Introduction to Algorithms | **Problem Set 2 (3 problems)**
CS 4820 Fall 2018 | **Solutions**

(1) Earliest Deadline First In the class on Friday, Aug 31st, we showed that the Earliest Deadline First scheduling algorithm minimizes the maximum lateness. The problem gives with n jobs, each with deadline d_i and a processing time t_i. If job i finishes at time C_i (completion time) than its lateness is $\max(0, C_i - d_i)$, that is, its not late if $C_i \leq d_i$ and otherwise late by $C_i - d_i$ amount. We showed that the Earliest Deadline First scheduling algorithm minimizes the maximum lateness, i.e., minimizes the value $\max_i \max(0, C_i - d_i)$ among all schedules.

Minimizing maximum lateness is often not the right objective function. Two alternative objectives that may describe the goal better are

(i) minimize the sum of all latenesses: using the notation above, this objective is to minimize $\sum_i max(0, C_i - d_i)$.

(ii) minimize the number of jobs that are late: using the same notation, minimize the number of jobs i with $C_i > d_i$.

(a) Consider the objective (i). Either prove that the Earliest Deadline First greedy algorithm is guaranteed to produce the optimal schedule also for this objective, or give an example to show that this is not the case.

Solution: Not optimal. For example, deadlines are $d_1 = 5$, $d_2 = 6$, and processing times are $t_1 = 10$ and $t_2 = 1$. With earliest deadline first, we get both jobs late, for the total lateness as 5+5=10. Doing job 2 first would make that finish on time, and leave the first job to finish 1 unit of time later, with a total lateness of 6 only.

Common mistakes: Providing a correct counterexample that lacks sufficient labeling, thereby allowing it to be interpreted as an incorrect counterexample.

(b) Consider objective (ii). It makes sense to modify the Earliest Deadline First greedy algorithm by delaying all jobs that won't complete in time until the very end (as the objective is not sensitive to the amount of lateness). So, the modified version of the algorithm is now as follows:

- Order the jobs by deadline.
- Consider them in this order: when it's job i's turn, check if i can finish by its deadline, do it if it can, and move to the next job, if it cannot.
- Once all deadlines passed, process all the late jobs.

Either prove that this algorithm is guaranteed to produce the optimal minimizing the number of late jobs, or give an example to show that this is not the case.

Solution Not optimal. For example, deadlines are $d_1 = 10$, $d_i = 11$ for all $i = 2, \ldots, 11$, with $t_1 = 10$ and $t_i = 1$ for $i = 2, \ldots, 11$. If we do job 1 only at the very end, all other jobs finish on time, with the proposed schedule, only job 1 is on time.

Common mistakes: Providing a counterexample that works for EDF but not for *modified* EDF. Providing a correct counterexample that lacks sufficient labeling, thereby allowing it to be interpreted as an incorrect counterexample.

(2) Mentor Assignment You are managing a company with n regular employees, and you just hired some $m < n$ students for summer internships. You would like to pair each new student with one of the n employees as a mentor, where each worker can mentor at most one new student. With the summer being vacation time for many, you need to be aware that not all employees are available for monitoring all the time. For each employee i you asked them for an interval of days $[s_i, f_i]$ that they are available. You would like the assignment to be such that for all new students, if student j starts on day a_j, and is assigned mentor i than $s_i \leq a_j \leq f_i$. So the input to your problem consists intervals $[s_i, f_i]$ for the workers, and start date for the students a_j.

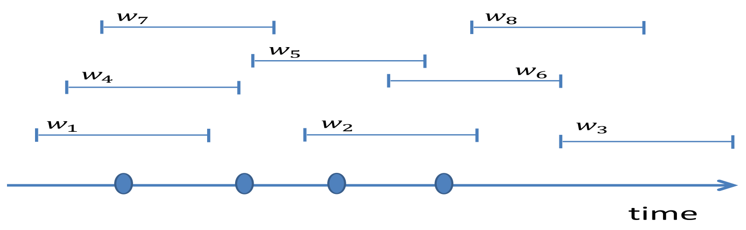

Figure 1: Time intervals of $n = 8$ workers, with dots on the time axis indicating the arrival times of $m = 4$ students.

In the example depicted on the Figure above, we can pair the students in arrival order with workers w_1, w_7, w_2, and w_6.

Give an $O(mn)$ polynomial time algorithm that assigns each new student to a mentor satisfying the requirements above, whenever such an assignment is possible. You may assume that $\log n \leq m$.

Karma question: For extra challenge, you can do this in $O(n \log n)$ time, but this part will not be graded.

Solution: Consider the greedy algorithm that sorts mentors by finish time f_i, sorts students by their arrival time a_j. Assume that for simplicity of notation both students are workers are numbered in this order, so $a_1 \leq \ldots \leq a_m$ and $f_1 \leq \ldots \leq f_n$. Now consider students in this order and assign each student to the first mentor that is available at the student's arrival time, and is not yet assigned as a mentor.

We need to implement this in $O(n \log n)$ time, sorting the two lists takes this time (as $m \leq n$). To be able to implement the assignment in this time, we need to find the first mentor that is available at the student's arrival time, and is not yet assigned as a mentor. This is easy to do in $O(n)$ time, just walk through the list of free workers from the point where $f_i \geq a_j$, and find the first one with $s_i \leq a_j$ (if there is such an item). This is a total of $O(mn + n \log n)$ time. This works as $m \geq \log n$. If m is very small, you can skip sorting the mentors, and simply walk through the mentor list each time to find the mentor identified by our greedy rule.

We prove correctness by an exchange argument. We need to prove that there is a valid assignment then the greedy algorithm above must also succeed. By show this by contradiction. Consider a valid assignment M that agrees with the greedy assignment for as long as possible. Let j be the first student assigned to a different mentor k that is not the first mentor i that is available at the student's arrival time, and is not yet assigned as a mentor. If i is not assigned a student in M, we can swap, assign j to i, and get a valid assignment that agrees with the greedy assignment longer.

If i was assigned to be a mentor, let ℓ be the mentee. We will show that swapping the assignment (j, i) and (ℓ, k) is also a valid assignment, an assignment that agrees with the greedy one longer.

The mentor k is an option also, so $i \leq k$ must be the case, and hence $f_i \leq f_k$. The mentee ℓ assigned to i in M must come after j so $j \leq \ell$, and so $a_j \leq a_\ell$ also. We know k was assigned to j, so $s_k \leq a_j \leq a_\ell$, and i was before k on the list so $f_k \geq f_i \geq a_\ell$, so indeed (k, ℓ) is also a valid assignment.

(To improve on this running time, you can keep a more complicated data structure of mentors, e.g. a self-balancing tree. Start with the list of mentors sorted in order of start time s_i. When about to find a mentor for a student with arrival time a_j, go through the list of mentors and insert them in the tree using their finish time f_i for placement until you reach a mentor in the list with start time after a_j, at which point you stop inserting. Then, you look up the mentor with the earliest finish time after a_j in the search tree. Once a mentor is assigned to a student mentee, delete the mentor from the tree. For e.g. a red-black tree, inserting each mentor into the tree takes $O(\log n)$ time (and happens n times), looking up the next mentor takes $O(\log n)$ time (and happens $m \leq n$ times), and deleting each mentor takes $O(\log n)$ time (and happens $m \leq n$ times). In total, then, this would take $O(n \log n)$ time.)

Common mistakes: The most common mistake was to not choose the first student that was assigned a different mentor. Also, several solutions forgot to repeat the exchange until the optimal solution makes the same assignments as the greedy solution.

(3) Priority Scheduling. You are helping your friend decide on how to schedule jobs in his operating system. Here is a simple version of this problem. You have n jobs, where job i requires time $t_i > 0$ to complete, and has an importance or priority value w_i (with higher weight being more important). You are asking for a simple policy of what order to do these jobs. Let C_i denote the finishing time of job i. For example, if job j is done first, its finishing time will be $C_j = t_j$; if it is done right after job i, we would get $C_j = C_i + t_j$.

You would like to order jobs, so that the important jobs will complete quickly. More concretely, you would like to order the jobs so as to minimize the weighted sum of completion times, i.e., to minimize $\sum_{i=1}^n w_i C_i$. Note that there are no deadlines in this problem, unlike problem 1 above.

For example, if we have two jobs with $t_1 = 2$ and $w_1 = 8$, and $t_2 = 5$ and $w_2 = 6$, then doing job 1 first, we get completion times $C_1 = 2$ and $C_2 = 2 + 5 = 7$, and the sum $\sum_{i=1}^n w_i C_i = 8 \cdot 2 + 6 \cdot 7 = 16 + 42 = 58$. Ordering the jobs with job 2 first, will get us $C'_1 = 7$ and $C'_2 = 5$ and $\sum_{i=1}^n w_i C'_i = 8 \cdot 7 + 6 \cdot 5 = 56 + 30 = 86$, which is worse.

They are considering three different greedy algorithms:

(a) order jobs in increasing order of the time they require, i.e., do short jobs first.

(b) order jobs in decreasing order of weight, i.e., do jobs for important customers first.

(c) order jobs in decreasing order of the ratio of the weight to the time, w_i/t_i.

Assume for simplicity that all jobs require different amount of time, all customers have different weights, and all ratios w_i/t_i are different.

For each of these proposed greedy algorithms, either prove that the algorithm is guaranteed to produce the optimal schedule (minimizing the total $\sum_{i=1}^n w_i C_i$), or give an example to show that this is not the case. (You do not have to consider how to implement the greedy algorithms, and do not have to analyze running time).

Solution:

(a) not optimal. for example $t_1 = 1$, $t_2 = 2$, and $w_1 = 1$ and $w_2 = 100$. Ordering the 2nd job first give us 200+3=203 in total cost, while ordering the 1st job first gets us 1+300=301, which is worse

(b) not optimal. for example $t_1 = 100$, $t_2 = 1$, and $w_1 = 2$ and $w_2 = 1$. Ordering the 2nd job first give us $2 + 2 \cdot 101 = 204$ in total cost, while ordering the 1st job first gets us $2 \cdot 100 + 101 = 301$, which is worse.

(c) This is optimal. To show this, we use the exchange argument. Consider the optimal order. We will show that this has to be the order from part (c). To simplify notation, assume items are named in the order in the optimum. Assume this is not the order from (c), then there must be two items i and $i+1$ such that $w_i/t_i < w_{i+1}/t_{i+1}$. Now consider the change in total weighted completion time, if we swap the order of i and $i+1$. For all other jobs j the completion time is not affected. What is changing is C_i and C_{i+1}. C_i increases by t_{i+1} and C_{i+1} decreases by t_i. So the total $\sum_j C_j w_i$ increases by $w_i t_{i+1}$ and decreases by $w_{i+1} t_i$. By our assumption $w_i/t_i < w_{i+1}/t_{i+1}$ and so $w_i t_{i+1} < w_{i+1} t_i$, so the total sum decreases more than it increases, reaching the claimed contradiction.

Common mistakes: Using the exchange argument is best if we try to swap two neighboring items. If you swap two items i and j with $w_i/t_i < w_j/t_j$ but don't assume they occur next to each other, this hard to argue about. Assume jobs are numbered in the order the optimum put them, so by this assumption $i < j$. Now the change in the objective by swapping jobs i and j is

$$w_j \sum_{k=i}^{j-1} t_k - t_j \sum_{k=i}^{j-1} w_k$$

and I don't see how to show this is a decrease.

As another note, when using exchange argument you must properly set up the argument. Remember to state all your assumptions and make it clear how you are using exchange argument to prove correctness.