CS345 Theoretical Assignment 2 Submission

Naman Kumar Jaiswal
220687
namankj22@iitk.ac.in

Fall Semester 2024

Question 1: Weighted Job Scheduling

We need to find an efficient algorithm for Job Scheduling N jobs where each job j$_i$ has some importance w$_i$ and time required t$_i$. We need to minimise the sum of cumulative time taken weighted with it's importance.

Minimisation Expression:

We need to minimise

$$S = \sum_{i=0}^{N-1} C_i w_i$$

$$C_j = \sum_{i \in Q_j} t_i$$

Where Q_j is the set of jobs done before the jth jobs, also including jth job.

Intuition:

Usual Job Scheduling problems can be solved by applying Greedy Strategy based on some function of job finishing time t$_i$ and other parameters. If we wish to find a greedy paradigm solution we need to find the function and prove the correctness of the solution.

1. The rearrangement inequality states that for every choice of real numbers $x_1 \leq \cdots \leq x_n$ and $y_1 \leq \cdots \leq y_n$, and every permutation σ of the numbers $1, 2, \ldots, n$, we have

   $$x_1 y_n + \cdots + x_n y_1 \leq x_1 y_{\sigma(1)} + \cdots + x_n y_{\sigma(n)} \leq x_1 y_1 + \cdots + x_n y_n.$$

   Informally, this means that in these types of sums, the largest sum is achieved by pairing large x values with large y values, and the smallest sum is achieved by pairing small values with large values. Hence, we should try to do the task with more w_i sooner than the ones with lower w_i.

2. Also what is a neat heuristic in Job Scheduling problems is the fact that for the same importance, if we chose to do the job that requires less time before the lengthier ones, the overhead seen in C$_i$ of the time taken by the previous jobs will be minimised. Informally, if the task k is done first, the term t$_k$ will be seen in each and every C$_j$ afterword. It makes sense in choosing the less time consuming task to be done first.

The priority is proportional to w$_i$ and inversely proportional to t$_i$. The relation may not be linear, we need to check.

Function and Claim:

Let us take a function

$$f(w_i, t_i) = \frac{t_i}{w_i}$$

and make the following claim.
Claim: The most optimal job scheduling will be based on the increasing order of f, where the job with the least f(w,t) will be completed first.

Proof:

Let us assume there exist an optimal Job Scheduling J = $\{J_1, J_2, J_3, \cdots J_n\}$, where $J_i = (W_i, T_i)$

1. **Theorem 1:** If we have an optimal Job Scheduling J where J_l has the smallest $f(W_i, T_i)$ and l $\neq$ 1, either J is not the optimal scheduling or there exist another optimal scheduling where J_l and J_{l-1} can be swapped.

Proof:
Before swapping

$$S_1 = W_1{*}T_1{+}W_2{*}(T_1{+}T_2){+}\cdots W_{l-1}{*}(T_1{+}T_2{+}\cdots T_{l-1}){+}W_l{*}(T_1{+}T_2{+}\cdots T_{l-1}{+}T_l){+}\cdots W_n{*}(T_1{+}T_2{+}\cdots T_n)$$

After swapping

$$S_2 = W_1{*}T_1{+}W_2{*}(T_1{+}T_2){+}\cdots W_l{*}(T_1{+}T_2{+}\cdots T_{l-2}{+}T_l){+}W_{l-1}{*}(T_1{+}T_2{+}\cdots T_l){+}\cdots W_n{*}(T_1{+}T_2{+}\cdots T_n)$$

Now,

$$S_1 - S_2 = W_l * T_{l-1} - W_{l-1} * T_l$$

Since,

$$\frac{T_l}{W_l} \leq \frac{T_{l-1}}{W_{l-1}}$$
$$S_1 \geq S_2$$

Hence, proved that either J is not optimal or J after swapping l and l-1 is also optimal.

2. **Theorem 2:** If we have an optimal Job Scheduling J where J_l has the smallest $f(W_i, T_i)$ and l $\neq$ 1, either J is not the optimal scheduling or there exist another optimal scheduling where J_l is the first element.
   Proof:
   To prove this we are going to apply theorem 1 recursively until J_l becomes the first element. First we will bring J_l at the (l-1) position, then at the (l-2) position and so on until it goes at the first position. The new J will be
   $$J = \{J_l, J_1, J_2, \cdots J_{l-1}, J_{l+1} \cdots J_n\}$$
   Hence proved, either J is not the optimal scheduling or there exist another optimal scheduling where J_l is the first element.

3. **Theorem 3:** If we have an optimal Job Scheduling J where J_i's are not in the increasing order of $f(W_i, T_i)$, then either J is not the optimal solution or there also exist another solution where increasing order of $f(W_i, T_i)$ is also the optimal solution.
   Proof:
   To prove this we are going to use induction on the claim that for n jobs the most optimal solution requires J_i's are going to be in the increasing order of $f(W_i, T_i)$.

   (a) **Base Case:** for n = 1, this is trivial, there only exist 1 element, and the element with the small $f(W_i, T_i)$ is at the first.

   (b) **Induction Step:** Assuming the claim to be true will n-1, we need to prove the claim to be true for n.
   Using Theorem 2 we can say that there also exists a job scheduling where the element with the smallest $f(W_i, T_i)$ is at the first. Now, for the next n-1 elements we have another sub-problem of Job Scheduling. The minimising expression for S_n and the sub-problem S_{n-1} is
   $$S_n = T_l * (\sum_{i=1}^{n} w_i) + S_{n-1}$$
   To minimise, S_n, we need to minimise S_{n-1}. For n-1 elements, we need to arrange the n-1 jobs in increasing order and the first element at the start is the minimum, hence the claim is correct.

   Hence Proved.

Pre-Processing:

We have proved that the job with the smallest f(w,t) will be completed first. Hence we need to keep the sorted array at hand and keep completing the jobs in the order of the array.

1. Make an array of tuples of length 3 containing $((f(w_i.t_i), w_i, t_i)$ for all the jobs from i = 0 to N-1.
2. Sort the array using any sorting algorithm in increasing order of $f(w_i, t_i)$. For efficient sorting, we will be using Heap Sort.
3. We will start by building the heap from the tuples using Heapify method.

Logic:

1. **Heapify**: We will build the heap using heapify method, while for each comparison we use the relation,

$$(f(w_1, t_1), w_1, t_1) > (f(w_2, t_2), w_2, t_2) \ iff \ f(w_1, t_1) > f(w_1, t_1)$$

2. **Extract Min**: We will keep extracting the minimum of the heap until the heap is empty. We will append this job to a array of tuples. This array is the most optimal job scheduling to minimise the minimisation expression.

Pseudocode:

1. We will need to sort the array of Jobs in the increasing tuple form.

```
jobArray = []
for i = 0 to N-1:
    f_value = f(jobs[i].w, jobs[i].t)
    jobArray.append((f_value, jobs[i].w, jobs[i].t))

Heap = []
function compare_less ((f(w_1, t_1), w_1, t_1), (f(w_1, t_1), w_1, t_1)):
    return f(w_1, t_1) < f(w_1, t_1)

Heap ← Heapify (JobArray, compare_less)

function heapExtractMin(H):
    minElement = H.top()

    # Balance the heap using the compare_less function as the comparator
    H ← BalanceHeap(H, compare_less)
    return minElement
```

2. Job Scheduling

```
function JobScheduling(H):

    J = []

    while H is not empty:
        minJob = heapExtractMin(H)
        J.append(minJob)

    return J
```

Question 2: Cost Computation over a Directed Acyclic Graph

We need to find an efficient algorithm that runs over a DAG G (V, E) which computes $\forall u \in G$, cost(u) which is defined to be the price of the cheapest node reachable from u (including u itself), where the price is a value associated with each graph node.

Heuristic:

1. Let us assume that v is the node with the smallest price, lets say r, somewhere in the graph, and there exist a path p from a node u to v = $\{u, u_1, u_2, \cdots u_k, v\}$, if price of v is the smallest, then cost(v) = r, cost(u_k) = r $\cdots$ cost(u_1) = r and cost(u) = r.
2. The heuristic here is that the cost value is propagated from bottom to the top. The meaning of bottom to the top here is with reference to the DFS tree from any node u to v. So, if we store the nodes from in a specific order which
   (a) tells us approximate relative positioning of the nodes in the some DFS tree of the graph
   (b) has the node with zero in-degree in the front and zero out-degree in the end (this is possible because the graph is a DAG)
   (c) if the node u comes before v, there might be a path from u to v but definitely never a path from v to u
3. The order which satisfies the above property is the topological sort of the graph G.

Logic:

1. **Topological Sort**: First we will compute the Topological Sort of the graph G using Kahn's algorithm.
2. **Cost Computation**: Let T = $\{a_0, a_1, a_2, \cdots a_{n-1}\}$ be the topological ordering of the graph G. We will start from the end of the order and go up the order, for any node u we encounter we will look for all edges (v, u) in G, and update cost[v] as min (cost[v], cost[u]).

Pseudocode:

1. Kahn's Algorithm, G = (V,E)

```
function topologicalSort(G):
    N = G.numVertices()               # number of vertices in G
    in_degree = ← Array containing in-degree of each node
    topo_order = []                   # List to store the topological sort
    queue = []                        # Queue to store vertices with in-degree
     0

    # Add vertices with in-degree 0 to the queue
    for v in V:
        if in_degree[v] = 0:         # If a vertex has no incoming edges
            queue.append(v)

    # Process vertices from the queue
    while queue is not empty:
        u = queue.pop(0)             # Remove a vertex from the queue
        topo_order.append(u)          # Add it to the topological sort

        for each neighbor v of u in G:
            in_degree[v] -= 1         # Reduce in-degree of its neighbors by 1
            if in_degree[v] = 0:
                queue.append(v)       # Add the neighbor to the queue

    return topo_order              # Return the topological sort
```

2. Function to Relax the edges

```
function Main():

    T ← topologicalSort(G)

    # We will initialise the cost array where cost[i] = price[i]
    array cost_arr ← price_arr

    while T is not empty:
        u = T.pop()
        for each (v, u) in E:
            cost_arr[v] = min(cost_arr[v], cost_arr[u])

    return cost_arr
```