Scheduling Algorithms

Rohith Prabakar
Computer Science 351H
University of Maryland
UID: 115653030

May 11, 2019

1 Abstract

In this paper, we shall go over the problem of scheduling jobs that have no intrinsic weight. We shall discuss 4 different algorithms to solve the problem. The paper will go over their benefits, drawbacks, time and space complexity and when to use the algorithm. But in short, we find that the problem can be solved optimally in $\mathcal{O}(nlogn)$ time. We will also go over other similar research paper in the literature review (Prior Research on Related Algorithms).

2 Prior Research on Related Algorithms

2.1 K-Bounded Preemptive Scheduling

2.1.1 Problem Statement

Given a set of jobs along with their release time, deadline, weight(higher weights are more important), number of preemptions(K) and processing time we have to find a set of jobs that can be completed such that their weights are maximized.
Since we allowed preemptions (meaning that a job can be interrupted by a more important job) we have to take that into account.
This Problem has been studied by Sivan Albagli-Kim, Baruch Schieber, Hadas Shachnai and Tami Tamir.

2.1.2 Input

- **line 1** n, k - where n is the number of jobs and k is the number of preemptions
- **lines 2 to n+1** processing time(p), release time(r), due date, weight(d) (+ve)

2.1.3 Goal

To schedule tasks such that the weights of the jobs are maximized.

2.1.4 Why Bounded Preemption

For some practical tasks such as IO Scheduling, preemption is either impossible or too expensive. Thus the only good solution is to have bounded preemptive scheduling. In other words, a task can only be preempted k times.

2.1.5 Algorithmic Approximations

The 3 algorithms are proposed to solve the problem Albagli-Kim et al. [1]. We shall look at the algorithms and state the observations.

Algorithm 1 Maximum Utilization

1: Sort the jobs in non-increasing order of processing times, that is, p1 $\geq$ p2 $\geq$...$\geq$ pn.
2: Consider the jobs in the sorted order
3: **for** the current job j **do**
4: If j can be added to the schedule, that is, if the total length of at most k + 1 idle-segments in [rj,dj) is at least pj, schedule j in the leftmost feasible way. Otherwise, reject j.
5: **end for**

Algorithm 2 General Greedy Scheme

1: S = $\emptyset$.
2: Use a certain rule to sort the jobs; consider the jobs in this order.
3: **for** the current job j **do**
4: If j can be added to the schedule, that is, if the total length of at most k + 1 idle-segments in [rj,dj) is at least pj, then schedule j in the leftmost feasible way and update S. Otherwise, reject j.
5: **end for**
6: Return S.

Algorithm 3 The heuristics H_1 and H_2

1: S = $\emptyset$.
2: Use a certain rule to sort the jobs; consider the jobs in this order.
3: **for** the current job j **do**
4: If no interval in j's window is available, or if j was already scheduled in k + 1 segments, reject j.
5: Let I be an interval with the lowest demand in j's window; schedule j in I.
6: (H_1) If j is not fully scheduled, and at least one of the two intervals adjacent to I is idle, schedule j on the one having minimal demand, and proceed in the same way. If both of the adjacent intervals are busy, go to step 4.
(H_2) If j is not fully scheduled, go to step 4.
7: Update S and the demands on all intervals.
8: **end for**
9: Return S.

2.1.6 Results

1 This Problem is NP hard for K = 1. This is only with unit weights. If the weights are uneven then it it's strongly np hard for k = 1. Albagli-Kim et al. [1]

2 This Problem is strongly NP hard for K > 1. Albagli-Kim et al. [1]

2.1.7 Observations

1 Maximum utilization algorithm uses the maximum CPU usage. We get this as we get k + 1 slots that have the maximum processing time. Albagli-Kim et al. [1]

2 General greedy scheme coincides with Maximum utilization algorithm when run on an instance of the maximum utilization problem and the jobs are considered in non-increasing order of weights.Albagli-Kim et al. [1]

3 H_1 and H_2 give higher priority to idle-intervals that are less likely to be required by other jobs. Albagli-Kim et al. [1]

4 Let density be $\frac{np_{\max}}{2L}$ where L is the last deadline. When running all algorithms with the same inputs and different densities it is found that the performance of greedy (AlgMU) is much better than the theoretical 4-ratio. The heuristics H1 and H2 perform better than greedy for low densities and are close to each other, while for very high density, greedy performs better. Albagli-Kim et al. [1]

5 Sorting by weight is much better than sorting by the load. Albagli-Kim et al. [1]

6 The heuristics performance is mainly affected by the order in which the jobs are considered. Sorting the jobs in non-decreasing order of the ratio processing-time/weight yields the best performance. Comparing the different heuristics with the same order of jobs, the researchers conclude that the best heuristic is H2. Albagli-Kim et al. [1]

2.2 Polynomial Time Algorithms for Minimizing the Weighted Number of Late Jobs on a Single Machine with Equal Processing Times

2.2.1 Problem Statement

The problem is to schedule jobs such that we maximize the weights of the jobs to be completed in a given duration. In this problem, we assume that the time taken to complete each process is the same. This problem is analyzed by Philippe Baptiste.

2.2.2 Input

- **line 1** n, p, d - where n is the number of jobs, p is the processing time for each process and d is the total duration
- **lines 2 to n+1** - w, due - where w is the weights for each process and due is when the process is due

2.2.3 Goal

To get process p_1 p_2 ... p_n such that $\sum_1^n weight(p_i)$ is maximum.

2.2.4 Results

1 When preemptions are not allowed the problem can be solved in $\mathcal{O}(n^7)$ Baptiste [2]

2 When preemptions are allowed the problem can be solved in $\mathcal{O}(n^4)$ Baptiste et al. [4]

3 Both algorithms involve dynamic programming Baptiste [2]

2.3 A Dynamic Programming Algorithm for Preemptive Scheduling of a Single Machine to Minimize the Number of late jobs

2.3.1 Problem Statement

The problem is to schedule jobs such that we maximize the weights of the jobs to be completed in a given duration. For this problem, preemption is allowed and is not bounded by any constraint. This problem has been analyzed by E.L. Lawler in 1990.

2.3.2 Input

- **line 1** n - where n is the number of jobs
- **lines 2 to n+1** - w, due, release, p - where w is the weight for each process, due is when the process is due, release is the release date prior to which the job cannot be processed and p is the processing time.

2.3.3 Goal

To get process p_1 p_2 ... p_n such that $\sum_1^n weight(p_i)$ is maximum in a given time.

2.3.4 Results

1 The time complexity of the algorithm is $\mathcal{O}(nk^2W)$ where W is the sum of all w and k is the number of distinct release dates. Lawler [8]

2 The space complexity of the algorithm is $\mathcal{O}(k^2W)$ where W is the sum of all w and k is the number of distinct release dates. Lawler [8]

3 For the problem in which the objective is simply to minimize the number of late jobs, the pseudopolynomial time bound becomes polynomial, i.e. $\mathcal{O}(n^3k^2)$ where W is the sum of all w and k is the number of distinct release dates. Lawler [8]

2.3.5 Prelude to Algorithm

In this section, we go over some basic concepts to understand the time. This is to help better understand the algorithm provided by Lawler.

EDD (Earliest Due Date) rule • Schedule jobs such that there is a decision point after every job is completed and when a new release date is encountered.

- At each point choose the job with the earliest due date.
- This can be implemented by first sorting by release date and then by using a priority queue (Heap). This can be done in $\mathcal{O}(nlogn)$

r(S) $= \min\{r_j\}, \forall\ j \in S$

p(S) $= \sum p_j, \forall\ j \in S$

w(S) $= \sum w_j, \forall\ j \in S$

c(S) = the time the last job in S is completed in an EDD schedule.

$C_j(r, w)$ Let r be a release date and w be an integer, $0 \leq w \leq W$. Define $C_j(r, w)$ to be minimum value of c(S) with respect to feasable sets $S \subset \{1, 2, ..., j\}$ with $r(S) \geq r$ and $w(s) \geq w$. If there is no such feasible set S, then let $C_j(r, w) = +\infty$. Accordingly, the maximum weight of a feasible set is given by the largest value of w such that $C_n(r_{min}, w)$ is finite where $r_{min} = \min_j \{r_j\}$.

$P_{j-1}(r, r', w'')$ is the minimum processing done in interval $[r_j, r']$, with respect to feasible sets $S'' \subseteq \{1, 2, ..., j - 1\}$ with $r(S'') \geq r$, $c(S'') \leq r'$, $w(S'') \geq w''$. If there is no such feasible set S", then let $P_{j-1}(r, r', w'') = +\infty$

2.3.6 Algorithm

Here we go over algorithm(mostly just recurrence relations) provided by Lawer. Lawler [8]

1 Sort jobs by due date as $d_1 \leq d_2 \leq ... \ d_n$

2 We get a recurrence relation Lawler [8]

$$C_j(r, w) = \min \begin{cases} C_{j-1}(r, w) \\ max\{r_j, C_{j-1}(r, w - w_j)\} + p_j \\ min\{C_{j-1}(r', w') + max\{0, p_j - r' + r_j + P_{j-1}(r, r', w - w_j - w')\}\} \end{cases}$$

Note that the second condition is only true if the right-hand side does not exceed d_j. If this isn't true we set $C_j = +\infty$.

3 We get a relation for P. We have Lawler [8]

$$P_{j-1}(r, r', w'') = \begin{cases} C_{j-1}(r, w) \\ max\{r_j, C_{j-1}(r, w - w_j)\} + p_j \\ min\{C_{j-1}(r', w') + max\{0, p_j - r' + r_j + P_{j-1}(r, r', w - w_j - w')\} 0 \leq w' \leq w''\} \end{cases}$$

With the initial conditions
$P_{j-1}(r, r', 0) = 0$ for $j = 1, 2, ..n$.
$P_0(r, r', w'') = +\infty$ for $w'' > 0$.

4 With this we can get a subset S with maximum weight.

2.3.7 Complexity

- Time taken to complete $P_{j-1}(r, r', w'')$ for each iteration is $\mathcal{O}(k^2W^2)$ Lawler [8]
- Time taken to complete $C_j(r, w)$ for each iteration is $\mathcal{O}(k^2W^2)$ Lawler [8]
- Thus the time complexity is $\mathcal{O}(nk^2W^2)$ Lawler [8]
- Thus space complexity is $\mathcal{O}(k^2W)$ Lawler [8]

2.4 Scheduling n Jobs with Equal Release Dates

2.4.1 Problem Statement

The problem is to n schedule jobs such that we maximize the weights of the jobs to be completed in a given duration. For this problem, we assume that all jobs have the same release time and are associated with a monotone non decreasing deferral cost function. The Well known algorithm of Moore and Hodgson solves this problem. Moore [9]

2.4.2 Input

- **line 1** n - where n is the number of jobs
- **lines 2 to n+1** - w, d - where w is the weight for each process and due is when the process is due

2.4.3 Goal

To get process p_1 p_2 ... p_n such that $\sum_1^n weight(p_i)$ is maximum in a given time.

2.4.4 Result

1 This problem can be solved in $\mathcal{O}(nlogn)$ Moore [9]

2.5 Brief Note on Preemptions

What are Preemptions?
Preemptions is just a fancy way of saying interruptions while running a program. If we find a proper schedule of jobs j_1, j_2, ...j_n that maximizes weight and we get a new job j_k that when included gives a different set of jobs that maximizes weight we have two options

1 We can stop running the current job and run the sequence of jobs that produce the optimum solution.

2 We can finish running the current job and then recompute the optimum solution and run those set of jobs.

The first version is an example of preemption. There are some benefits and drawbacks to preemption.
The **benefits** are that it might be faster for certain types of problems.
The **Drawbacks** of preemptions are that preemptions have a certain overhead which might be more expensive than just letting the program finish. Hence if the overhead is more expensive than an average program, preemptions should be avoided.

3 Problem Statement

The problem is to schedule jobs such that we maximize the weights of the jobs to be completed in a given duration. However, for this problem we consider all of the weights to be the same. Hence we just want the set of jobs that can be completed in a given duration such that the set has the maximum number of jobs. For this problem, there can be a constant stream of input and our algorithm must produce the best set of jobs. For this problem, we do not allow preemptions. The reasons are stated in section 2.5

3.1 Why This Problem?

These days there are many applications that have processes without a notion of weight(importance). If we look at WhatsApp, there is a server that receives and sends messages. It is difficult to gauge the importance of one message over another. In fact, If you wanted to do analyze the message to get an estimate of its importance, it will be a huge privacy violation of the user. Similarly, in applications such as email, youtube and even car ride services such as uber, all of the individual

transactions are equally important. Hence, the performance of a service is based on the number of individual transactions that are completed.
The algorithm must be able to accept streams of input to model real-world data. Person A might request a ride at 10 am and a person B can request a ride at 10:01 am. We should organize our service such that both user's request is fulfilled.

3.2 Input

Line 1 n where n is the number of Jobs.

Line 2 to n + 1 b d p where d is the deadline and p is the processing time required and b is the start time.

Note For this problem, we only get the info about a job exactly on its begin time. To implement a streaming interface we will use an array of all info and only add on the to be processed list on ints begin time. For this problem, each loop iteration is a unit of time. So the first time we go through the loop one second will pass. This is to abstract all the complicated time comparisons away and get a small simplified problem to work with. For this problem, we store the input as a list. so the first index represents the begin time, second the deadline and third the processing time.

3.3 Goals

To get a set of process p_1 p_2 ... p_n such that n is maximum. For simplicity, we shall just return n. In the following few sections, we will discuss various algorithms to solve this problem, analyze its complexity and find its tradeoff. We will then go over the ideal algorithms one should use to solve the problem.

3.4 Preprocessing

Before we discuss the algorithms, let's talk about how we are going to implement it.
Note Not all algorithms require preprocessing.

- Lets store all the jobs to be processed in a list. Let us then sort it by beginning time.
- Lets now loop over till this list is empty. We will have another list (newlist) that stores all the jobs that we can process (begin time $<$ current time).
- Here We pick a job from newlist. Then we remove it from the newlist and the original list. We then increment the time as time is simplified to measure iterations. Our algorithms deal with how we pick the job to be processed from the newlist.

Some of the Code(item 1) for preprocessing is present in the main method in main.py. Others will be handled within the different algorithms.

3.5 Naive Trivial Algorithm

3.5.1 Logic

The Easiest way to solve the problem is to permute all of the elements in the newlist. We then check each of the permutations as to how many jobs can be completed within the time.

3.5.2 Complexity

The Time complexity for finding all permutations is $\mathcal{O}(n!)$. The Algorithm for finding the number of jobs for each list is $\mathcal{O}(n)$.
Thus overall time complexity for this algorithm is $\mathcal{O}(n * n!)$. Also, we need $\mathcal{O}(n!)$ space to store the permutations.

3.5.3 Algorithm

Let us go over the trivial algorithm

Algorithm 4 Naive Trivial Algorithm

```
1: allPermutations = permute(list) // Note list is the list of jobs we got
2: result = 0
3: for lst in allPermutations do
4:     maxNo = findNoComplete(lst)
5:     result = max(maxNo, result)
6: end for
7: return result
```

Now let us go over the algorithm to find the number of jobs that can be accomplished (findNoComplete).
Remember that this function takes a list as an argument.

3.5.4 Algorithm For Finding the number of jobs that can be Accomplished

Algorithm 5 findNoComplete

```
1: if length of list = 0 then we return 0
2: time = list[0][0] // begin time of first element in list
3: res = 0 // the number of jobs in list we can finish consecutively
4: for lst in list do
5:     B, D, P = lst[0], lst[1], lst[2] // the begin time deadline and end time for the given process
6:     if time + P < D  then
7:         // Note that here the process can be completed within the given time
8:         res = res + 1 time += P
9:     end if
10: end for
11: return res
```

3.5.5 Pros and Cons of this algorithm

This algorithm involves very little mental effort to code. However, it takes $\mathcal{O}(n * n!)$ time which is bad. Also it takes $\mathcal{O}(n!)$ space too. We can do much better.

3.6 Implementing a Streaming Algorithm

In this section, we will go over how we implement the streaming algorithm.
We have a timer variable that keeps track of the current time. We will go over the list (Which is sorted when preprocessed). We also have a list(toProcess) that contains all the elements that can be processed.

- First we loop over till the time < the last elements begin time (which is the last begin time since its sorted) -
  1. While the first element in the lst has begin time which is in between the previous begin time and the current time and has a deadline after the current time, pop the first element and add to toProcess.
  2. Find the best item that is to be processed using another algorithm. Add the processing time to time
  3. Remove all elements that have a deadline before the current time.
- Now that the toProcess list is completed, we just have to finish processing the remaining elements. So we repeat the previous step (processing and updating time). We do this until the toProcess list has no elements.
- We can implement the same algorithm to the trivial algorithm to make it a streaming capable algorithm.

3.7 An Improved Algorithm

Now let us try to find a much better algorithm for the problem. With a little bit of effort, we can do much better than $\mathcal{O}(n * n!)$ time!

3.7.1 Logic

Instead of looking at all possibilities, a simple observation can get us very far.
Observation: If we can complete jobs A and B within the given time, then we can **always** process them in increasing order of deadlines.
This might look obvious but can help us simplify the algorithm a lot. With this, we can sort by the deadlines, and find a sequence of consecutive jobs such that they can all be completed in time. The maximum such pair is the result.
Let us now look at the algorithm

Note This makes an assumption that there will not be a task that is so much more expensive than the others. If so that algorithm won't give the best result (but will give a very close approximation). To see the problem if we have a sequence of jobs 1, 2, 3 (sorted by the deadline). If 2 is computationally very expensive then the best solution will be 1, 3. Our algorithm will not find the solution (but gives a really close result).

3.7.2 Complexity

If we look over the algorithm we see that we sort the list. Sorting is an $\mathcal{O}(nlogn)$ operation. After that, we find the best starting job. This operation takes $\mathcal{O}(n^2)$. So overall the total time complexity is $\mathcal{O}(n^2)$. We also don't use any more space than the space for storage of data, hence it is space efficient.

Algorithm 6 Improved Algorithm (takes input: toProcess - a list and time)

```
sort toProcess based out of deadline
res, min = 0, 0
for i from 0 to toProcess.length do
    cnt, t = 0, time
    for j from i to toProcess.length do
        cur = toProcess[j]
        if time + cur[2] <=cur[1] then
            cnt += 1
            time += cur[2]
        else
            if cnt > min then
                min = cnt
                res = i
            end if
        end if
    end for
end for
return toProcess[res]
```

3.7.3 Naive vs Improved Algo

Lets us now compare the two mentioned algorithms. We already know the time complexity but how do they compare in a real-world test? If you look at figure 1 we see that improved algorithm is much better than the naive algorithm. However, we can do much better. Let's look for a more optimal solution.

3.8 Optimal Algorithm

3.8.1 Logic

Instead of going through each index and iterating over the index, we can use a sliding window approach to shorten the time. We can have a start index and end index. We can move the end pointer until we reach a point such that the tasks cannot be completed within the given time. At this point, we can move the start pointer to the right. We can look at the algorithm below.

3.8.2 Complexity

If we look over the algorithm we see that we sort the list. Sorting is an $\mathcal{O}(nlogn)$ operation. After that our sliding window approach takes $\mathcal{O}(n)$. So overall our time complexity is $\mathcal{O}(nlogn)$. It is much better than our Improved algorithm. We see in Figure two their real-world performance.

3.9 Other Algorithms

There are two other algorithms we can use to solve this problem. We will look at both below and discuss their benefits and drawbacks.

3.9.1 Greedy Algorithm 1

At each part of the process, we can choose to perform the job that takes the minimum processing time that can be computed. This algorithm can be done in $\mathcal{O}(n)$ time.

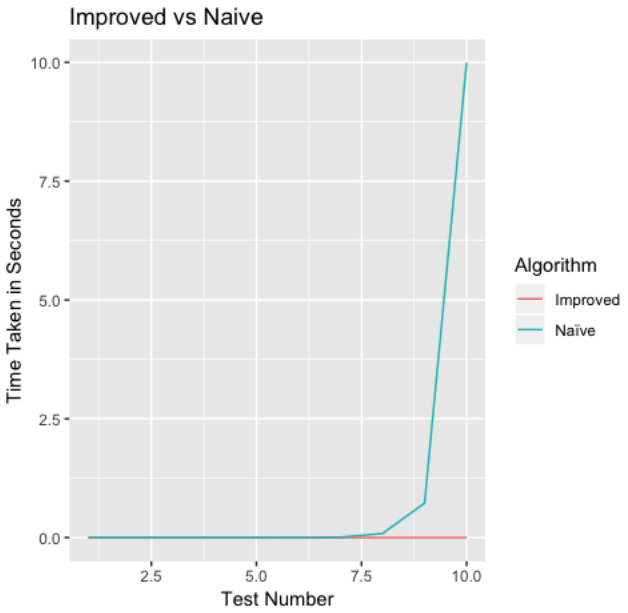

Figure 1: Improved Algo vs Naive Algo

Algorithm 7 Optimal Algorithm (takes input: toProcess - a list and time)

```
1: sort toProcess based out of deadline
2: res, cnt, start, end = 0, 0, 0, 0
3: while start < toProcess.length do
4:     count = 0
5:     while end < toProcess.length do
6:         cur = toProcess[end]
7:         if time + cur[2] <=cur[1] then
8:             count += 1
9:             end += 1
10:            time += cur[2]
11:        else
12:            break
13:        end if
14:    end while
15:    if cnt > cnt then
16:        cnt = count
17:        res = start
18:    end if
19:    time -= toProcess[start][2]
20:    start += 1
21: end while
22: return toProcess[res]
```

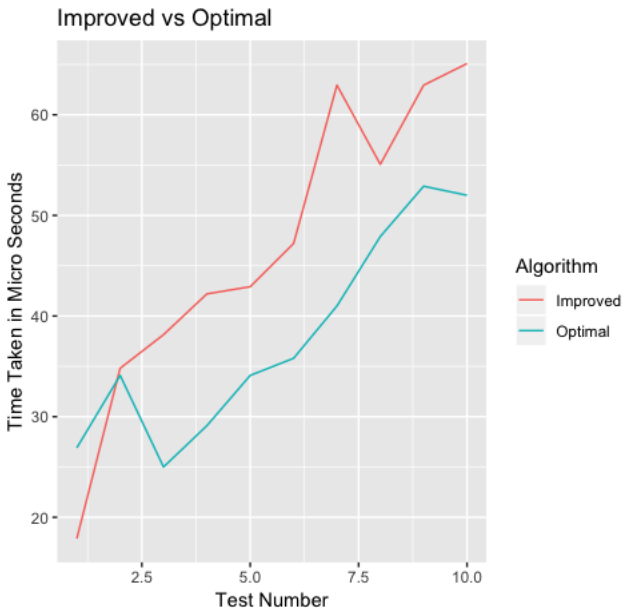

Figure 2: Improved Algo vs Naive Algo

Benifits: We can see that such an algorithm would lead to high efficiency. This can be done if the jobs to be processed is a lot faster than the scheduling algorithm. (We don't want the algorithm to do the scheduling take more time than the process).
Drawbacks: It will not lead to accurate results.

3.9.2 Greedy Algorithm 2

At each part of the process, we can choose to perform the job that has the first deadline. This algorithm can be done in $\mathcal{O}(n)$ time. This has the same benefits and drawbacks as greedy algorithm 1. However, the difference is that we will always choose to compute the task that has to be done first. This has different use cases as Greedy Algorithm 1.

3.10 Summary

We have discussed different solutions to the scheduling problem with equal weights. We have found that

1 The Algorithm can be solved optimally in $\mathcal{O}(nlogn)$ time.

2 There are good greedy solutions that can be done in $\mathcal{O}(n)$ time that has different use cases.

4 Appendix

References

[1] S. Albagli-Kim, B. Schieber, H. Shachnai, and T. Tamir. *Real-Time k-bounded Preemptive Scheduling*, pages 127–137. doi: 10.1137/1.9781611974317.11. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611974317.11.

[2] P. Baptiste. Polynomial time algorithms for minimizing the weighted number of late jobs on a single machine with equal processing times. *Journal of Scheduling*, 2(6):245–252. doi: 10.1002/(SICI)1099-1425(199911/12)2:6⟨245::AID-JOS28⟩3.0.CO;2-5. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291099-1425%28199911/12%292%3A6%3C245%3A%3AAID-JOS28%3E3.0.CO%3B2-5.

[3] P. Baptiste, C. Le Pape, and L. Peridy. Global constraints for partial csps: A case-study of resource and due date constraints. In M. Maher and J.-F. Puget, editors, *Principles and Practice of Constraint Programming — CP98*, pages 87–101, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. ISBN 978-3-540-49481-2.

[4] P. Baptiste, M. Chrobak, C. Dürr, W. Jawor, and N. Vakhania. Preemptive scheduling of equal-length jobs to maximize weighted throughput. *CoRR*, cs.DS/0209033, 2002. URL http://arxiv.org/abs/cs.DS/0209033.

[5] S. Dauzère-Pérès. Minimizing late jobs in the general one machine scheduling problem. *European Journal of Operational Research*, 81:134–142, 02 1995. doi: 10.1016/0377-2217(94)00116-T.

[6] M. C. Dourado, R. d. F. Rodrigues, and J. L. Szwarcfiter. Scheduling unit time jobs with integer release dates to minimize the weighted number of tardy jobs. *Annals of Operations Research*, 169(1):81–91, Jul 2009. ISSN 1572-9338. doi: 10.1007/s10479-008-0479-y. URL https://doi.org/10.1007/s10479-008-0479-y.

[7] H. Kise, T. Ibaraki, and H. Mine. A solvable case of the one-machine scheduling problem with ready and due times. *Oper. Res.*, 26(1):121–126, Feb. 1978. ISSN 0030-364X. doi: 10.1287/opre.26.1.121. URL http://dx.doi.org/10.1287/opre.26.1.121.

[8] E. L. Lawler. A dynamic programming algorithm for preemptive scheduling of a single machine to minimize the number of late jobs. *Annals of Operations Research*, 26(1):125–133, Dec 1990. ISSN 1572-9338. doi: 10.1007/BF02248588. URL https://doi.org/10.1007/BF02248588.

[9] J. M. Moore. An n job, one machine sequencing algorithm for minimizing the number of late jobs. *Management Science*, 15(1):102–109, 1968. doi: 10.1287/mnsc.15.1.102. URL https://doi.org/10.1287/mnsc.15.1.102.

[10] Yun Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No.PR00306)*, pages 328–335, Dec 1999. doi: 10.1109/RTCSA.1999.811269.