CS345: Assignment 2

Pragati Agrawal (220779)

September 2024

Pragati Agrawal: 220779 (Question 1)

Description of the Algorithm:

We would sort the jobs in non-decreasing order of their $\frac{t_i}{w_i}$. Then we return the jobs' order as the order in which they appear after sorting.

Proof of correctness for the Algorithm:

We prove by contradiction that the sorted ordering is an optimal ordering:
In any unsorted permutation of the jobs, we can say by the correctness of the idea of **Bubble sort algorithm**, that we can always find two consecutive jobs i and j, s.t. $\frac{t_i}{w_i} > \frac{t_j}{w_j}$. Let their times to complete, weights and finish times be t_i, w_i, C_i and t_j, w_j, C_j respectively. Let us call P as the value $P = \sum_{i=1}^{n} w_i C_i$. Let the total time spent upto the job completed just before jobs i and j be T_l. Also, let these jobs have value P_l. Similarly, for the jobs to their right, we have values T_r as the total time spent from beginning to complete these jobs, and P_r be their P value (P_r only for jobs occurring after the jobs i and j). We must notice here that on swapping the order of jobs i and j, the completion times of all the jobs to their left and right remained unaffected.

$$i \textbf{ occurs before } j \rightarrow P_1 = P_l + w_i * (T + t_i) + w_j * (T + t_i + t_j) + P_r$$

We prove that if we swap the order of the jobs i and j, P must decrease:

$$j \textbf{ occurs before } i \rightarrow P_2 = P_l + w_j * (T + t_j) + w_i * (T + t_i + t_j) + P_r$$

$$P1 - P2 = P_l + w_i * (T + t_i) + w_j * (T + t_i + t_j) + P_r - P_l + w_j * (T + t_j) + w_i * (T + t_i + t_j) + P_r$$

$$= w_j * t_i - w_i * t_j > 0 \qquad (since \frac{t_i}{w_i} > \frac{t_j}{w_j})$$

So we have proved that any unsorted permutation cannot be optimal. Now if there are multiple sorted permutations, all will give the same value of P and it would be the minimum. This is because all jobs having the same ratio of $\frac{t_i}{w_i}$ would be contiguous in the sorted permutation, and swapping these internally would keep the cost the same.

Time Complexity: We just need to sort, which takes $O(nlogn)$ time.

Pragati Agrawal: 220779 (Question 2)

We are given a directed acyclic graph $G = (V, E)$ in which each node $u \in V$ has an associated price, denoted by $price(u)$, which is a positive integer. The cost of a node u, denoted by $cost(u)$,is defined to be the price of the cheapest node reachable from u (including u itself). We wish to design an efficient algorithm that computes $cost(u)$, $\forall u \in V$.

Description of the Algorithm:

Since we are given a directed acyclic graph, we would use topological sorting technique to solve this question. To find the $cost(u)$ for any node, we need to process all nodes that are reachable from it. So for any node u, if we have the values $cost(v)$ for all nodes v s.t. $(u, v) \in E$ (let's call these children of u), then the $cost(u)$ would simply be the minimum among them. Topological ordering ensures that for any u, all its children and further descendants lie to its right. So we will process the nodes from **right-to-left** direction in the topological order. This guarantees that while computing $cost(u)$, we have the values of all children of u, and hence we can find the value of $cost(u)$.

Pseudo-code for the Algorithm:

$find\_min\_cost(G)\{$
    $\mathbf{T} \leftarrow toposort(G)$;
    $reverse(\mathbf{T})$;
    $for(i = 1\ to\ n)$ $\mathbf{cost[i]} = \infty$;
    $for(u\ in\ \mathbf{T})$ {
        $\mathbf{cost[u]} = \mathbf{price[u]}$;
        $for(\mathbf{w}\ in\ adj\_list(u))$
            $\mathbf{cost[u] = min(cost[u], cost[w]);}$
    };
    return **cost**;
};

Time Complexity Analysis:

As taught in lectures, topological sorting takes $O(m + n)$ time.
The first *for* loop iterates n times.
In the second *for* loop, we would be visiting each node exactly once in the topological ordering. And for each node u, we would be visiting all its children and perform some constant time operations. Therefore, for each node u, we take $O(outdegree(u))$ time. Summing this up for all nodes would be equal to $O(m)$, where m is the total number of edges (each edge would be visited exactly once).
Therefore, the overall time complexity is $O(m + n)$.