Algorithms #5

Jose Nino

Problem 1 a)

$e_1 = 40 \quad e_2 = 15 \quad e_3 = 65 \quad e_4 = 40$

$h_1 = 90 \quad h_2 = 65 \quad h_3 = 140 \quad h_4 = 90$

With the algorithm

Choses $h_2 \rightarrow 65$
Choses $e_3 \rightarrow 65$
Choses $e_4 \rightarrow 40$
170

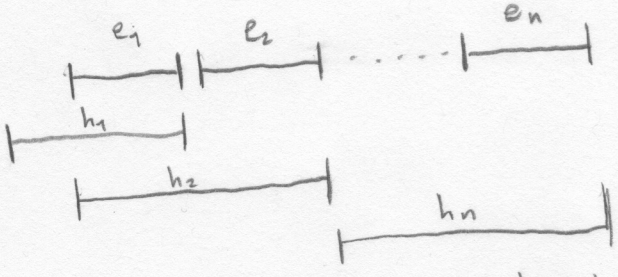
Maximum Possible

Choose h_1 or e_1 does not make a difference, either choice results in higher
chose h_3 140
chose e_4 40

220 if choose e_1
270 if choose h_1

Problem 1b)

lets format this problem like a scheduling problem where:



We can see that this reformating fits the original rules. where we can't schedule the h_{i+1} job if we had scheduled the h_i or e_i job.

Now lets suppose that OPT(n) is the optimal solution for up to week n.

at week n we can say that

a) if $e_n \in OPT(n)$ then $OPT(n) = e_1 + OPT(n-1)$ because we can chose h_{n-1} or e_{n-1}

b) if $e_n \notin OPT(n)$ the either a) $h_n \in OPT(n)$ then $OPT(n) = h_n + OPT(n-2)$

because we can't choose anything from the previous week.

or $h_n \notin OPT(n)$ so $OPT(n) = OPT(n-1)$
but note that that last situation is taken into account by h_{n+1} been selected so it is the case that no assignment been choosen on week n was already accounted for.

This leads us to the following algorithm (I have chosen to explain it in the bottom-up manner).

```
Week(n){
1. M[0] = 0
2. M[1] = h1
3. i = 2
4 while i <= n {
5     M[n] = max(e_n + M[n-1], h_n + M[n-2])
  }
6 return M[n]
}
```

Proof of Correctness.

Base case: for week 0 the accumulated value is trivially 0
for week 1 the accumulated value is h_1 (note that it could have been e_1, and it would have change the max amount of points but the algorithm would still return the correct answer).

Inductive hypothesis: lets assume that M[n] holds the maximum amount of points up to week n, and this is true for all values up to n.

Inductive step: Want to show that when we calculate the value stored at M[n+1] (the Max value up to week n+1) this is in fact the max value up to week n+1.

The algorithm chooses either $h_{n+1} + M[n-1]$ or $e_n + M[n]$ for the value at M[n+1]. And we know that M[n] contains the max value at week n and M[n-1] has the max value at week n-1. by inductive hypothesys. Moreover we know that we are restricted to selecting h_{n+1} if we use the value M[n-1] and select e_{n+1} if we use the value at M[n]. This means that for either value [M[n-1], M[n]) there is no way to increase

their value even further, and thus if we chose the maximum between M[n-1] + hn ; M[n] + en it must be that we have selected the maximum value we can put at index n+1 in the M array. Thus, the value at M[n+1] represents the maximum amount of points accumulated at week n+1. ▨

Proof of Running Time.

In order to calculate the value for n weeks we execute step 1 one time in O(1) time, step 2 in O(1) time, step 3. in O(1) time, step 6 in O(1) time.

The loop is the interesting part.

Note that each call of step 5 just takes the maximum of two sums (each indexing the array, which is constant time) and doing the sum and taking the max is a constant time process.

Now this loop occurs n-1 times, therefore because the cost of the steps in the loop is constant, then the cost of the loop is O(n).

Altogether the total cost of the algorithm to calculate the maximum point value for n weeks is O(n).

Problem 2.

First I am going to describe the graph structure and then I am going to write the algorithm, then I am going to prove its correctness and running time.

The graph will be represented by an adjacency list. Moreover, each node will have its id (what v_i it is), the current longest distance from 1 (initialized to -1, except v_1 which starts with 0), and a list of nodes you can take to get from yourself to V1 (the reverse of this list is obviously a path from 1 to that node).

Now lets write the algorithm.

```
LongestPathFromV1(n){
1  if V1 does not have outgoing edges return NULL
2  else{
3     for each neighbor of V1
4        Change the neighbor's distance to 1
5        Add to its list V1
   }
6  for V2 to Vn {
7     If Vi has distance -1
8        Skip him, there is no path from V1 to Vi
9     else {
10       for each neighbor of Vi {
11          Distance of neighbor = max (distance of neighbor, distance of Vi + 1)
12          if (Distance of neighbor == distance of Vi + 1)
13             Update the list of the neighbor is the list of Vi + Vi
         }
      }
14 Return the list of Vn
}
```

This algorithm depends on a lot of things being true so I am going to break up its prove of correctness.

Proof of Correctness

(1.) The ordering $v_1, v_2, \ldots, v_n$ is a topological sort over the graph. By definition a topological sort is a linear ordering of the vertices of the graph such that for every directed edge uv, vertex u comes before vertex v in the ordering. In a forward graph every edge is in the form uv such that $u < v$ so if we

Order the vertices from 1 to n in ascending order if edges (v_i, v_j) exists in the graph, then it must be that $(i < j)$ because it is a forward graph and v_i comes before v_j in our ordering, making our ordering a topological sort.

(2) The fact that our ordering is a topological sort of the graph will aid us in proving that by the time we have reached vertex v_i in the algorithm we have visited all paths that lead to v_i from v_1 exactly once.

Base case: Our graph is composed by vertices $v_1, \dots, v_n$ where if edge (v_i, v_j) exists then $i < j$. Trivially when we visit node v_1, we have visited all paths that lead to v_1 from v_1 because there are none. And we have seen all edges of the type (v_x, v_y) where $x < 1$ and $1 \leq y$ because there are none.

Inductive hypothesis: Assume that when we visit v_i we have seen all the paths that lead to v_i from v_1. Moreover, because our ordering of the vertices is a topological sort of the graph it must be that we have also explored all edges of the type (v_x, v_y) where $x < i$ and $i \leq y$ if there was a path from v_1 to v_x.

Inductive step: Now we visit node $i+1$. Because of topological sorting we know that we had visited all edges from nodes $< i$ that had a path from v_1 that led to node v_{i+1}. Lastly in our inductive hypothesis we claim we had also seen all paths that lead from v_1 to v_i so if there was an edge from v_i to v_{i+1}, it was checked exactly once when we visited v_i, and this was the last possible path that might have led from v_1 to v_{i+1}. So by the time we visit node $i+1$. We have had to explore all paths from v_1 to v_{i+1} exactly once (if such paths exist).

(3) So now we know that once we have reached v_i in our algorithm we definitely know all possible paths leading to v_i from v_1. This lets us state two things. 1) Because when we were exploring paths to v_i from v_1 we were taking the longest path, if must be that when we visit v_i itself the length of the path and the path from v_1 to v_i is maximal.

2) If when we visit v_i the length of the path is -1, then the list of the node is Null, and there is no path from v_1 to v_i.

Altogether, by the discussion of the last three points if we explore all vertices from V_1 to V_n in ascending order by the time we get to node V_n we would have explored all possible paths leading to V_n from V_n and the value and list stored in V_n will be maximal. If this length is -1 and the list is NULL then there was no path from V_1 to V_n.

Running time.

It is clear that steps 1, 2, 4, 5, 7, 8, 9, 11, 12, 13, 14 take O(1) time as they are all comparisons, assignments, or appending to the end of the list (which if we keep a pointer to the tail of the list, is O(1)).

The interesting running time analysis comes in the loops that I have established at steps 3, 6, 10

First of all one could imagine having embedded the exploration of V_1 done in line 3 to the loop at 6 and the nested loop at 10, but I decided to keep the special case outside of the bigger iteration.

In any case, what we are doing in general for every node V_i from 1 to n is that if node V_i does not have distance -1 we explore every neighbor of V_i (thus having to traverse all of V_i edges). (the distance is not -1).

Therefore for every node we visit, in the worst case we end up doing a constant amount of work for all of its neighbors (that is the degree of that node).

Moreover the algorithm visits all the nodes exactly Once to determine if we have to work on its edges (neighboring nodes).

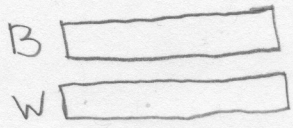
Altogether, we visit all the nodes, and in the worst case we do a constant amount of work for all edges in a particular node, therefore by the end of the algorithm we have visited all vertices exactly once and in the worst case done constant amount of work for all edges in the graph making the algorithm run in $O(V+E)$ time were V is the total amount of vertices and E is the total amount of edges.

Note that the algorithm returns a list with the longest path from V_1 to V_n

Problem 3

Note: in discussing the problem I used n as an arbitrary day within the range of days. not the final day in the optimal schedule.

For this problem lets consider 2 arrays.



1-indexed not 0 indexed

The array B contains at B[n] the maximum total profit possible if at day n we are in Baltimore, regardless of where we have been in the previous n-1 weeks. The same logic holds for array W.

WLOG Note that if we are in Baltimore in day n our profit for day n depends on if we were in Baltimore the day before, in which case our profit for day n will be B_n, or if we were in Washington the day before, in which case our profit for day n will be $B_n - M$.

Therefore at day n we are either in Baltimore or Washington and for each option we have two possible profit values.

Altogether we can calculate our profit for an optimal schedule for n days with the following algorithm

```
MaxProfit(n){
1  B[1] = B1
2  W[1] = W1
3  i = 2
4  while i <= n {
5       B[i] = max(B[i-1] + Bi , W[i-1] + Bi - M)
6       W[i] = max(W[i-1] + Wi , B[i-1] + Wi - M)
7       i+=1
   }
8  return max(B[n], W[n])
}
```

Proof of correctness.

Base case for one day it is trivial that the profit for an optimal schedule if we are in Baltimore in day 1. is B_1 and if we are in Washington it is W_1. So the profit for the optimal schedule is the max between B_1 and W_1.

Inductive hypothesis: Assume that B[n] holds the profit for the optimal schedule that ends in baltimore at day n, and the same for W[n], regardless of where we were in the past n-1 days. So it must be that the max of these

two values give us the profit of an optimal schedule.

Inductive step.: Now for day $n+1$ we have two options, represented by $B[n+1]$ and $W[n+1]$: we end up in Baltimore on day $n+1$ or we end up in Washington in day $n+1$.

(1) the accumulated profit that can be made on day $n+1$ if we end up in Baltimore is the accumulated profit of day n if we were in Baltimore on day n + the profit made on day $n+1$ in Baltimore, or the accumulated profit of day n if we were in Washington on day n + the profit made on day $n+1$ in Baltimore - M (the cost of moving from washington to Baltimore). The algorithm choses the maximum of these two values.
By inductive hypothesis we know that the value at $B[n]$ is maximum if we end up in Baltimore at day n and the value at $V[n]$ is maximum if we end up in Washington at day n so because for day $n+1$ we choose the maximum of the two values described above, the value at $B[n+1]$ is maximum if we end up in Baltimore on day $n+1$, regardless of the schedule for the previous $n-1$ days.

(2.) WLOG the same argument can be made about the fact that $W[n+1]$ holds the profit for an optimal schedule if on day $n+1$ we end up in baltimore

(3.) Therefore because we have maximal values for both $B[n+1]$ and $W[n+1]$, it must be that the maximum of these two values gives us the profit for an optimal schedule at day $n+1$ so the algorithm will return the profit for an optimal schedule up to the required day. (n in the problem description). ∎

Proof of running time.

It is clear that steps 1, 2, 3, 8 run in $O(1)$ and run just once
The interesting time complexity is in the loop at step 4.
lets analyze what happens in this loop. Step 5. is an assignment to an array index between the max between two sums. The sums involve array values that we have previously calculated. All of these operations

run in O(1) time so step 5 runs in O(1) time. By the same logic step 6 runs in O(1) time. And trivially 7 runs in O(1) time.

Now how many times do we run the loop, n-1 times. So because each loop iteration costs O(1) and the loop runs n-1 times, the total cost of the loop is O(n) time.

Therefore, the running time of the algorith to calculate the profit of an optimal schedule for n days is O(n).

Worked w/
Emily Wagner
Corby Rosset