Assignment 2

Shruti Shivakumar (sshivakumar9)

October 1, 2017

1 Scheduling with Weights

Given time t_i and weight w_i for each email i, we need to compute

$$\min X = \sum_{i=1}^{n} w_i C_i$$

.

1.1 Greedy by smallest time t_i first

This criteria does not produce the optimal schedule of emails. Consider two emails – email 1 with $t_1 = 1$ and $w_1 = 1$ and email 2 with $t_2 = 10$ and $w_2 = 100$. By the smallest time first criteria, we would get $X = t_1 w_1 + (t_1 + t_2) w_2 = 1101$. However, if we swap the order of processing the emails, we would get $X = t_2 w_2 + (t_2 + t_1) w_1 = 1011$. Thus, this criteria is incorrect.

1.2 Greedy by largest weight w_i first

This criteria does not produce the optimal schedule of emails. Consider two emails – email 1 with $t_1 = 3$ and $w_1 = 2$ and email 2 with $t_2 = 1$ and $w_2 = 1$. By the largest weight first criteria, we would get $X = t_1 w_1 + (t_1 + t_2) w_2 = 10$. However, if we swap the order of processing the emails, we would get $X = t_2 w_2 + (t_2 + t_1) w_1 = 9$. Thus, this criteria is incorrect.

1.3 Greedy by largest weight-per-unit-time $\frac{w_i}{t_i}$ first

This greedy algorithm will produce the optimal solution. Note that a problem instance in this case is picking the emails in an order that would minimise X. Consider a problem instance I and let email j be the one with the largest ratio of weight to processing time. We need to show that there exists an optimal scheduling order solution S to I that has value X and includes the greedy choice j before any other email. Let S' be any optimal scheduling order to I whose value is X'. Let email i be the one with the shortest completion time in S'. Thus, S' includes j at later completion time. Now, $\frac{w_i}{t_i} \leq \frac{w_j}{t_j}$ and $C_j > C_i$.

Construct a solution S by swapping email i and j in the scheduling order. Now,

$$\begin{aligned} X &= X' - \frac{w_i C_i}{t_i} - \frac{w_j C_j}{t_j} + \frac{w_j C_i}{t_j} + \frac{w_i C_j}{t_i} \\ &= X' - (C_j - C_i)\Big(\frac{w_j}{t_j} - \frac{w_i}{t_i}\Big) \\ &\leq X' \end{aligned}$$

Thus, since greedy solution S performs no worse than S', it is just as good as any optimal solution and is hence optimal in itself.

2 Divide and Conquer

2.1 Maximum value contiguous subarray

Given the values in the array, the maximum sum of contiguous elements is 32, with starting index being 4 and ending index equal to 7.

2.2 Algorithm

We divide the array T into two parts recursively and check if the maximum value subarray lies in the left subarray or the right subarray or is present across the two subarrays.

Maximum-value-subarray(T, low, high)

1. if high==low, then return (low, high, T[low])
2. else
   (a) mid = floor((high + low)/2)
   (b) (left-low, left-high, left-sum) = Maximum-value-subarray(T, low, mid)
   (c) (right-low, right-high, right-sum) = Maximum-value-subarray(T, mid, high)
   (d) (across-low, across-high, across-sum) = Maximum-value-across-subarray(T, low, high, mid)
   (e) if left-sum>right-sum and left-sum>across-sum then return (left-low, left-high, left-sum)
   (f) else if right-sum>left-sum and right-sum>across-sum then return (right-low, right-high, right-sum)
   (g) else return (across-low, across-high, across-sum)

Maximum-value-across-subarray(T, low, high, mid)

1. Set left-sum = 0
2. Loop from T[mid] down to T[low] and keep adding element to left-sum if it increases left-sum. Also keep track of the leftmost element added
3. Repeat the above two steps for right sum by looping from T[mid] to T[high] and keeping track of the rightmost element added
4. return (leftmost-element, rightmost-element, left-sum + right-sum)

2.3 Linear-time algorithm

We can get a linear-time algorithm using dynamic programming. We need to find two indices i and j such that the sum $\sum_{k=i}^{j} T[k]$ across this window is maximum. Let $M(i)$ be the maximum value subarray possible across all windows ending in i. At i, we can either extend the maximum value subarray ending at $i-1$ or we can start a new maximum value subarray at i depending on which has the larger value. Thus, we can write the recursion

$$M(i) = max(M(i-1) + T[i], T[i])$$

The maximum value subarray has a value $M = \max\limits_{1 \leq i \leq n} M(i)$.
This algorithm has a time-complexity of $\mathcal{O}(n)$ since there are n subproblems and each subproblem takes $\mathcal{O}(1)$ time to compute.

3 Master Theorem

The master theorem states that if $T(n)$ is a monotonically increasing function that satisfies $T(n) = aT(n/b) + f(n)$ and $T(1) = c$ and $f(n) = \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & a < b^d \\ \Theta(n^d \log n) & a = b^d \\ \Theta(n^{\log_b a}) & a > b^d \end{cases}$$

Thus, according to master theorem,

1. $T(n) = \Theta(n^2 \log n)$
2. $T(n) = \Theta(n)$
3. Master theorem does not apply to this case since $T(n)$ is not monotonically increasing.
4. $T(n) = \Theta(n^{0.6})$
5. $T(n) = \Theta(n^{\log_2 3})$

4 Dynamic Programming

Given string $y = y_1 y_2 \dots y_n$, we need to compute a segmentation that maximises total plausibility of y. There are a total of 2^{n-1} ways to segment y. Let $MP(1, i) =$ maximum total plausibility of string $y_1 y_2 \dots y_i$. Thus, we can write the recursion

$$MP(1,i) = \begin{cases} \underset{1 \leq j \leq i}{max} \left\{ MP(1,j) + plausibility(y_{j+1} y_{j+2} \dots y_i) \right\} & i \neq 0 \\ 0 & i = 0 \end{cases}$$

4.1 Optimal Substructure Property

Note that the problem instance here is to find the maximum total plausibility, $MP(1,n)$ of the string $y_1y_2y_3\ldots y_n$. Let $P(j,k) = plausibility(y_j\ldots y_k)$. Thus, if an optimal solution segments the string into k substrings, we have

$$n = i_1 + i_2 + \ldots + i_k$$
$$MP(1,n) = P(1,i_1) + P(i_1+1, i_1+i_2) + \ldots + P(i_1+i_2+\ldots+i_{k-1}+1, i_1+i_2+\ldots+i_k)$$

Thus, for the first segmenting cut to the string, we get

$$\begin{aligned} MP(1,n) = max(&MP(1,n-1) + MP(n,n), MP(1,n-2) + MP(n-1,n), \\ &\ldots,\ MP(1,1) + MP(2,n),\ MP(1,0) + MP(1,n) \end{aligned} \quad (1)$$

Here, the last argument corresponds to not segmenting the string at all. The previous $i-1$ arguments correspond maximum plausibility obtained by segmenting the string into two parts of length j and $i-j$ for $j = 1,\ldots,i-1$. Now, we need to further optimally segment the two substrings further. Since the optimal solutions is written entirely in terms of optimal sub-solutions, the problem shows optimal substructure property. Note that for any argument $MP(1,j)+MP(j+1,n)$ in equation 1, we can write $MP(j+1,n) = P(j+1,n)$. If it were not so, we could segment $y_{j+1}y_{j+2}\ldots y_n$ further at some point k such that $MP(1,n)$ will have a higher total plausibility than before segmenting at k. However, this corresponds to the subproblem of $MP(1,k) + MP(k+1,n)$. Thus, we can write $MP(j+1,n) = P(j+1,n)$. Also, we can drop the first index of $MP(1,n)$ since it does not affect equation 1.

4.2 Recursive Expression

Let $MP(n)$ = maximum total plausibility of string $y_1y_2\ldots y_n$. Let $P(j,k) = plausibility(y_j\ldots y_k)$. Thus, we can write the recursion

$$MP(n) = \begin{cases} \max\limits_{1\le j\le n} \left\{MP(j) + P(j+1,n)\right\} & n \neq 0 \\ 0 & n = 0 \end{cases}$$

4.3 Recursive Algorithm

Memoized-Recursive-Segmentation(n)

1. let MP[0..n] be a new array with all elements assigned to $-\infty$
2. return **Memoized-Recursive-Segmentation-Helper(n, MP)**

Memoized-Recursive-Segmentation-Helper(n, MP)

1. if $MP[n] \geq 0$ return $MP[n]$
2. if $n = 0$, then $mp = 0$, else set $mp = -\infty$
3. if $n \neq 0$, then use j to loop through the list of possible indices i.e. from $n-1$ to 0 and set $mp = max(mp,$ **Memoized-Recursive-Segmentation-Helper(j, MP)**$+$ $P(j+1,n)$ at each iteration of the loop

4. set $MP[n] = mp$
5. return mp

The time complexity of this algorithm is $\mathcal{O}(n^2)$ since for each problem $MP(n)$, we need check all the subproblems from $MP(1)$ to $MP(n-1)$ and each subproblem takes $\mathcal{O}(1)$ time. The space complexity of the algorithm is $\mathcal{O}(n)$ since we need to initialize the MP array.

4.4 Bottom-Up Approach

Since for each problem $MP(n)$, we need to solve all subproblems from $MP(1)$ to $MP(n-1)$, the bottom-up algorithm computes these subproblems before computing $MP(n)$.

Bottom-Up-Segmentation(n)

1. let MP[0..n] be a new array with all elements assigned to $-\infty$
2. Let $MP[0] = 0$
3. for i = 1 to n
   (a) set $mp = -\infty$
   (b) for j = i-1 to 0
      i. $mp = max(mp, MP[j] + P(j+1, i)$
   (c) MP[i] = mp
4. return MP