Assignment 1

Navneel Singhal

2018CS10360

Contents

1 Problem 1

1.1 Statement

Give a greedy algorithm that achieves an approximation guarantee of $O(\log n)$ for set multicover, which is a generalization of set cover in which an integral coverage requirement is also specified for each element and sets can be picked multiple number of times to satisfy all coverage requirements. Assume that the cost of picking α copies of set S_i is $\alpha \cdot cost(S_i)$.

1.2 Solution

Suppose the set $U = \{e_1, \ldots, e_n\}$ is the set of all elements, and $\mathcal{S} = \{S_1, \ldots, S_n\}$ is a collection of sets corresponding to this instance of the problem. Let the coverage requirement for e_i be req_i.

For now, we will assume that the cost of each set is 1.

Consider the following (non-polynomial-time) algorithm (at the end of the proof, we will give a polynomial-time algorithm that is equivalent to this algorithm):

```
1:  function ApproxSetMulticover(U = {e_1, ..., e_n}, S = {S_1, ..., S_m}, req[1...n])
2:      Let X ← U                          ▷ The set of elements whose requirements haven't been met so far
3:      Let A be a multiset of subsets of U, initialized to ∅.
4:      Let f[1...n] be an array initialized to all 0s      ▷ f[i] will be the number of copies of e_i covered so far
5:      while X is non-empty do
6:          Let S_i be a set that covers the most number of elements in X
7:          Insert S_i into A
8:          for e_j in S_i do
9:              f[j] ← f[j] + 1
10:             if f[j] = req[j] then
11:                 Remove e_j from X.
12:             end if
13:         end for
14:     end while
15:     return A
16: end function
```

Note that this terminates since each element is in some S_i.

Now we claim that this algorithm achieves an approximation guarantee of $O(\log n)$.

As in the case of set cover, we shall assign some value v_i to element e_i, such that we have $val(S) := \sum_{e_i \in S} v_i \leq 1$ for all $S \in \mathcal{S}$. We shall call an assignment that satisfies this condition a feasible assignment.

Define **OPT** as the multiset (hence it can have duplicate elements) that achieves the minimum cost. Let A be the multiset that we get from this algorithm.

Claim 1.1

$|\mathbf{OPT}| \geq \sum_{e_i \in U} req_i v_i$ for any feasible assignment of v_i's.

Proof. Note that we have the following (note that the sum is taken over all copies too):

$$\begin{aligned}
|\mathbf{OPT}| &= \sum_{S \in \mathbf{OPT}} 1 \\
&\geq \sum_{S \in \mathbf{OPT}} val(S) \\
&= \sum_{S \in \mathbf{OPT}} \sum_{e_i \in S} v_i \\
&= \sum_{e_i \in U} \sum_{S \in \mathbf{OPT} \wedge e_i \in S} v_i \\
&\geq \sum_{e_i \in U} req_i v_i
\end{aligned}$$

Where the last inequality comes from the fact that **OPT** satisfies all coverage requirements. □

For a set S in A, let X_S be the set of elements with unsatisfied coverage requirements belonging to S when S was added into A. Note that each S uniquely corresponds to some iteration of the loop, since A is a multiset.

Now consider the following assignment of the v_i's. At iteration j, suppose the k^{th} set was picked. Then we add $\frac{1}{|X_{S_k}|}$ to the cost of all elements in X_{S_k}.

At the end of the algorithm, replace v_i by $\frac{v_i}{req_i \cdot (1+\log n)}$ for all i.

To make it clearer, we can label each copy of S with the iteration number, but we choose not to do it to make the proof lighter on notation.

Claim 1.2

This is a feasible assignment of v_i's, i.e., this assignment satisfies the inequality $val(S) \leq 1$ for each $S \in \mathcal{S}$.

Proof. Consider any set $S = \{e_1, \ldots, e_k\} \in \mathcal{S}$, and wlog let the order in which the elements have their coverage requirement met in the algorithm be $e_1, \ldots, e_k$ (reorder the e_i's if needed, in case any of these conditions are not satisfied).

Firstly note that the number of elements with unsatisfied coverage requirements that are present in a set chosen by the algorithm is always non-decreasing (else, we could have chosen the set with a larger number of uncovered elements earlier to get a contradiction to the way we choose the set at an iteration). Suppose S_r was the last set that contributed to v_i. Then e_i had its requirement satisfied when S_r was added to A, and any set before S_r that had e_i as one of the unsatisfied elements contained in it would have at most as much contribution to v_i as the contribution of S_r. Moreover, it must have at least $k - i + 1$ uncovered elements, since otherwise we could have chosen S at that time. So we have

$$\begin{aligned} v_i &\leq \frac{1}{req_i \cdot (1+\log n)} \cdot \sum_{d=1}^{req_i} \frac{1}{|X_{S_r}|} \\ &= \frac{1}{req_i \cdot (1+\log n)} \cdot \frac{req_i}{|X_{S_r}|} \\ &\leq \frac{1}{(1+\log n) \cdot (k-i+1)} \end{aligned}$$

Hence, using this inequality, we have

$$\begin{aligned} val(S) &= \sum_{i=1}^{k} v_i \\ &\leq \sum_{i=1}^{k} \frac{1}{1+\log n} \cdot \frac{1}{k-i+1} \\ &\leq \frac{1+\log k}{1+\log n} \\ &\leq 1 \end{aligned}$$

The second-last inequality comes from the standard bound on the harmonic sums, and the last inequality comes from the fact that $k \leq n$. This completes the proof of the claim. □

Claim 1.3

$|A| \leq (1+\log n) \cdot |\mathbf{OPT}|$

Proof. Note that we have the following:

$$\begin{aligned}
|\mathbf{OPT}| &\geq \sum_{e_i \in U} req_i \cdot v_i \\
&= \frac{1}{1+\log n} \cdot \sum_{e_i \in U} \sum_{S \in A \wedge e_i \in X_S} \frac{1}{|X_S|} \\
&= \frac{1}{1+\log n} \cdot \sum_{S \in A} \sum_{e_i \in X_S} \frac{1}{X_S} \\
&= \frac{1}{1+\log n} \cdot \sum_{S \in A} 1 \\
&= \frac{|A|}{1+\log n}
\end{aligned}$$

This completes the proof of the claim. □

Since A satisfies the coverage requirements (since the termination condition is the set X being empty, and we remove an element from a set upon its coverage requirements being satisfied), we are done.

Now it only remains to show an algorithm that is equivalent to the above algorithm, but which runs in polynomial time.

```
1: function EfficientApproxSetMulticover(U = {e_1, ..., e_n}, S = {S_1, ..., S_m}, req[1...n])
2:     Let X ← U                          ▷ The set of elements whose requirements haven't been met so far
3:     Let count[1...m] be an array initialized to all 0s ▷ count[i] will be the number of copies of S_i picked so
   far.
4:     Let f[1...n] be an array initialized to all 0s      ▷ f[i] will be the number of copies of e_i covered so far
5:     while X is non-empty do
6:         Let S_i be a set that covers the most number of elements in X
7:         Let e_r be an unsatisfied element in S_i with the least number of copies needed to satisfy the requirement
   goal.
8:         Let h = req[r] − f[r]
9:         count[i] ← count[i] + h
10:        for e_j in S_i do
11:            f[j] ← f[j] + h
12:            if f[j] = req[j] then
13:                Remove e_j from X.
14:            end if
15:        end for
16:    end while
17:    return count
18: end function
```

Claim 1.4

This is equivalent to the previous algorithm.

Proof. Note that there is a possible run of the previous algorithm which picks the same set until one of the unsatisfied elements of S is fully satisfied, since we didn't restrict the choice of S_i. So, if e_r is an element with the least number of copies needed to satisfy its requirement, we can repeat the same set S for as many times as required for it to reach the goal, which is $req[r] - f[r]$. Note that doing this doesn't make the f-value of any uncovered element jump over its req-value, else that would have been an element with a strictly less number of copies needed to satisfy its requirement, and would have been chosen instead of e_r. Hence, we can always simulate the next $h = req[r] - f[r]$ steps of the algorithm in a single step, which is what our algorithm does. □

Claim 1.5

This is a polynomial-time algorithm.

Proof. Note that at any step, we remove at least one element from X, since e_r is removed from X. Hence, the number of iterations is bounded above by n. In an iteration, we check for the total number of elements in X, which is done in $O(1)$, then we look for a set with the maximum number of elements in X, and that can be done in $O(nm)$ time, and iterate over all elements in S_i to get the element with the least $freq - f$ value in time $O(n)$, and then iterate over elements, update f, and potentially remove it from X, which is also $O(n)$ time (if we maintain X as a boolean array, and the count of elements in X as an integer variable). So the overall time taken is $O(n^2m)$ (and if we take into account the word size, it'll be multiplied by the word size, which still is $O(n)$, and hence keeps the time taken polynomial). □

2 Problem 2

2.1 Statement

Consider the following 2-approximation algorithm for the vertex cover problem. Find a depth first search tree in the given graph, G, and output the set, S, of non-leaf vertices of this tree. Show that S is indeed a vertex cover for G and $|S| \leq 2 \cdot \mathbf{OPT}$.

2.2 Solution

Firstly, note that if G is not connected, we can note that a minimum vertex cover is the union of minimum vertex covers of each component, so it suffices to show the conclusion for a connected graph.

We shall use the abbreviation DFS-tree for depth-first-search tree in what follows.

Since the DFS tree is a rooted tree, we will use the definition of a leaf as a vertex which doesn't have any children in the DFS tree (since otherwise, the claim in the problem statement doesn't hold for graphs consisting of a single cycle, or a connected simple graph with 2 vertices).

We shall also use the term tree to refer to a rooted tree, unless mentioned otherwise.

Note that this algorithm is indeed a polynomial time algorithm, since DFS is linear in the input size, and storing the children of each vertex in a list associated to that vertex is linear in the input size as well, and we can check for a vertex being a leaf by checking whether this list is empty or not, and add the non-leaves to the answer in linear time as well.

Claim 2.1

S is a vertex cover for G.

Proof. Suppose S is not a vertex cover for G. Then there must exist an edge (u, v) with none of u, v in S, i.e., an edge between two leaves in the DFS-tree.

Note that the DFS-tree of any undirected graph partitions the set of edges of the graph into two sets - tree edges and back edges, where the tree edges correspond to the edges which appear in the DFS-tree, and the back edges (due to the properties of the DFS-tree) join a vertex to an ancestor of its own.

(u, v) can't be a tree edge, since for that to happen, either of u, v has to be the parent of the other in the DFS-tree, and since u, v are distinct leaves, this is a contradiction. It can't be a back edge either, since otherwise either of u, v has to be an ancestor of the other in the DFS-tree, which leads to a similar contradiction.

Hence we have shown that such a case can never arise, which shows that S is a vertex cover for G. □

Claim 2.2

The minimum vertex cover of a graph is at least as large as the minimum vertex cover of its DFS-tree.

Proof. For this, we note that the minimum vertex cover of a graph is also a vertex cover of its DFS-tree, since the set of edges of the DFS-tree is a subset of the set of edges of the graph itself, and hence any edge of the DFS-tree has an endpoint in the vertex cover of the original graph.

Hence the minimum vertex cover of a graph, being a vertex cover of the DFS-tree, is at least as large as the vertex cover of its DFS tree. □

Now if we show that the number of non-leaf vertices of a tree is at most twice the size of a vertex cover of a tree, we shall be done, since the size of S will then be at most twice the size of a vertex cover of the DFS tree, and hence at most twice the size of a vertex cover for the graph, which would complete the proof. Hence, it suffices to show the following claim:

Claim 2.3

For any tree, the size of the minimum vertex cover $VC(T)$ is at most twice the number of non-leaf vertices in the tree.

Proof. For this, we shall need the following two claims.

Claim 2.4

For a tree, there is always a minimum vertex cover which doesn't contain any leaf, but contains all vertices adjacent to at least one leaf.

Proof. Note that there is no edge between two leaves by the definition of leaves in a rooted tree.

Consider the set of edges that are incident on a leaf. Note that the number of vertices adjacent to a leaf is at most the number of leaves, since the degree of a leaf is at most 1 (to see why, note that the partial function from the set of leaves to the set of vertices adjacent to a leaf induced by the adjacency relation is surjective).

Now we do the following: replace each leaf in the minimum vertex cover with the vertex it is joined to by an edge. The resulting set is at most the size of the original vertex cover, and it is still a vertex cover, since the edges not incident to a leaf are still covered, while the edges incident to a leaf are now covered by a possibly different vertex.

Note that since this set is a vertex cover, it must consist of all vertices adjacent to a leaf, since otherwise the corresponding edge won't be covered by any vertex.

All in all, we have shown that we can modify a minimum vertex cover to another vertex cover which has no more vertices (hence it is also a minimum vertex cover) such that no leaf is in the vertex cover but all vertices adjacent to a leaf are, which proves this claim. □

Claim 2.5

Let $L(T)$ be the set of leaves of a non-empty tree T, and let $P(T)$ be the set of vertices adjacent to a leaf. Then $P(T) \cup VC(T \setminus (P(T) \cup L(T))$ is a minimum vertex cover of T (here $VC(\cdot)$ gives an arbitrary minimum vertex cover of a tree (rooted or empty)).

Proof. From the previous claim, we know that there exists a vertex cover $VC(T)$ such that $P(T) \subseteq VC(T)$.

Note that removing $L(T)$ from T makes $P(T)$ the set of leaves of the resulting tree, so $T \setminus (L(T) \cup P(T))$ is still a tree.

Firstly we show that $VC(T) \setminus P(T)$ is a vertex cover of $T \setminus (P(T) \cup L(T))$. Consider any edge (u, v) in the tree formed by removing $P(T)$ and $L(T)$. It can't be incident to any vertex in $P(T)$ or $L(T)$ since we have removed these vertices. Since $VC(T)$ is a vertex cover, at least one of u, v must be in $VC(T)$. Since none of u, v is in $P(T)$, at least one of u, v must be in $VC(T) \setminus P(T)$. Since our edge was arbitrarily chosen, we are done with this part.

Now we show that $P(T) \cup VC(T \setminus (P(T) \cup L(T)))$ is a vertex cover of T. Consider any edge of T. If the edge is incident to a leaf of T, it is covered by $P(T)$. If the edge is incident to any vertex of $P(T)$, it is covered by that vertex in $P(T)$. If none of these is true, then both endpoints must be in $T \setminus (P(T) \cup L(T))$, and hence it is covered by a vertex in $VC(T \setminus (P(T) \cup L(T)))$. So the exhibited set is indeed a vertex cover.

Now we show that this is a minimum vertex cover. Suppose this is not a minimum set cover. By the previous claim, there exists a minimum vertex cover of T which contains $P(T)$. Then since we have shown that if we remove $P(T)$ from any minimum vertex cover of T, we get a vertex cover for $T \setminus (P(T) \cup L(T))$. Doing this to a minimum set cover of T which has $P(T)$ as a subset gives us a smaller vertex cover of $T \setminus (P(T) \cup L(T))$ than $VC(T \setminus (P(T) \cup L(T)))$, which is a contradiction. This completes the proof. □

From the claims above, we can note that the following algorithm gives us a minimum vertex cover of a rooted tree:

```
1: function MinVertexCover(Rooted Tree T)
2:     let A ← {}
3:     let i ← 1
4:     while T has at least 1 vertex do
```

```
        let L_i ← the set of leaves of C
        let P_i ← the set of vertices adjacent to a leaf of C.
        A ← A ∪ P_i
        T ← T \ (L_i ∪ P_i)
        i ← i + 1
    end while
    return A
end function
```

Consider the sets L_i and P_i. We can partition the set of vertices of T as $\cup_{i=1}^{m}(L_i \cup P_i)$ where m is the number of steps taken by the while loop. The corresponding minimum set cover is $\cup_{i=1}^{m} P_i$. The non-leaf vertices of this tree are $S = \cup_{i=1}^{m}(P_i \cup L_{i+1})$, where $L_{m+1} = \emptyset$ for the sake of convenience. Note that all unions here are disjoint unions.

As observed before, we have that P_i is the set of leaves of the tree formed when we remove L_i from T at the i^{th} step, and L_{i+1} is the set of leaves of the tree formed when we remove P_i from this tree. So by an argument similar to the one in the proof of Claim 2.4, we have that $|P_i| \geq |L_{i+1}|$. Hence we have

$$\begin{aligned}
\mathbf{OPT} &= |\cup_{i=1}^{m} P_i| \\
&= \sum_{i=1}^{m} |P_i| \\
&\geq \sum_{i=1}^{m} \frac{|P_i| + |L_{i+1}|}{2} \\
&= \frac{1}{2} \sum_{i=1}^{m} |P_i| + |L_{i+1}| \\
&= \frac{1}{2} |\cup_{i=1}^{m} (P_i \cup L_{i+1})| \\
&= \frac{|S|}{2}
\end{aligned}$$

Rearranging this gives $|S| \leq 2 \cdot \mathbf{OPT}$, as needed. □

3 Problem 3

3.1 Statement

In the *maximum coverage problem*, we have a set of elements E, and m subsets of elements $S_1, \ldots, S_m \subseteq E$. The goal is to choose k sets such that we maximize the number of elements that are covered; an element s is covered if it belongs to one of the sets picked. Give a $\left(1 - \frac{1}{e}\right)$-approximation algorithm for this problem.

3.2 Solution

Consider the following greedy algorithm.

```
1: function ApproxMaxCoverage(E = {e_1, ..., e_n}, S = {S_1, ..., S_m}, k)
2:     Let X ← E                                   ▷ The set of uncovered elements
3:     Let A be a set of subsets of E, initialized to ∅.
4:     for k steps do
5:         Let S_i be a set that covers the most number of elements in X
6:         if X is empty then
7:             break
8:         end if
9:         Insert S_i into A
10:        for e_j in S_i do
11:            if e_j is in X then
12:                Remove e_j from X.
13:            end if
14:        end for
15:    end for
16:    if the total number of sets in A is less than k then
17:        Keep adding an arbitrary set from S to A till we have k sets in A
18:    end if
19:    return A
20: end function
```

Firstly, note that if we implement this algorithm with A being an array where $A[i]$ is the number of copies of S_i we have taken, then initialization of A is $O(m)$, insertion of a set is $O(1)$, and the final check is $O(n)$ too (if we ignore the word size, but if we take it into account, we have to just multiply it by the word size, which can be at most equal to the size of the input, so dealing with A is always a polynomial-time step). Also, the number of iterations is bounded above by n, since we remove at least one element from X at each step. Since traversing a set is polynomial in the size of the input, the whole algorithm is also polynomial in the size of the input.

We claim that this gives us an $\left(1 - \frac{1}{e}\right)$ approximation algorithm.

Let c_i be the number of newly covered elements at the i^{th} step of the algorithm, and define **OPT** as a possible set of elements that are finally covered by an optimal algorithm. We will try to lower bound c_i in terms of **OPT** to get an estimate about the quality of our solution.

Claim 3.1

$$c_i \geq \frac{|\mathbf{OPT}| - \sum_{j=1}^{i-1} c_j}{k}$$

Proof. Consider the set of elements T_i which are in **OPT** but haven't been covered the algorithm till the $(i-1)^{\text{th}}$ step. Since there are k sets in the optimal solution, by the pigeonhole principle, at least one set in the optimal solution covers at least $\frac{|T_i|}{k}$ elements in this set. Since the optimal solution is restricted to choosing sets only from the sets S_k, we get that at least one of the sets S_k covers at least $\frac{|T_i|}{k}$ elements. Since at each iteration, we choose the set which covers the largest number of uncovered elements, we have $c_i \geq \frac{|T_i|}{k}$.

Now note that $\mathbf{OPT} \setminus T_i$ is the set of elements of **OPT** which are covered by the algorithm as of yet, so

$|\mathbf{OPT}| - |T_i| = |\mathbf{OPT} \setminus T_i| \leq |\text{covered elements}| = \sum_{j=1}^{i-1} c_j$, so we get

$$\begin{aligned} c_i &\geq \frac{|\mathbf{OPT}| - (|\mathbf{OPT}| - |T_i|)}{k} \\ &\geq \frac{|\mathbf{OPT}| - \sum_{j=1}^{i-1} c_j}{k} \end{aligned}$$

as needed. □

Claim 3.2

$$\sum_{j=1}^{i} c_j \geq |\mathbf{OPT}| \cdot \left(1 - \left(1 - \frac{1}{k}\right)^i\right)$$

Proof. For $i = 1$, the claim is true due to the claim above (and noting that $c_0 = 0$).

Now for $i > 1$, suppose the claim is true for $i - 1$.

We then have

$$\begin{aligned} \sum_{j=1}^{i} c_j &= c_i + \sum_{j=1}^{i-1} c_j \\ &\geq \frac{|\mathbf{OPT}| - \sum_{j=1}^{i-1} c_j}{k} + \sum_{j=1}^{i-1} c_j \\ &= \left(1 - \frac{1}{k}\right) \cdot \sum_{j=1}^{i-1} c_j + \frac{|\mathbf{OPT}|}{k} \\ &\geq |\mathbf{OPT}| \cdot \left(\frac{1}{k} + \left(1 - \frac{1}{k}\right) \cdot \left(1 - \left(1 - \frac{1}{k}\right)^{i-1}\right)\right) \\ &= |\mathbf{OPT}| \cdot \left(1 - \left(1 - \frac{1}{k}\right)^i\right) \end{aligned}$$

which completes the proof. □

From the last claim, we can lower bound the total number of elements the algorithm chooses (i.e., $\sum_{i=1}^{k} c_i$) by $\left(1 - \left(1 - \frac{1}{k}\right)^k\right) \cdot |\mathbf{OPT}|$. All that remains to be shown is that $\frac{1}{e} \geq \left(1 - \frac{1}{k}\right)^k$, and we would be done. We prove this as follows.

Claim 3.3

For any natural number k, we have

$$\frac{1}{e} \geq \left(1 - \frac{1}{k}\right)^k$$

Proof. Let $f(k) = \left(1 - \frac{1}{k}\right)^k$. Then we have

$$\begin{aligned}
\frac{f(k+1)}{f(k)} &= \frac{\left(1 - \frac{1}{k+1}\right)^{k+1}}{\left(1 - \frac{1}{k}\right)^k} \\
&= \frac{k}{k+1} \cdot \left(\frac{k^2}{k^2-1}\right)^k \\
&= \frac{k}{k+1} \cdot \left(1 + \frac{1}{k^2-1}\right)^k \\
&\geq \frac{k}{k+1} \cdot \left(1 + \frac{k}{k^2-1}\right) \\
&> \frac{k}{k+1} \cdot \left(1 + \frac{k}{k^2}\right) \\
&= 1
\end{aligned}$$

This shows that f is an increasing function on the naturals.

Hence $f(k) \leq \lim_{k\to\infty} f(k) = \frac{1}{e}$, as needed. □

4 Problem 4

4.1 Statement

In the *related machines scheduling problem*, we are given m machines and n jobs. Job j has processing time p_j and machine i has speed s_i. Machine i takes p_j/s_i time to complete job j. Given T design a polynomial time algorithm which either schedules jobs on the m machines such that they finish by time $2T$ or proves that there is no way of scheduling jobs so that they all finish by time T.

4.2 Solution

Without loss of generality, we assume that $s_1 \leq s_2 \leq \ldots \leq s_m$ (else we can always reorder the machines), and we also assume that the jobs are processed by the following algorithm in the order $1, \ldots, n$ (else we can always reorder the job ids). Consider the following algorithm:

function ApproxRelatedMachineScheduling($p[1 \ldots n]$, $s[1 \ldots m]$, T)
    Let $t[1 \ldots m]$ be an array initialized to all zeroes. ▷ $t[i]$ will store the total time taken by all the jobs assigned to machine i so far.
    Let $jobs[1 \ldots m]$ be an array of empty lists ▷ $jobs[i]$ will store the jobs assigned to machine i so far.
    for each job j **do**
        Let machine i be the machine with the smallest i (and hence also the least speed) such that $t[i]+p_j/s_i$ is at most $2T$.
        if no such machine exists **then**
            return failure
        end if
        Add j to $jobs[i]$
        $t[i] \leftarrow t[i] + p[j]/s[i]$
    end for
    return $jobs$
end function

Firstly note that this algorithm is polynomial in the input size, since the number of iterations is upper bounded by n, and in each iteration, we look at each machine, which takes $O(m)$ time, and we add a job and update the t-value for some machine, which is $O(1)$. So the algorithm is overall polynomial-time in the size of the input.

Note that if the algorithm doesn't fail, by the condition that says that we assign any job to a machine whose total execution time won't exceed $2T$, we have that the scheduling that is returned by the algorithm always satisfies the condition that all jobs are finished by time $2T$.

Hence, all there is to show is that if the algorithm fails, there is no way of scheduling jobs so that they all finish by time T. Taking the contrapositive, we need to show that if there is a way to schedule all jobs on the machines with execution time $\leq T$, then the algorithm doesn't fail.

Suppose, for the sake of contradiction, that there exists an instance of the problem where our algorithm fails, and there is a way to schedule jobs in a way that every machines finishes execution in time $\leq T$.

Let **OPT** be the time taken for the optimal scheduling. Then we have **OPT** $\leq T$, since otherwise that would contradict the definition of **OPT**.

Suppose the first job that wasn't processed was job k, where $1 \leq k \leq n$.

Consider the state of the jobs before we tried to schedule job k. Let S_i^{OPT} be the set of jobs assigned to machine i in an optimal solution, and let S_i^{ALG} be the set of jobs assigned to machine i in our algorithm (just before the failure).

We call a machine *bad* if the total time taken by jobs assigned to it by the algorithm is more than T, and *good* otherwise.

Claim 4.1

There is at least one *good* machine.

Proof. Suppose all machines are bad. Then for each machine i, we have:

$$\begin{aligned}\sum_{j \in S_i^{ALG}} \frac{p_j}{s_i} &> T \\ &\geq \mathbf{OPT} \\ &\geq \sum_{j \in S_i^{OPT}} \frac{p_j}{s_i}\end{aligned}$$

This reduces to

$$\sum_{j \in S_i^{ALG}} p_j > \sum_{j \in S_i^{OPT}} p_j$$

Summing this over all machines, and noting that we have only assigned jobs 1 through $k-1$ to the machines in the algorithm, and jobs 1 to n to the machines in the optimal solution, we have

$$\sum_{j=1}^{k-1} p_j > \sum_{j=1}^{n} p_j$$

This is a contradiction, as for all j, we have $p_j \geq 0$, and $k - 1 \leq n - 1 < n$. □

Let i be the largest index such that machine i is good, i.e., i is the index of the fastest good machine.

Claim 4.2

There exists a job that has been assigned to a machine with index $> i$ by our algorithm, but is assigned to a job with index $\leq i$ in an optimal solution.

Proof. Suppose that this is not the case. Then any job that has been assigned to a machine with index $> i$ by our algorithm is also assigned to a job with index $> i$ in any optimal solution. Let the set of jobs assigned to the machines with indices $> i$ by our algorithm be S^{ALG}, and let the set of jobs assigned to the machines with indices $> i$ in an optimal solution be S^{OPT}. Then we have $S^{ALG} \subseteq S^{OPT}$.

Since all machines with index $> i$ are bad, for all machines with index $i' > i$, we have the following inequality from the previous proof:

$$\sum_{j \in S_{i'}^{ALG}} p_j > \sum_{j \in S_{i'}^{OPT}} p_j$$

Summing this over all $i' > i$, we have

$$\sum_{j \in S^{ALG}} p_j > \sum_{j \in S^{OPT}} p_j$$

This is a contradiction, since $S^{ALG} \subseteq S^{OPT}$. □

Consider any such job j that is guaranteed to exist by the previous claim.

Claim 4.3

Job j could have been added to machine i when it was being assigned a machine, without the violation of any constraints imposed by the algorithm.

Proof. Note that since machine i is good, the total time taken by jobs assigned to it at the end of the algorithm is $\leq T$. Since at each step, we can never remove a job from the set of jobs assigned to a machine, the total time taken by the jobs assigned to machine i when job j was being assigned a machine would have also been $\leq T$.

Since an optimal solution assigns job j to a machine i' with $i' \leq i$, we must have $\frac{p_j}{s_i} \leq \frac{p_j}{s_{i'}} \leq \mathbf{OPT} \leq T$, since i' can have speed at most that of i.

So, had we assigned job j to machine i at that point, the total time taken by machine i would have been at most $T + T = 2T$, which doesn't violate the constraints. □

Using this claim, we shall argue our way to a contradiction. Note that at that iteration when we were assigning a machine to job j, we could have assigned it to machine i, however, by the assumption on job j (by the previous claim), we assigned it to a machine with a higher index, which is a contradiction.

This contradiction shows that our assumption about the existence of an instance of the problem where the algorithm fails to return a schedule but there is a schedule where all the machines are free by time T, was wrong, which completes the proof.