CS 4820, Spring 2014 **Homework 2, Problem 1**
Name: Piyush Maheshwari
NetID: pm489
Collaborators: None

(3) *(10 points)* In class, we have seen several examples of optimization problems. These problems come with a notion of what constitutes a solution, and they come with an objective function that measures the quality of a solution. The goal is to find a solution that minimizes (or maximizes) the objective function. For example, in the minimum spanning tree problem, solutions are spanning trees of the given graph and the objective function is the total edge weight of a spanning tree. In the scheduling problem to minimize lateness, solutions are valid schedules for the given requests and the objective function is the maximum lateness of a request.

This way of framing the task of algorithm design — that problems have predefined, mathematically precise objectives, and that the the algorithm designer's job is to solve those predefined problems — is convenient from the standpoint of teaching lectures on algorithms or assigning exercises, but it isn't always an accurate reflection of how algorithm design takes place in real life. In many cases, there is more than one objective function that may be deemed appropriate for the problem at hand. For example for spanning trees, an alternative objective function could be the maximum weight of an edge of the tree (instead of the sum of the weights), whereas for valid schedules, we could consider the sum of the latenesses of requests (instead of the maximum lateness).

What is the right objective function? The choice of the objective function is often guided (and sometimes misguided) by applications and heuristic reasoning. However, it is important to take algorithmic considerations into account. For some objective functions the problem may be hard to solve, whereas for other objective functions, there may be efficient algorithms to solve the problem. In such cases, it often happens that the precise formulation of the objective is dictated by the choice of algorithm and not the other way around.

The following problems study the interaction between algorithms and objective functions.

(3.a) Consider the scheduling problem in Section 4.2 of the textbook. Suppose the goal is to minimize the sum of the latenesses of requests. Show that for this objective function, the earliest-deadline-first algorithm does not always find an optimal schedule.

(3.b) Again consider the scheduling problem in Section 4.2 of the textbook. Suppose every request has a positive weight w_i and the goal is to minimize the weighted sum of latenesses $\sum_i w_i \ell_i$.

Give an efficient algorithm for the special case that all deadlines are equal to the time the resource becomes available (i.e., $d_i = s$ for all $i \in \{1, \ldots, n\}$).

Karma exercise: One of the remarkable features of the minimum spanning tree problem is its robustness: in a very strong sense, the minimum spanning tree is optimal for any reasonable choice of the objective function, not just for the sum of the edge weights. In this sense, the minimum spanning tree problem behaves very differently from the scheduling problem addressed in problems 3a and 3b above.

More precisely, suppose f is a function that assigns a value to every $(n-1)$-tuple of numbers, and suppose

- f **is symmetric:** its value is invariant under permuting the elements of the $(n-1)$-tuple;
- f **is monotone:** its value never decreases when we increase one element of the $(n-1)$-tuple while holding the other elements fixed.

Examples of symmetric monotone functions are the sum, the maximum, and the median.

For any such f, we can consider the "f-minimum spanning tree problem" in which the goal is to find a spanning tree that minimizes the value of f, when it is applied to the $(n-1)$-tuple of edge weights in the tree. Prove that any minimum spanning tree of an edge-weighted graph is also an f-minimum spanning tree for *every* symmetric, monotone objective function f.

Solution

(3.a) Consider two schedules (s_i, t_i, d_i) and (s_j, t_j, d_j) where (s_i, t_i, d_i) represent start time, duration and deadline of schedule i. Assume that both d_i and d_j are less than the time we schedule the first job. Let $d_i < d_j$. Let l_i and l_j denote the lateness of interval i and j. Now if we have follow an earliest deadline algorithm then i will be scheduled before j.

$l_i = s_i + t_i - d_i$
$l_j = f_j - d_j$
So the total sum of lateness will be $(l_i + l_j) = s_i + t_i + f_j - d_i - d_j$.

Now take a counter example when j is scheduled before i and rest everything is the same. In that case we will have the following :
$l'_i = f_j - d_i$
$l'_j = s_i + t_j - d_j$
So the total sum of lateness will be $(l_i + l_j) = s_i + t_j + f_j - d_i - d_j$.

Now the sum of lateness in the second case will be less than the sum of lateness in the schedule output by the algorithm if : $s_i + t_j + f_j - d_i - d_j < s_i + t_i + f_j - d_i - d_j \equiv t_j < t_i$. This means that if we have $t_j < t_i$ in this case, then the earliest deadline first algorithm does not find an optimal solution. This proves that for this objective function earliest deadline first does not always find an optimal solution.

(3.b)

Building the algorithm

Goal is to minimize the weighted sum of lateness $\sum_i w_i * l_i$ which can be written as $\sum_i w_i * (f_i - d_i)$. Since all deadlines are equal to s, the only factors affecting the objective function are w_i and f_i. Our goal is to minimize the sum of lateness, hence we would want to attach the least finishing time to the job that has the maximum weight. But we would also want to jobs to finish quickly i.e. we want to reduce f_i. This means that we would want to run the job with minimum t_i first. So w_i wants to schedule jobs in decreasing order of weights but t_i wants to schedule the jobs in increasing order of weight. It is obvious that using any one of the parameters in a greedy approach won't be optimal. Lets try to the w_i/t_i as a parameter for greedy algorithm. Since we

want the max w_i and least t_i to schedule first, we should use the job in decreasing order of w_i/t_i. As it turns out this approach minimizes the objective function given the problem.

Algorithm

1: Sort the request based on decreasing values of w_i/t_i
2: Schedule the requests in order obtained in step 1

Proof of correctness

Let $x_i = w_i/t_i$ for $\forall i$

We call a pair i, j an inversion if i is scheduled before j and $x_i < x_j$.

Lemma 1. *If there is a solution which has an inversion i, j, then S has a consecutive inversion.*

Proof. With loss of generality assume $i < j$. Consider the element $i+1$. If $i, i+1$ is an inversion then our lemma is proved, otherwise consider $i+1, i+2$. Here we know that since i, j was an inversion and $i, i+1$ was not, $i+1, j$ must be an inversion using the definition of inversion. We can use the same argument on $i+1, i+2$ and again consider the same two cases. We can see that either there must be an element m, n between i, j which is an inversion or $j-1, j$ must be an inversion since no two consecutive pairs between $i, j-1$ (included) were an inversion and i, j is an inversion. Hence there must be a consecutive inversion in any case. □

Lemma 2. *Any optimal schedule which minimizes the objective function has no gaps between consecutive jobs scheduling.*

Proof. This is similar to the proof done in the class for the scheduling problem. If there is any schedule with a gap in between, we can remove the gap and get a schedule which is strictly better than the previous schedule since the lesser the finishing time is, the lower is our objective function. □

Suppose the solution output from our greedy algorithm was S_0. There will be no inversions in S_0 since we schedule the jobs in an reverse sorted order of w_i/t_i. Also let a solution S' be closer to S_0 than S if S' has less inversions than S.

Also if an job is scheduled at any index i, its f_i is $\sum_{1}^{i} t_i$ since by lemma 2 we know that any optimal solution won't have any gaps.

Now consider any candidate solution S different from S_0. Assume that S has no gaps since we know that from lemma 2 that if a solution has gaps, we can find another solution which is better by just removing the gaps. Since it is different from S_0 it must have some inversions. From lemma 1 we know that it must have a pair of consecutive inversion. Let that pair be $i, i+1$. Consider the objective function in this case -S

$$f = [\sum_{k=1}^{i-1} w_k * f_k] + w_i * f_i + w_{i+1} * f_{i+1} + [\sum_{k=i+1}^{n} w_k * f_k]$$

$$f = [\sum_{k=1}^{i-1} w_k * f_k] + w_i * (f_{i-1} + t_i) + w_{i+1} * (f_{i-1} + t_i + t_{i+1}) + [\sum_{k=i+1}^{n} w_k * f_k]$$

Now consider a transformation in which we exchange i and $i+1$. In this case the objective function would be

$$f' = [\sum_{k=1}^{i-1} w_k * f_k] + w_{i+1} * (f_{i-1} + t_{i+1}) + w_i * (f_{i-1} + t_i + t_{i+1}) + [\sum_{k=i+1}^{n} w_k * f_k]$$

Now consider the difference f' - f . Removing the common terms before i and after $i+1$.

$$f' - f = w_{i+1} * (f_{i-1} + t_{i+1}) + w_i * (f_{i-1} + t_i + t_{i+1}) - w_i * (f_{i-1} + t_i) + w_{i+1} * (f_{i-1} + t_i + t_{i+1})$$

$$f' - f = w_i * t_{i+1} - w_{i+1} * t_i$$

Since we know that $i, i+1$ is an inversion, $(w_i/t_i) <= (w_{i+1}/t_{i+1})$, which means $(w_i * t_{i+1}) <= (w_{i+1} * t_i)$.
This means that $f' - f <= 0$, which means that $f' <= f$. So after applying this transformation to S, we have a solution S' which has one less inversion that S which means that it is closer to S_0 than S and it is doing no worse that S since $f' <= f$. Using this fact and the exchange argument, we can say that the greedy solution S_0 the optimal.

Analysis

Since step 1 of the algorithm uses sorting, the running time of the algorithm is $O(nlogn)$.