CS345: Design and Analysis of Algorithms

Assignment 2

Submitted by: Aditi Khandelia, 220061

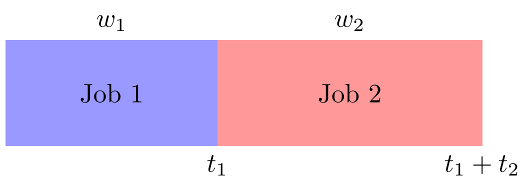
Question 1.
A photocopy shop has a single large machine. Each morning the shop receives a set of jobs from customers. The shopkeeper wants to do the jobs on the single photocopying machine in an order that keeps their customers happiest. Customer $i's$ job will take t_i time to complete. Given a schedule (ordering of the jobs), let C_i denote the finishing time of job i. For example, if job i is the first to be done, we would have $C_i = t_i$; and if job j is done right after job i, we would have $C_j = C_i + t_j$. Each customer has a given weight w_i that represents his or her importance to the business. The happiness of customer i is expected to be dependent on the finishing time of $i's$ job. So the company decides that they want to order the jobs to minimize the weighted sum of the completion time, $\sum_1^{i=n} w_i C_i$ Design an efficient algorithm to solve this problem. That is, you are given a set of n jobs: job i has a processing time t_i and a weight w_i. You want to order the jobs so as to minimize the weighted sum of the completion time, $\sum_1^{i=n} w_i C_i$.

Answer.
This problem can be solved by undertaking a greedy approach.

Idea
Execute jobs in non-increasing order of $\frac{w_i}{t_i}$.

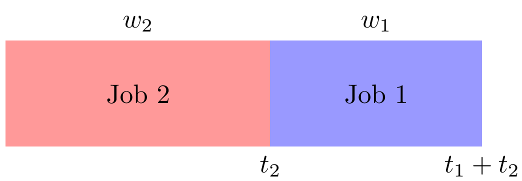
Proof of Correctness
To get the intuition, first consider the case of two jobs. The first job has processing time t_1 and weight w_1, while the second job has processing time t_2 and weight w_2.

Case 1. Job 1 is executed first



Here, the weighted sum of completion times is : $w_1 * t_1 + w_2 * (t_1 + t_2)$

Case 2. Job 2 is executed first



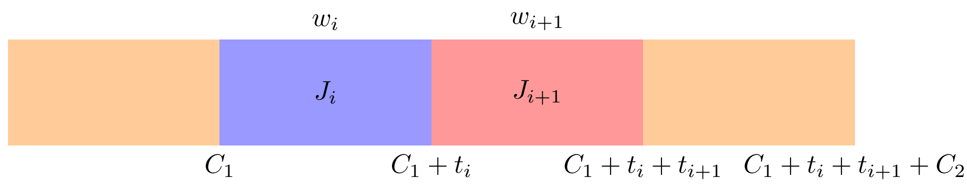
Here, the weighted sum of completion times is : $w_2 * t_2 + w_1 * (t_1 + t_2)$

Now, we wish to minimize the weighted sum of completion times. WLOG, assume that the first case is optimal :

$$w_1 * t_1 + w_2 * (t_1 + t_2) <= w_2 * t_2 + w_1 * (t_1 + t_2)$$
$$w_2 * t_1 <= w_1 * t_2$$
$$\frac{w_2}{t_2} <= \frac{w_1}{t_1}$$

It can be seen in the 2-job case that the optimal solution has $\frac{w_i}{t_i}$ in non-increasing order.

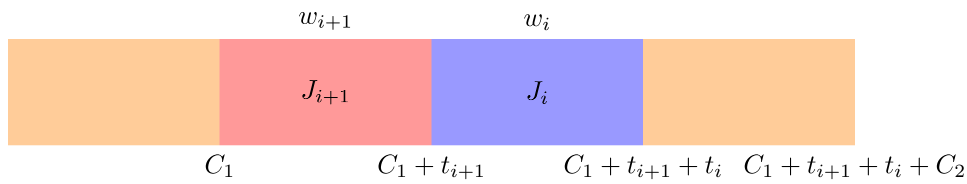
Proof by contradiction for the general case :
Let the optimal solution be the sequence A and WLOG, assume $A = J_1, J_2, ..., J_n$
Consider any i and its adjacent job $i + 1$.
Suppose the following is the representation of the job scheduling :



In this case, the weighted sum of completion time T_1 will be :

$$(w_1 * t_1 + w_2 * (t_1 + t_2) + \cdots + w_{i-1} * (t_1 + t_2 + \cdots + t_{i-1}))+$$
$$(w_i * (C_1 + t_i))+$$
$$(w_{i+1} * (C_1 + t_i + t_{i+1}))+$$
$$(w_{i+2} * (C_1 + t_i + t_{i+1} + t_{i+2}) + w_{i+3} * (C_1 + t_i + t_{i+1} + t_{i+2} + t_{i+3}) + \cdots + w_n * (C_1 + C_2 + t_i + t_{i+1}))$$

Now suppose we swap the jobs i and $i + 1$.
Suppose the following is the representation of the job scheduling :



In this case, the weighted sum of completion time T_2 will be :

$$(w_1 * t_1 + w_2 * (t_1 + t_2) + \cdots + w_{i-1} * (t_1 + t_2 + \cdots + t_{i-1}))+$$
$$(w_{i+1} * (C_1 + t_{i+1}))+$$
$$(w_i * (C_1 + t_{i+1} + t_i))+$$
$$(w_{i+2} * (C_1 + t_i + t_{i+1} + t_{i+2}) + w_{i+3} * (C_1 + t_i + t_{i+1} + t_{i+2} + t_{i+3}) + \cdots + w_n * (C_1 + C_2 + t_i + t_{i+1}))$$

$$\text{Now, } T_1 - T_2 = w_{i+1} * t_i - w_i * t_{i+1}$$

There is no increment in the weighted sum of completion times on swapping jobs J_i and J_{i+1} when $\frac{w_{i+1}}{t_{i+1}} >= \frac{w_i}{t_i}$.
Thus, we can keep swapping two adjacent jobs whenever there is an inversion w.r.t. $\frac{weight}{processingTime}$.We would continue this process till there are no inversions anywhere in the array. This would lead to an optimal solution since -

1. The optimal solution is supposed to have jobs in non-increasing order of $\frac{weight}{processingTime}$ since otherwise, there will be at least one pair of adjacent jobs J_i, J_{i+1} such that $\frac{w_{i+1}}{t_{i+1}} > \frac{w_i}{t_i}$. In this case, the quantity $T_1 - T_2$ as calculated above would be positive, and thus swapping the two jobs would lead to a better solution.

2. If the sequence S_1 of jobs $J_i, J_{i+1}, J_{i+2}, ..., J_j$ have the same value of $\frac{weight}{processingTime}$, their relative order of execution doesn't matter :
Consider any job J_{k_1} in the sequence S_1. Suppose we wish to move it to a position k_2 from k_1 in S_1 itself. WLOG, assume $k_2 < k_1$. When we swap the jobs J_{k_1} and J_{k_1-1}, the quantity $T_1 - T_2 = 0$. Thus, swapping doesn't change the value at optimal position. We can hence keep pushing the job J_{k_1} towards the left till we attain the desired position k_2.
Thus, we can achieve any permutation of the jobs $J_i, J_{i+1}, ..., J_j$ without altering the value of weighted sum at optimal solution.

Algorithm.

1. Receive the array of weights w
2. Receive the array of processing times t
3. Create the array of $\frac{weight}{processingTime}$
4. $\forall i$ do:
   (a) if $t_i = 0$, add ∞ to the array
   (b) if $t_i \neq 0$, add $\frac{w_i}{t_i}$ to the array
5. Sort the array in non-increasing order
6. Output the jobs according to the order of the array

Pseudocode.

Algorithm 1 Weighted Completion Time Algorithm

```
Require: Array of weights w, Array of processing times t
Ensure: Ordered list of jobs
Receive the array of weights w
Receive the array of processing times t
Create an empty array ratio
for i = 1 to length(w) do
    if t[i] = 0 then
        ratio[i] ← ∞
    else
        ratio[i] ← w[i]/t[i]
Sort ratio in non-increasing order, keeping track of original indices
return Jobs ordered according to sorted ratio indices
```

Time Complexity Analysis.
Following is the time complexity analysis of the algorithm:

1. Iteration through the array of weights and processing times, and creation of the array *ratio* take $O(n)$ time.
2. Sorting takes $O(nlogn)$ time.
3. Creation of return array according to the sorted array *ratio*, takes $O(n)$ time.

Total time complexity :

$$T(n) = O(n) + O(nlogn) + O(n) = O(nlogn)$$

Question 2.

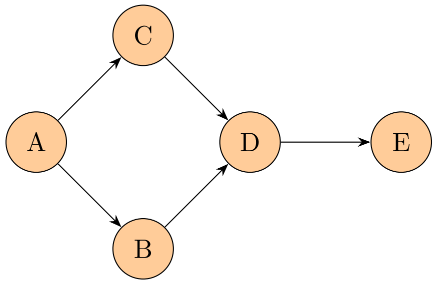
You are given a directed acyclic graph G = (V, E) in which each node u ∈ V has an associated price, denoted by price(u), which is a positive integer. The cost of a node u, denoted by cost(u),is defined to be the price of the cheapest node reachable from u (including u itself). Design an algorithm that computes cost(u) for all u ∈ V .

Answer.

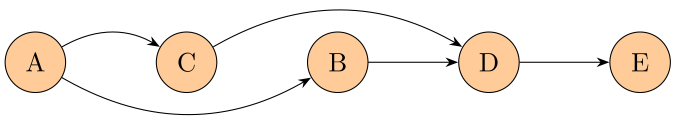
The given problem can be solved as an application of topological sort.

Approach.

Since the given graph is a directed acyclic graph, a topological sort will always exist for it.
Suppose the graph is :



The topological sorting for this graph will be :



For any vertex v of the graph, the *cost(v)* will be min(min(*cost(u)*, $(v, u) \in E$), *price*(v)).
$cost(A) = \min(cost(C), cost(D), price(A))$, $cost(B) = \min(cost(D), price(B))$, $cost(D)$ =min$(price(D))$
This can be proven as follows :

1. Proof that the node u such that $cost(v) = price(u)$ is reachable from the node v.

   Proof by induction on the position of node v:
   Base case: v lies in the rightmost position.
   In this case, v will not have any edges to any other nodes in the graph, implying that $cost(v) = price(v)$ and since every node is reachable from itself, the base case holds true.
   Induction hypothesis :
   Suppose that for any node v in position $\leq n$ from the right end, the hypothesis holds true.
   Induction step :
   Consider the node v at position $n + 1$ from the right end.
   It will only have edges to nodes u such that $position(u) \leq n$.
   Suppose min(min(*cost(u)*, $(v, u) \in E$), *price*(v)) is *price*(v). Since v is reachable from itself, induction step holds true.
   Suppose min(min(*cost(u)*, $(v, u) \in E$), *price*(v)) $\neq$ *price*(v). Thus, $\exists u$ such that $cost(v) = cost(u)$ and $(v, u) \in E$.
   Suppose $cost(u) = price(w)$ for some node w. Thus, $cost(v) = price(w)$. By induction step, w is reachable from u. Since there exists an edges from v to u, w is reachable from v.
   Hence, proven.

2. Proof that $\forall u$ such that u is reachable from v, $cost(v) = \min(price(u))$.

   Proof by mathematical induction on the position of node v:
   Base case : node v is at the rightmost position.
   In this case, the only node reachable from v is v itself, whose price is taken into account. Hence, the base case holds true.
   Induction hypothesis :
   Suppose that $\forall u$ such that $position(u) \leq n$, the hypothesis holds true.
   Induction step :
   Consider the node v at position $n + 1$. Consider any node u which is reachable from v. The following cases arise:

(a) $u = v$: In this case, $price(u)$ gets taken into account since $price(u) = price(v)$.

(b) $(v, u) \in E$: In this case, $cost(v)$ takes into account $cost(u)$, which according to the induction hypothesis takes into account $price(u)$ since $position(u) < position(v) = n + 1$.

(c) $(v, u) \notin E$: Since u is reachable from v, $\exists$ a sequence of vertices $v, v_1, v_2, \ldots, u$ such that every pair of adjacent vertices represents an edge in the graph.
As can be seen in the formula for cost calculation, vertex $cost(v)$ takes into account $cost(v_1)$. by induction hypothesis, $cost(v_1)$ takes into account $cost(u)$.

Hence, proven.

The above two proofs complete our proof since we have proven that the formula calculates the minimum of all the vertices reachable from node v.

Algorithm.
Following is the algorithm for the problem :

1. Obtain the graph G in the form of an adjacency list
2. Obtain the *price* array for the vertices v of the graph
3. Obtain the topological sort τ of G
4. Arrange the vertices in decreasing order of τ
5. Create an array *cost* to store the cost of each vertex
6. Traverse τ and for each vertex v corresponding to the topological sort, do:
   (a) set the cost of the vertex to be price of the vertex
   (b) for each vertex u such that $(v, u) \in E$, update the cost of v as min($cost(v)$, $cost(u)$)
7. Return the *cost* array

Pseudocode.

Algorithm 2 Minimum Cost Path in DAG

Require: Graph $G = (V, E)$ in adjacency list form, Price array *price* for vertices
Ensure: Cost array *cost* for vertices
Obtain the graph G in the form of an adjacency list
Obtain the *price* array for the vertices v of the graph
$\tau \leftarrow$ TopologicalSort(G)
Arrange the vertices in decreasing order of τ
Create an array *cost* to store the cost of each vertex
for each vertex v in τ (in reverse order) **do**
    $cost[v] \leftarrow price[v]$
    for each vertex u such that $(v, u) \in E$ **do**
        $cost[v] \leftarrow \min(cost[v], cost[u])$
return *cost* array

Algorithm 3 Topological Sort

Require: Directed graph $G = (V, E)$ represented as an adjacency list
Ensure: List L containing a topological ordering of G
$L \leftarrow$ Empty list that will contain the sorted elements
$S \leftarrow$ Set of all nodes in G with no incoming edge
$inDegree \leftarrow$ Array storing the in-degree of each node
for each node n in G **do**
    $inDegree[n] \leftarrow$ number of incoming edges to n
    if $inDegree[n] = 0$ **then**
        add n to S
while S is non-empty **do**
    remove a node n from S
    add n to tail of L
    for each node m with an edge e from n to m **do**
        $inDegree[m] \leftarrow inDegree[m] - 1$
        if $inDegree[m] = 0$ **then**
            add m to S
return L (a topologically sorted order)

Time Complexity Analysis.
Following is the time complexity analysis of the algorithm :

1. Topological sorting takes time $O(V + E)$.
2. The list τ contains V elements, and each element takes $O(degree(vertex))$ time. Thus, total time complexity for this becomes $\sum_{v \in V}(degree(v) + 1) = O(E + V)$.

Thus, final time complexity is :

$$T(V, E) = O(V + E) + O(V + E) = O(V + E)$$