IIT Kanpur

CS345A - AlgorithmsII

Assignment 3

Anjani Kumar
11101

Sumedh Masulkar
11736

February 14, 2014

Contents

1 A Job Scheduling Problem

Consider the following jobs starting from t=0:

Job	Deadline	Penalty
Job1	1	1
Job2	2	1
Job3	3	100
Job4	3	100

We have to schedule these jobs such that overall penalty is minimum.

1.1 Strategy 1:

Schedule the jobs in increasing order of deadlines.

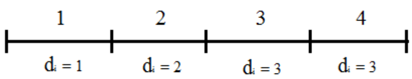


Figure 1: Job Schedule for strategy 1

- According to this strategy, the jobs would be scheduled as per fig 1.
- In this case, job 4 exceeds its deadline. Hence total penalty is $P_4 = 100$.

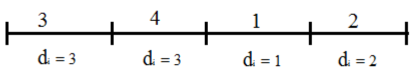


Figure 2: A counter-example for strategy 1

- In fig 2, jobs 1 and 2 exceed their deadlines. Hence total penalty for this case is $P_1 + P_2 = 2$.
- Since the penalty for fig 2 is less than that of fig 1. Therefore Strategy 1 is not optimal.

- Job schedule according to above strategy

Job	Slot	Deadline	Penalty
Job1	0-1	1	0
Job2	1-2	2	0
Job3	2-3	3	0
Job4	3-4	3	100

- A Possible Counter-example for above strategy

Job	Slot	Deadline	Penalty
Job3	0-1	3	0
Job4	1-2	3	0
Job1	2-3	1	1
Job2	3-4	2	1

1.2 Strategy 2:

Schedule the jobs in decreasing order of penalties and while scheduling a job if there is an empty slot available before its deadline then schedule it as early as possible.

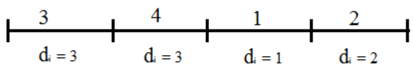


Figure 3: Job schedule for strategy 2

- According to this strategy, the jobs would be scheduled as per fig3.
- In this case, jobs 1 and 2 exceed their deadlines. Hence total penalty is $P_1 + P_2 = 2$.

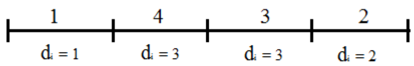


Figure 4: A counter example for strategy 2

- In fig 4, only job 2 exceeds its deadline. Hence total penalty for this case is $P_2 = 1$.
- Since the penalty for fig 4 is less than fig 3. Therefore Strategy 2 is not optimal.

- Job schedule according to above strategy

Job	Slot	Deadline	Penalty
Job3	0-1	3	0
Job4	1-2	3	0
Job1	2-3	1	1
Job2	3-4	2	1

- A Possible Counter-example for above strategy

Job	Slot	Deadline	Penalty
Job1	0-1	1	0
Job4	1-2	3	0
Job3	2-3	3	0
Job2	3-4	2	1

1.3 Strategy 3:

schedule the jobs in decreasing order of penalties and while scheduling a job if there is an empty time slot available before its deadline, then schedule it to the latest such slot.

Data Structure Used:

- Let the number of jobs be n.
- An array (Temp) of size n, which after sorting, contains the index of jobs according to non-increasing order of their penalty.
- A complete Binary search tree (T) of size n, used to store the available slots. Such that, on an in-order traversal of T, the slots are printed in an ascending order (1,2,...n).

- An array (Opt) of size n, which contains the optimal job schedule.
- An array (flag) of size n. flag[j] = 1, if the deadline of job in $Temp[j]$ has been exceeded. Otherwise it is 0.
- index of all the arrays start from 1.

Brief description:

- The jobs are taken one by one (according to decreasing order of their penalty) from the array $Temp$ (from index 1 to n). Let the job corresponding to current element i be j. The deadline corresponding to j is found. Let it be d_j.
- A node with value same as that of the d_j is searched in T. Following cases are possible:

  1. Case1: The Node is present in T.
     - The corresponding node is deleted from T.
     - The value of Opt[d_j] is updated to j
  2. Case2: The Node is absent and its predecessor is present in T.
     - Predecessor(d_j) in tree T is calculated. Let it be p_j.
     - The corresponding predecessor node is deleted.
     - The value of Opt[p_j] is updated to j.
  3. Case3: The Node is absent and its predecessor does not exist in T.
     - In this case, Predecessor(d_j) returns value -1.
     - This implies that the job's deadline has been exceeded.
     - The value of flag[i] is switched from 0 to 1.

- After the array $Temp$ is traversed completely:
  - The jobs with high flag are appended to array Opt in the end.
  - The array Opt is returned.

1.4 Functions used:

- **Sort**($Temp$): Sorts the passed array $Temp$(which contains the jobs) in non-increasing order of their penalties.
- **Search**(T, V): Searches for the node with value V in tree T and returns $TRUE$ if found, else it returns $FALSE$.
- **Predecessor**($Node$): Return the value of predecessor of $Node$ if it is present, otherwise it returns -1.

1.5 Pseudo Code

Algorithm 1: Pseudo code for Job_Scheduler(J)

```
Job_Scheduler(J){
begin
    Initialize arrays Temp, Opt, flag of size n.
    Sort(Temp);          //Sorts array of jobs in decreasing order of penalties
    T ← A complete binary tree of size n as described in section 1.3.
    found ← false;
    i ← 1;          //i keeps note of number of slots filled in Opt.
    for j ← 1 to n do
                        //O(n log(n)).
        found ← Search(T, Temp[j]→deadline);
        if found==true then
            Opt[Temp[j]→deadline] ← Temp[j];
            i + +;
            Delete(T, Temp[j]→deadline);
            found ← false;
        else
            pred ← Predecessor(Temp[j]→deadline);
            if pred<>-1 then
                Opt[pred] ← Temp[j];
                i + +;
                Delete(T, pred);
            else
                flag[j] = 1;
    for j ← 1 to n do
        if flag[j]<>0 then
            Opt[i] ← Temp[j];
            i + +;
    return Opt;
end
}
```

1.6 Time Complexity

- It takes $O(n)$ time to build the tree T.
- **Sort** takes $O(\text{n} \log n)$ time, where n is the number of jobs.
- The functions **Search**,**Delete** and **Predecessor** takes $O(\log k)$ time, where k is the size of the tree T at any given time.

- Traversing the arrays $flag$ and Opt takes $O(n)$ time.
- As shown in the Pseudo code, overall the algorithm takes $O(\mathbf{nlog}\ n)$ time.

1.7 Proof Of Correctness

Let J = $j_1, j_2, ...j_n$ be n jobs in increasing order of penalty.

Lemma 1.1

In the optimal solution $Opt(J)$, j_n must be scheduled at or before its deadline.

Proof

Proof by Contradiction:
Let us assume that lemma 1.1 is false. Therefore $\exists$ an optimal solution $Opt(j')$ s.t. j_n is scheduled after its deadline. Let j_i be the job replaced by j_n in the optimal solution.

$$Opt(j') = Opt(j) + Penalty(j_n) - Penalty(j_i) \tag{1}$$

$$Penalty(j_n) - Penalty(j_i) \geq 0 \tag{2}$$

$$\Rightarrow Opt(j') \geq Opt(j) \tag{3}$$

Therefore, $Opt(j')$ is not the optimal solution. Hence our assumption was wrong.
Hence Proved.

Let J = $j_1, j_2, ...j_n$ be n jobs in increasing order of penalty.
Let J' = J/j_n

Theorem 1.2 $Opt(J) = Opt(J') + Penalty(j_n)$

Proof

- Part A:
  When we have $OptJ'$ and j_n is to be scheduled. Following cases are possible:
  $$\text{Penalty}(j_n) = \begin{cases} 0 & \textbf{if } j_n \textbf{ is scheduled within its deadline} \\ P_{j_n} & \textbf{if } j_n \textbf{ is scheduled outside its deadline} \end{cases}$$
  $$\Rightarrow Opt(J) \leq Opt(J') + Penalty(j_n) \tag{4}$$
- Part B:
  When we have $Opt(J)$, using lemma 1.1, we can say that job j_n will be scheduled on or before its deadline.
  $$\Rightarrow Opt(J') \leq Opt(J) - Penalty(j_n) \tag{5}$$
  $$4\&5 \Rightarrow Opt(J) = Opt(J') + Penalty(j_n) \tag{6}$$

- The algorithm stops only when entire array $Temp$(which contains all jobs) is traversed. Therefore all the jobs are scheduled.
- The algorithm schedules the jobs with higher penalty in vacant slots that are closest to their deadlines (from the left side).
- So at any given step say i, $Opt(J_i)$ is kept as low as possible. Since the jobs are added in a non increasing order of their penalties, future optimum values would also be minimum. Hence $Opt(J)$ is minimum for this strategy.

1.8 Space Complexity

- Arrays- $Temp$, Opt and $flag$ take $O(n)$ space.
- Tree T takes $O(n)$ space.

$$\text{Overall Space complexity} = O(n)$$

2 Hierarchical Metric

2.1 Description

Given a set of points $P = \{p_1,\ p_2,\ \ldots,\ p_n\}$, with distance function d on the set P.

We need to build a hierarchical metric on P that is constructed as follows:

- We build a rooted tree T with n leaves, and associate with each node v of T, and $h(v)$. Value of $h(v)$ should be such that
  - $h(v) = 0$, for each leaf.
  - If u is parent of v in T, then $h(u) \geq h(v)$.
- For any pair of points p_i & p_j, their distance $\tau(p_i,\ p_j)$ is defined as: We determine the lowest common ancestor v in T of the leaves containing p_i and p_j, and define $\tau(p_i,\ p_j) = h(v)$.
- We say that a *hierarchical metric* τ is consistent with our distance function d, if for all pairs (p_i, p_j), we have $\tau(p_i,\ p_j)\ \leq\ d(p_i,\ p_j)$.
- Given a polynomial time algorithm that takes the distance function d and produces a *hierarchical metric* τ with the following properties:
  1. τ is consistent with d, and
  2. If τ' is any other hierarchical metric constant with d, then τ'$(p_i,\ p_j) \leq\ \tau(p_i,\ p_j)$ for each pair of points p_i and p_j.

2.2 Approach

- First of all, we shall sort the pair of points according to their distances in non-decreasing order. The size of array to sort shall be the number of pairs *i.e.*, $\binom{n}{2}$. Thus, the array will be an array of structures, where is structure keeps p_i, p_j, and $d(p_i,\ p_j)$. And sorting will be done using $d(p_i,\ p_j)$ as key. The **Sort(D)** function does this, in $O(n^2\ log(n))$ time.
- For the nearest pair $(p_i,\ p_j)$, we take an empty rooted tree T, with value of root as $d(p_i,\ p_j)$ and children as p_i and p_j.
- Now to insert the next pair of points $(p_i,\ p_j)$, there are three cases possible.
- **<u>Case 1:</u>** p_i and p_j both have already been added to the tree T.
  - This doesn't require us to do anything.
- **<u>Case 2:</u>** p_i and p_j both have not been added to the tree T.
  - Create a new node *temp*, with value $d(p_i,\ p_j)$ and children as nodes containing p_i and p_j.

- Create a new root, with left child as the old root, and right child as *temp* created above. The value of new root will be the maximum of its child *i.e.*, value of *temp*, since value of root must be less than equal to value of *temp*.

- **<u>Case 3:</u>** Among p_i and p_j, one has been added to the tree T.

  Let the node already added be p_x and the other be p_y.

  - $\tau(x,\ y)$ for any pair of points$(x,\ y)$ already added to the tree will definitely be smaller than $d(p_i,\ p_j)$.
  - Create a new root, with left child as the old root, and right child as node containing p_y. The value of this new root, *i.e.*, τ will be $d(p_i,\ p_j)$.

- An array V of size n can do the job of searching a point in a tree using $O(1)$ time and $O(n)$ space. If the point p_i has been inserted in the tree T, we set $V[i] = 1$, else $V[i] = 0$.

We claim that tree T such formed is the *hierarchical metric* for given (P, d).

2.3 Pseudo Code

2.4 Time Complexity

As shown in the pseudo code, in green text, the time taken by the algorithm will be $O(n^2) + O(n^2\ logn) + O(n^2)$.

Hence, the time complexity of the algorithm is $O(n^2\ log(n))$.

2.5 Space Complexity

The space taken by the algorithm will be, $O(n^2)$ for array D, $O(n)$ for list V, and $O(n^2)$ for tree T.

Hence, the space complexity of the algorithm is $O(n^2)$.

Algorithm 2: Pseudo code for Hierarchical_Metric(P, d)

```
Hierarchical_Metric(P, d){
begin
    Declare an empty array D of structures of size ^nC_2.
    k ← 0; n ← | P |;
    for i ← 1 to n do
            //O(n^2).
        for j ← i to n do
            D(k) → d = d(p_i, p_j); D(k) → points = (p_i, p_j);
            k++;
    Sort(D);          //O(n^2 log(n)).     //Using d as key to compare.
    Initialize an empty array V of size n;
    Create a tree T rooted at root;
    (p_i, p_j) = D(0) → points;
    root→val = d(p_i, p_j);
    create new leaves(children NULL) templeft, tempright;
    templeft→val = p_i; tempright→val = p_j;
    root→left = templeft; root→right = tempright;
    V[i] ← 1; V[j] ← 1;
    for k ← 1 to ^nC_2 do
        //O(n^2).      //O(1) for search, iterating over O(n^2) items of array.
        (p_i, p_j) = D(k) → points;
        if V[i]==1 & V[j]==1 then
            /*Ignore*/;                 //O(1).
        else if V[i]==0 & V[j]==0 then
            create new nodes newroot, temp,;
            create new leaves(children NULL) templeft, tempright;
            templeft→val = p_i; tempright→val = p_j;
            temp→left = templeft; temp→right = tempright;
            temp→val = newroot→val = D(k) → d;
            newroot→left = root; newroot→right = temp;
            root ← newroot;
            V[i] ← 1; V[j] ← 1;
        else
                //When one of them is in V.//O(1) for search in V, rest is O(1).
            create new nodes newroot,and a leaf node temp;
            if V[j]==0 then
                temp→val = p_j; V[j] ← 1;
            else
                temp→val = p_i; V[i] ← 1;
            newroot→val = D(k) → d;
            newroot→left = root; newroot→right = temp;
            root ← newroot;

    return T;
end
}
```

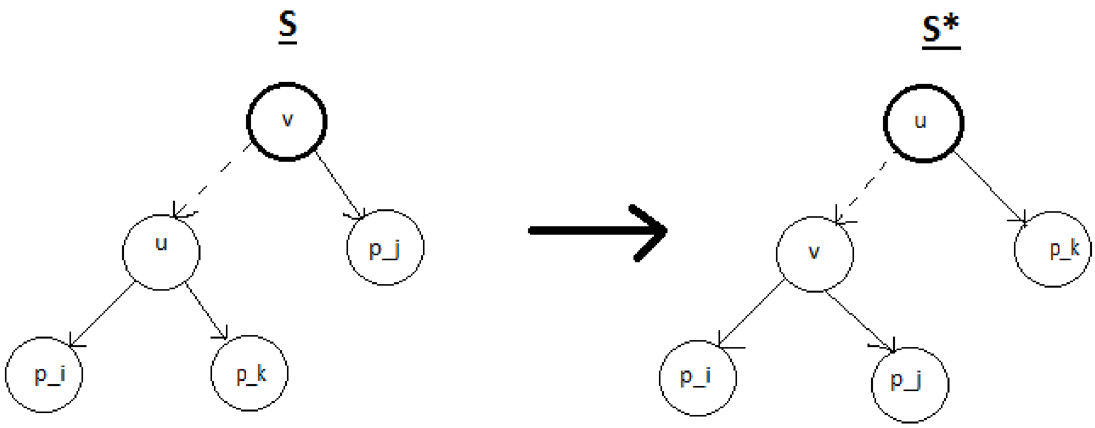
2.6 Proof of Correctness

Lemma

There exists an optimal solution in which the two points having least distance will be siblings in the corresponding binary tree(*hierarchical metric*).

Proof of Lemma

Let S be an optimal solution which is consistent with the given function d but does not follow the lemma. Let $(p_i,\ p_j)$ be the pair of points, that have minimum distance.



From the above figure, we can see that p_k is sibling of p_i. Since node v is the lowest common ancestor of p_i and p_j, $h(v) \leq d(p_i,\ p_j)$. For each node u in $Subtree(v)$, $h(u) \leq h(v)$. As $d(p_i,\ p_j)$ is the minimum distance, therefore each $h(u) = h(v)$ for an optimal solution.
To obtain S^* from S, we swap node p_k and p_j.

- **Consistent with *d*-** S^* is consistent as for each internal node u in $Subtree(v)$, $h(u) \leq h(v) \leq d(p_i,\ p_j)$. From our assumption, $d(p_i,\ p_j)$ is minimum. Therefore, for any two leaf nodes$(x,\ y)$ in $Subtree(v)$ in S^*, their lowest common ancestor(z) will lie in this *subtree* only and $h(z) \leq d(p_i,\ p_j) \leq d(x,\ y)$.

- **Optimal Solution-** As the original solution S was optimal, *i.e.*, if τ' is any other hierarchical metric consistent with d, then $\tau'(p_i,\ p_j) \leq \tau(p_i,\ p_j)$ for each pair of points$(p_i,\ p_j)$, and for all u in $Subtree(v)$, $h(u) \leq h(v)$, and as we can see in S^*, τ is as good as τ in S.

Proof of Correctness

Let A be the set of with n points and $d(p_i,\ p_j)$ as the minimum distance and

$$|\text{Opt(A')}| = |\text{Opt(A)}| - 1,$$

where A' is the set of points where $(p_i,\ p_j)$ are replaced by a single point p'.

<u>**Claim:**</u> **Opt(A) = Opt(A') $\cup$ {node with $\tau(p_i,\ p_j)$}.**

- Opt(A) $\geq$ Opt(A') $\cup$ {node with $\tau(p_i,\ p_j)$}
  - A solution for A can be obtained from A', by extending leaf node p' to include points $p_i,\ p_j$ as its children nodes, and $h(p') = d(p_i,\ p_j)$. Therefore, for an optimal solution, Opt(A),

$$\text{Opt}(A) \geq \text{Opt}(A') \cup \{\text{node with } \tau(p_i,\ p_j)\}$$

- Opt(A') $\geq$ Opt(A) - {node with $\tau(p_i,\ p_j)$}
  - From lemma proved above, there exists an optimal solution for A, such that points$(p_i,\ p_j)$ with minimum distance are siblings in their corresponding binary tree. To obtain a solution for A', we replace the leaf nodes p_i and p_j from the optimal solution of A with some point p'. Therefore, for an optimal solution, Opt(A'),

$$\text{Opt}(A') \geq \text{Opt}(A) \text{ - } \{\text{node with } \tau(p_i,\ p_j)\}$$

From the above two cases, it is clear that :

$$\text{Opt}(A) = \text{Opt}(A') \cup \{\text{node with } \tau(p_i,\ p_j)\}.$$