0$;
6     for all edges $(v_i, v_j)$ in $v_i$'s adjacency list do
7         $M[i] \leftarrow max(M[i], 1 + M[j])$
8 return $M[1]$
```

Correctness We will prove by induction that at the end of the algorithm, $M[i]$ contains the length of the longest path from v_i to v_n. Recall that the outdegree of each vertex, besides v_n, is at least one. It follows that there is a path from every node to v_n. We will prove this claim in the same inductive argument. Base case (when $i = n$) is trivial: clearly, v_n is reachable from v_n, and the length of the path is 0. For induction hypothesis, assume that v_n is reachable from $v_{i+1}, \ldots, v_n$ and that $M[i+1], \ldots, M[n]$ are computed correctly. Since v_i has an outgoing edge to v_k with $k > i$ and since, by induction hypothesis, v_n is reachable from v_k, we get that v_n is also reachable from v_i. Consider a path P of the maximum length from v_i to v_n, and let (v_i, v_k) be its first edge. By induction hypothesis, $M[k]$ contains the length of the longest path from v_k to v_n. The length of P is $1 + M[k]$, which is computed at some point on line 6. The value we store at $M[i]$ always corresponds to the length of a path from v_i to v_n, so it never exceeds the length of P.

It follows that $M[1]$, returned by the algorithm, holds the length of the longest path from v_1 to v_n.

Time and Space Complexity. To compute $M[i]$, we access the number of entries equal to the out-degree of vertex v_i. The total number of accesses to the table is $\sum_{v\in V}$ out-degree$(v) = m$. So, all updates to the table M take O(m) time. Recall that the problem stipulates that the outdegree of each node, besides v_n, is at least 1, so that $m \geq n - 1$.

The only thing we store is the table, so space is $O(n)$.

4. (**Graphs**, 2-page limit – your solutions should fit on two sides of 1 page).

Suppose that you have a directed graph $G = (V, E)$ with an edge weight function w and a source vertex $s \in V$. The weights can be negative, but there are no negative weight cycles. Furthermore, assume that all edge weights are distinct (i.e. no two edges have the same weight). The single source shortest path problem is to find the shortest path distances from s to every vertex in V.

(a) Suppose that you also guaranteed that for all $v \in V$, a shortest path from s to v has increasing edge weights. Give an algorithm to find the shortest path distances from s to every vertex in V. Analyze the running time of your algorithm and explain why it is correct. For full credit, your algorithm should run in time $O(V + E \log E)$.

(b) A sequence is *bitonic* if it monotonically increases and then monotonically decreases, or if by a circular shift it monotonically increases and then monotonically decreases. For example the sequences $(1, 4, 6, 8, 3, -2)$, $(9, 2, -4, -10, -5)$, and $(1, 2, 3, 4)$ are bitonic, but $(1, 3, 12, 4, 2, 10)$ is not bitonic. Now, suppose that the sequences of edge weights along the shortest paths are no longer guaranteed to be increasing, but instead are guaranteed to be bitonic. Give a single source shortest path algorithm, explain why it is correct, and analyze its running time. For full credit, your algorithm should run in time $O(V + E \log E)$.

Solution:

(a) First sort the edges in order of increasing weight. Call INITIALIZE-SINGLE-SOURCE (see below) and then relax the edges one at a time, according to this ordering. Then the edges along every shortest path would be relaxed in order of their appearance on the path. (We rely on the uniqueness of edge weights to ensure that the ordering is correct.) The path-relaxation property (Lemma 24.15) would guarantee that we would have computed correct shortest paths from s to each vertex. The running time of INITIALIZE-SINGLE-SOURCE is $O(V)$, of the sort step is $O(E \log E)$, and of the relaxation steps $O(E)$.

Algorithm 5: INITIALIZE-SINGLE-SOURCE($G = (V, E), Sources$)

```
1 for each vertex $v \in V[G]$ do
2     $d[v] = \infty$ #priority of vertex v in queue ;
3     $parent[v]$ = NIL #parent of v in the shortest path tree rooted at s
4 $d[s] = 0$
```

Lemma 24.15: Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \to \mathbb{R}$, and let $s \in V$ be a source vertex. Consider any shortest path $p = \langle v_0, v_1, ..., v_k \rangle$ from $s = v_0$ to v_k. If G is initialized by INITIALIZE-SINGLE-SOURCE(G, s) and then a sequence of relaxation steps occurs that includes, in order, relaxing the edges $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$, then v_k. $d = \delta(s, v_k)$ after these relaxations and at all times afterward. This property holds no matter what other edge relaxations occur, including relaxations that are intermixed with relaxations of the edges of p.

(b) Observe that a bitonic sequence can increase, then decrease, then increase, or it can decrease, then increase, then decrease. That is, there can be at most two changes of direction in a bitonic sequence. Any sequence that increases, then decreases, then increases, then decreases has a bitonic sequence as a subsequence.

If we weaken the condition so that the weights of the edges along any shortest path increase and then decrease, we could relax all edges one time, in increasing order of weight, and then one more time, in decreasing order of weight. That order, along with uniqueness of edge weights, would ensure that we had relaxed the edges of every shortest path in order, and again the path-relaxation property would guarantee that we would have computed correct shortest paths.

To make sure that we handle all bitonic sequences, we do as suggested above. That is, we perform four passes, relaxing each edge once in each pass. The first and third passes relax edges in increasing order of weight, and the second and fourth passes in decreasing order. Again, by the path-relaxation property and the uniqueness of edge weights, we have computed correct shortest paths.