Assignment 2

Shruti Shivakumar (sshivakumar9)

September 22, 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 next email j 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, $\frac{w_i}{t_i} < \frac{w_j}{t_j}$ and $C_j > C_i$. Construct a solution S by swapping email i and j in the scheduling order. Now

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 $M = \max_{1 \le i \le n} M(i)$.
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 $MP(i)$ = maximum total plausibility of string $y_1 y_2 \dots y_i$. Thus, we can write the recursion

$$MP(i) = \max_{1 \le j < i} \{MP(j) + plausibility(y_{j+1} y_{j+2} \dots y_i)\}$$