CMSC 27230: Problem Set 4

Owen Fahey

February 14, 2023

Problem 1

a)

	Node								
k	a	b	t	c	d	e	s	f	g
0	∞	∞	∞	∞	∞	∞	0	∞	∞
1	∞	∞	∞	8	7	∞	0	5	∞
2	14	4	6	8	7	8	0	4	3
3	14	2	6	8	7	7	0	4	2
4	14	2	5	8	6	7	0	4	2
5	14	2	5	8	6	7	0	3	2
6	14	2	5	8	6	6	0	3	1
7	14	2	4	8	6	6	0	3	1
8	14	2	4	8	6	6	0	3	1

There shortest path of length 4 is as follows:

$$s \to c \to a \to b \to d \to f \to e \to t$$

b)

If this were to happen, then $a \to b \to d \to c \to a$ would become a negative cycle since departing from a and traveling this cycle would cost $-12+4+1+6=-1$. Hence, infinitely negative walks from s to t become possible and we could not run the algorithm on this graph.

Problem 2

Algorithm:

Setup: Suppose we have some string $Y = y_1y_2...y_n$. As will be proven, $Opt(k)$ returns total quality of the optimal segmentation of $y_1y_2...y_k$

1. Subproblems: What is $Opt(k)$ for $k \in [n]$
2. Recurrence relationship:

$$Opt(k) = max\{Opt(j-1) + q(y_j...y_k)\} \text{ where } j \leq k$$

Solve the subproblems in ascending order. For each $Opt(k)$, record the sequence that produces the optimal segmentation. To do this, for every $Opt(k)$, append the j that produces $max\{Opt(j-1) + q(y_j...y_k)\}$ to the sequence recorded for $Opt(j-1)$. Return the sequence for $Opt(n)$.

Proof of Correctness:

Key fact: If $i_1i_2...i_m$ is the sequence that optimally segments Y, then $i_1..i_{m-1}$ is the sequence that optimally segments $Y' = y_1y_2...y_{i_m-1}$ (i.e. Y with $y_{i_m}...y_n$ removed). To restate this in words, if a string has been optimally segmented and the last segment is removed, the sequence that gives the optimal segmentation of the new string is the same as that of the old string except the last index is removed.

Reasoning: If this was not the case, we could take the sequence that optimally segments Y' and add y_{i_m} to arrive at a better segmentation for the Y. This is true because the maximum total quality of the segments when Y is optimally segmented is equal to:

$$\sum_{j=1}^{m-1} q(y_{i_j}, ...y_{i_{j+1}-1}) + q(y_{i_m}, ...y_{i_{m+1}-1})$$

If the first part of the sum was not equal to the maximum total quality of the segments given by the optimal segmentation of Y', then the whole sum would not maximize the total quality of the segments of Y.

Claim: $Opt(k)$ returns the total quality of the optimal segmentation (i.e. maximum total quality of any segmentation of $y_1y_2...y_k$)

Proof: By induction.

Base case: $k = 1$. This is trivially true since there is only one segmentation of any one letter word.

Hypothesis: Assume $Opt(k)$ returns the maximum total quality of any segmentation of $y_1y_2...y_k$

Inductive step: $k = k + 1$. Suppose the last segment of the optimal solution starts at $z \leq k$. Then by the inductive hypothesis, $Opt(z-1)$ returns the maximum total quality of any segmentation for $y_1y_2...y_{z-1}$. We know from our key fact that the maximum total quality of any segmentation for $y_1y_2...y_k$ must be equal to $Opt(z-1) + q(y_z...y_k)$. Hence, if we knew the last segment of the optimal segmentation of $y_1y_2...y_k$, we could find the optimal segmentation of $y_1y_2...y_k$. Our recurrence relation checks every possible last segment, $y_j..y_k$, where $j \leq k$, and the picks the one

that maximizes $Opt(j-1) + q(y_j...y_k)$. Thus, we are guaranteed to find $j = z$ and to return the maximum total quality of any segmentation for $y_1 y_2 ... y_k$.

Runtime analysis: For each subproblem, the algorithm must check all possible last segments and perform some constant amount of overhead. Since we know that k last segments must be checked for the k^{th} subproblems, we can conclude that the total number of last segments that must be checked is: $\sum_{j=0}^{n-1} j = \frac{1}{2}(n-1)n \rightarrow O(n^2)$. Hence, this is the runtime of the algorithm.

Problem 3

Algorithm:

Preprocessing: First, sort the times $\{a_i : i \in [n]\} \cup \{b_i : i \in [n]\}$ and let $(T_1, ..., T_{2n})$ be the sorted times from smallest to largest. If the elements in $\{a_i : i \in [n]\} \cup \{b_i : i \in [n]\}$ are not distinct, break ties in the following way:

1. For all $i, j \in [n]$, if $a_i = b_j$ then put a_i before b_j.
2. For all $i < j \in [n]$, if $a_i = a_j$ then put a_i before a_j.
3. For all $i < j \in [n]$, if $b_i = b_j$ then put b_i before b_j.

Subproblems: For each $T \in \{a_i : i \in [n]\} \cup \{b_i : i \in [n]\}$, what is the maximum revenue which can be attained by time T?

Definition: Let $S_T = \{S = (i_1, ..., i_m) : S \text{ is a valid schedule and } b_{i_m} \leq T\}$. In others words, let S_T be the set of all valid schedules which are completed by time T.

Let $p(T)$ be the maximum total revenue that can be attained by time T. We can write an equation for $p(T)$ as follows:

$$p(T) = \max_{S=(i_1,...,i_k) \in S_T} \{\sum_{j=1}^{k} p_{i_j}\}$$

Recurrence Relationship: Given $\{a_i : i \in [n]\} \cup \{b_i : i \in [n]\}$, for all $r \in [2n-1]$,

1. If $T_{r+1} = a_i$ for some $i \in [n]$, then $p(T_{r+1}) = p(T_r)$
2. If $T_{r+1} = b_j$ for some $j \in [n]$, then $p(T_{r+1}) = \max\{p(T_r), p(b_{j'}) + p_j\}$ where the last time in the schedule returned by $p(b_{j'})$ is $b_{j'}$ and $p(b_{j'}) = \max\{p(b_z)\}$ with $b_z + t_{zj} \leq a_j$ (and $z, j' \in [n]$)

Solution: Solve the subproblems in ascending order of T keeping tracking of the schedule that solves each problem. Return the schedules that solves $p(T_{2n})$

Proof of Correctness:

Approach: I will make an argument that establishes an upper and lower bound on $P(T_{r+1})$.

If $T_{r+1} = a_i$ for some $i \in [n]$, then for any schedule which finishes by time T_{r+1} also finishes by time T_r so $p(T_{r+1}) = p(T_r)$. If $T_{r+1} = b_j$, then we can show that $p(T_r + 1) = \max\{p(T_r), p(b_{j'}) + p_j\}$ by showing that:

i) $p(T_{r+1}) \geq \max\{p(T_r), p(b_{j'}) + p_j\}$

ii) $p(T_{r+1}) \leq \max\{p(T_r), p(b_{j'}) + p_j\}$

To show i), observe that for any time T_r, there is a schedule S that finishes by time T_r with total revenue $p(T_r)$. This schedules finishes by time T_{r+1} so we have $p(T_{r+1}) \geq p(T_r)$.

There also must exist a schedule that finishes before a_j to which we could add job j (even if that schedule is empty). However, this schedule is not necessarily given by $p(a_j)$ like it is in the version

of the weighted interval schedule we did in class. This is because there could be some $b_{j''}$ that ends before a_j but $b_{j''} + t_{j''j} > a_j$. In words, we want whatever the best schedule that ends before a_j to which we could add job j. This is given by $p(b_{j'})$ where the last time in the schedule given by $p(b_{j'})$ is $b_{j'}$ and $p(b_{j'}) = \max\{p(b_z)\}$ with $b_z + t_{zj} \leq a_j$. Adding j to this schedule gives $p(b_{j'}) + p_j$. This schedule will end at $T_{r+1} = b_j$ and will either be equal to or less than $P(T_{r+1})$. Putting this all together we have: $p(T_{r+1}) \geq \max\{p(T_r), p(b_{j'}) + p_j\}$.

To show that $p(T_{r+1}) \leq \max\{P(T_r), p(b_{j'}) + p_j\}$, observe that by definition of $p(T_{r+1})$, there is a schedule S' which has revenue $p(T_{r+1})$ and finishes by T_{r+1}. One of the two cases must be true:

1. If S' does contain job j then S' finishes by time T_r and thus has at most revenue $p(T_r)$ meaning $p(T_{r+1}) \leq p(T_r)$.
2. If S' contains j then S' consists of a schedule S which finishes by time $b_{j'}$. Hence, S' has at most revenue $p(b_{j'}) + p_j$, so $p(T_{r+1}) \leq p(b_{j'}) + pj$.

In both cases, $p(T_{r+1}) \leq \max\{p(T_r), p(b'_j) + p_j\}$, as needed. Hence, we have shown both parts of the recurrence relationship.

Runtime Analysis:

The time complexity of this algorithm is $O(n^2)$.

First, all the times that end up in T need to be sorted. This requires $2n \cdot log(2n) \rightarrow O(nlogn)$.

There are $2n$ subproblems and each subproblem takes at most $O(n)$ since each subproblem requires checking all previous subproblems to find max $\{p(b_{j'})\}$ with $b_z + t_{zj} \leq a_j$ along with some overhead that is constant.

$2n \cdot (O(n) + c) \rightarrow O(n^2)$.

Putting this all together, we have $O(n^2)$.

Problem 4

Algorithm:

1. Subproblems: Let x, y, z represent the number of players chosen from A, B, and C respectively. What is $P[x][y][z]$ for all combinations $x, y, z \in \{0, ..., n\}$?
2. Recurrence relationship:

   $P[x][y][z] = max\{P[x-1][y][z]+s(a_{x+y+z}), P[x][y-1][z]+s(b_{x+y+z}), P[x][y][z-1]+s(c_{x+y+z})\}$

   Special cases:

   If $x = y = z = 0$, return 0

   If $x = 0$, ignore the option where $P[x-1][y][z]$...

   If $y = 0$, ignore the option where $P[x][y-1][z]$...

   If $y = 0$, ignore the option where $P[x][y][z-1]$...

   In case of tie, pick a over b and/or c AND b over c (ties do not matter)

Solution: For each subproblem, record the sequence of choices previously made and append the chosen a_{x+y+z}, b_{x+y+z} or c_{x+y+z}. Solve subproblems as follows:

1. Let $k = 1$.
2. Solve $P[x][y][z]$ for all combinations of $x, y, z \in \{0, ..., k\}$ in ascending order of $x + y + z$
3. $k = k + 1$
4. If $k \leq n$, go back to 2
5. Else, terminate

Return the sequence recorded by $P[n][n][n]$

Correctness Discussion:

Each subproblem, $P[x][y][z]$ where $x + y + z = k > 0$, can be thought of as an answer to the following question: what is the best way to pick x people from A, y people from B, and z people from C in k rounds?

To answer this question, we consider picking a_k, b_k or c_k in round k. Hence, we need to know the best way to pick from rounds 1 though $k - 1$ so that after our k^{th} round pick, x, y, z people will have been taken from A, B, C respectively.

We know the following information:

1. If we pick a_k, then in rounds 1 through $k - 1$, $x - 1$ people were picked from A, y people were picked from B, and z people were picked from C
2. If we pick b_k, then in rounds 1 through $k - 1$, x people were picked from A, $y - 1$ people were picked from B, and z people were picked from C

3. If we pick c_k, then in rounds 1 through $k-1$, x people were picked from A, y people were picked from B, and $z-1$ people were picked from C

Therefore, we can find $P[x][y][z]$ by considering $P[x-1][y][z]+a_k, P[x][y-1][z]+b_k$, and $P[x][y][z-1]+c_k$ and taking the maximum.

Runtime analysis:

When the algorithm finishes, one subproblem for each combinations of $x, y, z = \{1, ..., n\}$ will have been solved. In other words, $n \cdot n \cdot n = n^3$ subproblems will have been solved. The runtime of each subproblem will be some constant. Hence the algorithm is $O(n^3)$.