Alcom-FT Technical Report Series
ALCOMFT-TR-03-191

Minimizing the number of tardy jobs *

Marjan van den Akker Department of Computer Science
Utrecht University
The Netherlands

Han Hoogeveen

Department of Computer Science
Utrecht University
The Netherlands

December 5, 2003

Abstract

In this chapter we review the main results that have been obtained in the area of minimizing the (weighted) number of tardy jobs. The problem setting here is that a company (which we model as a single machine and as a set of parallel machines, respectively) has been offered a number of jobs. Each job represents a certain amount of work, and when finished implies a certain given reward. The catch is that this reward will be paid only when the job is finished on time. The objective for the firm is of course to maximize the profit.

1 Introduction

In this chapter we consider a problem faced by many students: after having had a great time during the semester spent on other activities, examination time has (suddenly) come close, and hence there is not enough time for thoroughly preparing all of the exams. Since a time-machine has not been invented yet, there is no alternative but to concentrate on a number of courses, such that as many as possible of the exams are passed. The problem is of course: which ones?

Moore (1968) was the first one to study this problem in the context of machine scheduling. In this case, we are given a single machine (corresponding to the student), on which we have to execute a given set of jobs (corresponding to the preparations of the exams). The machine is assumed to be continuously available from time zero onwards, and it can perform at most one job at a time. The machine has to execute n jobs, denoted by $J_1, \ldots, J_n$. Performing task J_j requires a period of uninterrupted length p_j (the (estimated) time to prepare the exam), and the execution of this task is preferably finished by its due date d_j (the start of the examination). If job J_j is finished after its due date, then it is marked as late. The objective is to minimize the (weighted) number of late jobs. In the standard problem considered by

*This work was partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

Moore it does not matter at which time a late job is finished, late jobs are skipped altogether, and the machine only carries out the jobs that will finish on time.

Moore shows that the unweighted case, in which each job is equally important, can be solved in $O(n \log n)$ time by an algorithm that since then is known as Moore-Hodgson's algorithm. The weighted case, where w_j expresses the importance or the reward of executing job J_j ($j = 1, \ldots, n$), can be solved in $O(n \sum p_j)$ time by dynamic programming (Lawler and Moore, 1969). Karp (1972) shows that pseudo-polynomial running time is unavoidable for this problem (unless $\mathcal{P} = \mathcal{NP}$) by establishing $\mathcal{NP}$-hardness in the ordinary sense, even if all due dates are equal.

Throughout this paper we use the three-field notation scheme by Graham, Lawler, Lenstra, and Rinnooy Kan (1979) to denote the problems. We further use the standard scheduling notation: for each job J_j (or simply job j), we use p_j to denote the processing time, d_j to denote the due date of J_j, w_j to denote the weight (if no weights have been specified, then we put all weights equal to 1), r_j to denote the release date of job j (if not specified, then we assume that $r_j = 0$, at which time the machine becomes available), $\bar{d}_j$ to denote the deadline of job j (if this has not been specified, then $\bar{d}_j = +\infty$), and $C_j(\sigma)$ to denote the completion time of job j in schedule σ (and we use simply C_j if it is clear to which schedule we are referring). To denote the objective function we introduce the indicator variable U_j, which gets the value 1 if job j is late, that is, if $C_j > d_j$, and 0, otherwise. Hence, the number of late jobs is then equal to $\sum U_j$.

The remainder of the paper is organized as follows. In Section 2 we review the problem of minimizing the number of late jobs. Here we state Moore-Hodgson's algorithm and prove its correctness by interpreting it as a dynamic programming algorithm. In Section 3 we discuss the weighted case. We first show how to solve the problem through dynamic programming, where we present both Lawler and Moore's algorithm, which runs in $O(n \sum p_j)$ time and space, and an extension of the dynamic programming algorithm of Section 2 that runs in $O(n \sum w_j)$ time and space. We then present Karp's $\mathcal{NP}$-hardness proof, which shows that pseudo-polynomial running times are unavoidable. In Section 4 we add release dates, which imply a lower bound on the start time. In case of arbitrary release dates, the resulting problem is $\mathcal{NP}$-hard in the strong sense, but when the release dates and due dates are similarly ordered, then the problem is solvable in $O(n \log n)$ time by an algorithm due to Lawler (-). In Section 5, we introduce deadlines, which decree an upper bound on the completion time; in this case, we assume that the machine must execute the late jobs as well. We present Lawler's proof (Lawler (-)) that the problem with deadlines is $\mathcal{NP}$-hard in the ordinary sense. In Section 6 we consider the problem with parallel machines and show how to solve this using column generation; this approach is due to Van den Akker, Hoogeveen, and Van de Velde (1999) and Chen and Powell (1999).

2 Minimizing the number of late jobs on a single machine

In this section, we review the problem described by Moore (1968). Using the standard notation we denote this problem by $1||\sum U_j$. We first state Moore-Hodgson's algorithm without a proof, and then solve the problem again from scratch; fortunately, this detour of an $O(n^2)$ dynamic programming algorithm will bring us back to Moore-Hodgson's $O(n \log n)$ algorithm. We developed this approach ourselves, and only found out recently that it has been described in an unpublished paper by Lawler (-) for the more complicated case described in Section 4.

MOORE-HODGSON'S ALGORITHM

STEP 1. Set σ equal to the EDD-schedule for all jobs $J_1, \ldots, J_n$.

STEP 2. If each job in σ is on time, then this subschedule followed by the remaining jobs in any order is optimal.

STEP 3. Find the first job in σ that is late; let this be job J_j. Find the largest job from the set containing J_j and all its predecessors in σ and remove it from σ. Remove the idle time from the resulting schedule by shifting jobs forward; call this new schedule σ, and go to STEP 2.

We apply this algorithm to the following 5-job instance, the data of which are shown in Table 1. Due to the numbering of the jobs, σ is equal to J_1, J_2, J_3, J_4, J_5. The first job in σ

	J_1	J_2	J_3	J_4	J_5
d_j	6	7	8	9	11
p_j	4	3	2	5	6

Table 1: Due dates and processing times

that is late is J_3. The longest job in the subschedule J_1, J_2, J_3 is J_1, which hence is removed. Therefore, σ becomes equal to J_2, J_3, J_4, J_5, and J_4 is the first job in σ that is late. The longest job in J_2, J_3, J_4 is J_4, which gets removed. Now σ is equal to J_2, J_3, J_5, and each job in σ is on time, which means that the solution in which J_2, J_3, J_5 are on time and J_1, J_4 are late is optimal.

The optimality proof of Moore-Hodgson's algorithm is rather complicated. We therefore show its correctness by solving the problem from scratch. We start with some simple, well-known observations.

Observation 2.1 *There exists an optimal solution with the following properties:*

1. *The on time jobs precede all late jobs;*
2. *The on time jobs are executed in EDD-order, that is, in order of nondecreasing due date.*

Proof. The proof of the first part is straightforward: Suppose that an on time job is preceded by a late job. Moving this late job to the end of the schedule and shifting forward all the jobs currently succeeding it will not decrease the quality of the schedule. Repetition of this argument shows the correctness of the first property.

The proof of the second property follows from the result by Jackson (1955), who showed that executing the jobs in EDD-order minimizes the maximum lateness, where the lateness of a job j is defined as $C_j - d_j$ (this maximum lateness is at most equal to zero if all jobs are on time). ∎

Hence, instead of specifying the whole schedule, we can limit ourselves to specifying the set E (for early) containing the on time jobs, as it can be checked whether the jobs in E can be all on time together by putting them in EDD-order. A set E of jobs that are selected to be on time is called *feasible* if none of them is late when executed in EDD-order. The goal is therefore to find a feasible set E of maximum cardinality. From now on, we will use the following notation. By $|Q|$ and $p(Q)$ we denote the number and total processing time of the jobs in a given set Q, respectively. Since the EDD-order is crucial in the design of the algorithm, we assume from now on that the jobs are numbered such that

$$d_1 \leq d_2 \leq \ldots \leq d_n.$$

We want to solve the problem by applying dynamic programming. For this we need the following *dominance rule.*

Dominance Rule 2.2 *Let E^1 and E^2 be two feasible subsets of the tasks $\{J_1, \ldots, J_j\}$ with $|E^1| = |E^2|$. If $p(E^1) < p(E^2)$, then any solution E with $E \cap \{J_1, \ldots, J_j\} = E^2$ can be ignored.*

Proof. Let E correspond to an optimal solution, and suppose that $E \cap \{J_1, \ldots, J_j\} = E^2$. We will show that replacing the jobs in E^2 by the jobs in E^1 yields a feasible subset $\bar{E}$ of $\{J_1, \ldots, J_n\}$; since $|\bar{E}| = |E|$, the subset $\bar{E}$ must then correspond to an optimal solution, too.

To show that $\bar{E}$ is a feasible subset, we must show that in the EDD-schedule of $\bar{E}$ all jobs are on time. Let σ and π denote the EDD-schedule of the jobs in $\bar{E}$ and E, respectively. Due to the numbering of the jobs, we know that the jobs in E^1 precede the remaining jobs of $\bar{E}$ in σ; as E^1 is a feasible subset, the jobs in E^1 are on time in σ. The remaining jobs in $\bar{E}$ start $p(E^2) - p(E^1) > 0$ time units earlier in σ than in π, and hence these jobs are on time as well. ■

As a consequence of the dominance rule, the only feasible subset of $\{J_1, \ldots, J_j\}$ with cardinality k that is of interest is the one with minimum total processing time. We define mc_j (of maximum cardinality), for $j = 1, \ldots, n$, as the maximum cardinality of a feasible subset of the jobs $\{J_1, \ldots, J_j\}$; the value of mc_j will be determined through the algorithm. Since mc_j denotes the maximum number of on time jobs out of the first j, the value of the optimum solution is equal to $n - mc_n$. We further use $E_j^*(k)$ $(j = 1, \ldots, n; k = 0, \ldots, mc_j)$ to denote a feasible subset containing exactly k jobs from $\{J_1, \ldots, J_j\}$ with minimum total processing time. Our dynamic programming algorithm works as follows. We add the jobs one by one in EDD-order. For each combination (j, k), where j $(j = 1, \ldots, n)$ refers to the number of jobs that have been considered and k $(k = 0, \ldots, mc_j)$ denotes the number of on time jobs, we introduce a state-variable $f_j(k)$ with value equal to $p(E_j^*(k))$. As an initialization, we define $mc_0 = 0$ and put $f_j(k) = 0$ if $j = k = 0$ and $f_j(k) = \infty$, otherwise. Suppose that we have determined the values $f_j(k)$ for a given j and all $k = 0, \ldots, mc_j$. We first determine mc_{j+1}. Obviously, mc_{j+1} can only become equal to $mc_j + 1$ if it is possible that J_{j+1} is on time together with mc_j jobs from $\{J_1, \ldots, J_j\}$, and it follows immediately from the proof of Observation 2.1 that to achieve this (if possible) we must pick the subset $E_j^*(mc_j)$ for the mc_j jobs from $\{J_1, \ldots, J_j\}$. Since J_{j+1} succeeds the jobs in $E_j^*(mc_j)$, it can be on time only if $p(E_j^*(mc_j)) + p_{j+1} \leq d_{j+1}$, which condition is equal to $f_j(mc_j) + p_{j+1} \leq d_{j+1}$ by definition of $f_j(k)$. Therefore, we find

$$\begin{array}{ll} mc_{j+1} = mc_j + 1 & \text{if } f_j(mc_j) + p_{j+1} \leq d_{j+1} \\ mc_{j+1} = mc_j & \text{otherwise} \end{array}$$

Now we come to the process of finding the correct values of the state-variables $f_{j+1}(k)$, for $k = 0, \ldots, mc_{j+1}$. We simply put $f_{j+1}(0) = 0$ and determine $f_{j+1}(k)$ $(k = 1, \ldots, mc_{j+1})$ through the recurrence relation

$$f_{j+1}(k) = \min\{f_j(k), f_j(k-1) + p_{j+1}\}. \quad (1)$$

Assuming the correctness of our approach (which we shall show below), we can compute the values $f_j(k)$ in $O(n^2)$ time altogether, from which we immediately determine the minimum number of late jobs as $(n - mc_n)$, whereas the optimum schedule can be determined through backtracking. If we apply it to the example in Table 1, we find the values displayed in Table 2 (in which we have omitted the lines for $k = 4$ and $k = 5$, since these have ∞ in each field). From Table 2 we find that $mc_5 = 3$, which implies that in the optimum solution two jobs are

	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
$k=0$	0	0	0	0	0	0
$k=1$	∞	4	3	2	2	2
$k=2$	∞	∞	7	5	5	5
$k=3$	∞	∞	∞	∞	∞	11

Table 2: Result of the dynamic programming algorithm

late; through backtracking, we find that the jobs J_2, J_3, J_5 are on time and the jobs J_1, J_4 are late. What results is to show the correctness of the dynamic programming algorithm.

Lemma 2.3 *The values $f_j(k)$ $(j = 1, \ldots, n; k = 0, \ldots, n)$ computed through the recurrence relation are equal to $p(E_j^*(k))$.*

Proof. Because of the initialization, we find $f_j(k) = \infty$ if $k > mc_j$, which is the correct value by definition. We use induction to prove the correctness for all j and $k \leq mc_j$. For $j = 0$ and $k = 0$, correctness holds because of the initialization. Suppose that the values $f_j(k)$ are correct for all $j = 0, \ldots, m$ and $k = 0, \ldots, mc_m$ for some value $m \geq 0$. We will show correctness of $f_{m+1}(k)$, where k is any value from $\{1, \ldots, mc_{m+1}\}$ (the case $k = 0$ being trivial). If $k \leq mc_m$, then there are two possibilities: we add job J_{m+1} to the set of on time jobs, or we mark it as late. In the latter case, we find a minimum total processing time of $f_m(k)$. If we include J_{m+1} in the set of on time jobs, then we have to select a feasible subset from the jobs $\{J_1, \ldots, J_m\}$ containing $k - 1$ jobs, and we know that the best one has total processing time $f_m(k-1)$. Job J_{m+1} does not necessarily have to be on time when placed after this subset of $k-1$ jobs, but it will be if $f_m(k-1) + p_{m+1} < f_m(k)$, since $f_m(k)$ is equal to the completion time of the last job in the set of on time jobs, which amounts to no more than d_m and $d_m \leq d_{m+1}$ by definition. The recurrence relation selects the smaller of these two values, which implies the correctness of $f_{m+1}(k)$.

If $k = mc_{m+1} = mc_m + 1$, then we need to add job J_{m+1} to the set of on time jobs to get the desired number, as we have indicated above. In that case, the check performed when computing mc_{m+1} guarantees that J_{m+1} is on time when it starts at time $f_m(mc_m)$, and the correctness of the value $f_m(mc_m)$ settles the proof of the correctness of $f_{m+1}(mc_{m+1})$. ■

We will show that by looking at the dynamic programming algorithm in the right way we can obtain Moore-Hodgson's algorithm. Here we need the close relation between the sets $E_j^*(k)$ and $E_j^*(k-1)$ expressed in the following lemma.

Lemma 2.4 *A set $E_j^*(k-1)$ is obtained from the set $E_j^*(k)$ by removing a task from $E_j^*(k)$ with largest processing time.*

Proof. Suppose that the lemma does not hold for some combination of j and k. Let J_i be a task in $E_j^*(k)$ with maximum processing time. According to our assumption, $p(E_j^*(k-1)) <$

$p(E_j^*(k)) - p_i$. Determine J_q as the job with maximum due date that belongs to $E_j^*(k)$ but not to $E_j^*(k-1)$. Now consider the subset $\bar{E}_j^*(k) = E_j^*(k-1) \cup \{J_q\}$; we will prove that this subset is feasible, which contradicts the optimality of $E_j^*(k)$, since $p(E_j^*(k)) > p(E_j^*(k-1)) + p_i \geq p(E_j^*(k-1)) + p_q = p(\bar{E}_j^*(k))$. We have to check the EDD-schedule, which we denote by σ, for the tasks in $\bar{E}_j^*(k)$, which is obtained from the EDD-schedule for the jobs in $E_j^*(k-1)$ by inserting J_q at the correct spot. Due to the feasibility of $E_j^*(k-1)$, the only jobs that may be late are J_q and the jobs after J_q in $E_j^*(k-1)$. Due to the choice of J_q, we have that J_q and all jobs following J_q in the EDD-schedule of $E_j^*(k)$, which we denote by π, are also present in σ, but they are started at least $p(E_j^*(k)) - p(\bar{E}_j^*(k)) > 0$ time units earlier now, which implies that these jobs are on time in σ as well. The only jobs that still have to be checked are the jobs in σ following J_q that are not present in $E_j^*(k)$. Let J_l be any such job, and let J_k be the last job in σ before J_l that is also present in π; such a job J_k definitely exists, because of the presence of J_q. Since J_k precedes J_l in σ, we have that $d_k \leq d_l$. Moreover, we have that $C_k(\pi) > C_l(\sigma)$, since all jobs following J_k in π are also present in σ and $C_{\max}(\pi) > C_{\max}(\sigma)$. Putting things together, we find that $C_l(\sigma) < C_k(\pi) \leq d_k \leq d_l$, which implies that J_l is on time as well. ∎

As a consequence, if we use a state-variable $g_j(k)$ to denote the set $E_j^*(k)$ (and not just its total processing time), then we can determine $g_j(k)$ from $g_j(mc_j)$, for $k = 0, \ldots, mc_j$, by simply choosing the k smallest jobs. Hence, we then only need to compute and store $g_j(mc_j)$, for $j = 1, \ldots, n$, to find the optimum solution. We can update the recurrence relation correspondingly: If $mc_{j+1} = mc_j + 1$, then we simply put $g_{j+1}(mc_{j+1}) \leftarrow g_j(mc_j) \cup \{J_{j+1}\}$. If on the other hand $mc_{j+1} = mc_j$, then we must choose between the sets $g_j(mc_j)$ and $g_j(mc_j - 1) \cup \{J_{j+1}\}$ as the new set $g_{j+1}(mc_{j+1})$, where the choice depends on the total processing time of the two sets. Because of Lemma 2.4, we know that these two sets both contain $g_j(mc_j - 1)$ as a subset, which implies that there is only one job that makes the difference. Therefore, we have to compare the processing time of the longest job in $g_j(mc_j)$ to p_{j+1}. But this boils down to finding the largest job, say J_i, in the set $E_j^*(mc_j) \cup \{J_{j+1}\}$ and setting $E_{j+1}^*(mc_{j+1})$ equal to $E_j^*(mc_j) \cup \{J_{j+1}\} \setminus \{J_i\}$. Summarizing, we get the following:

- If $p(E_j^*(mc_j)) + p_{j+1} \leq d_{j+1}$, then $E_{j+1}^*(mc_{j+1}) \leftarrow E_j^*(mc_j) \cup \{J_{j+1}\}$.
- If $p(E_j^*(mc_j)) + p_{j+1} > d_{j+1}$, then find the longest job in $E_j^*(mc_j) \cup \{J_{j+1}\}$ and set $E_{j+1}^*(mc_{j+1})$ equal to $E_j^*(mc_j) \cup \{J_{j+1}\}$ minus this job.

But this is exactly Moore-Hodgson's algorithm, which we defined above. Hence, we have proven the following theorem.

Theorem 2.5 *Moore-Hodgson's algorithm solves the problem* $1||\sum U_j$. ∎

We want to conclude this section by discussing a special case of the problem $1||\sum U_j$ in which the job set has been partitioned into 'important' and 'less important' jobs; the objective is to complete as many 'important' jobs as possible on time, and in addition to have as many jobs as possible from the 'less important' category on time.

To solve this variant we apply dynamic programming like described above. In this case we need state-variables that measure the number of jobs from the first and the number of jobs from the second category out of the set $\{J_1, \ldots, J_j\}$ that are on time; the state-variable $f_j(k_1, k_2)$ then has value equal to the minimum total processing time possible for a feasible subset of the jobs $\{J_1, \ldots, J_j\}$ that contains exactly k_1 'important' jobs and k_2 'less important' jobs. This can be implemented in a straightforward manner to run in $O(n^3)$ time.

Note that, if it is feasible to complete all of the 'important' jobs on time, then we can find an optimal solution to the problem above, that is, a solution that completes all 'important' jobs on time and as many of the 'less important' jobs as possible, in $O(n \log n)$ time by adjusting STEP 3 in Moore-Hodgson's algorithm: If J_j is the first tardy job in σ, then remove from the set containing J_j and all its predecessors in σ the longest of the 'less important' jobs. We leave the proof of this statement to the reader.

3 Minimizing the weighted number of tardy jobs

In many applications it is necessary to differentiate between the jobs. This can be done by attaching a positive, integral weight w_j to job J_j, for $j = 1, \ldots, n$, which leads to the problem $1||\sum w_j U_j$. This problem has been studied by Lawler and Moore (1969), who show that it can be solved in $O(n \sum p_j)$ time and space through dynamic programming. The dynamic programming algorithm is based on Observation 2.1; it uses state-variables $f_j(t)$ $(j = 0, \ldots, n; t = 0, \ldots, \sum_{i=1}^{j} p_i)$, where $f_j(t)$ is equal to the value of the optimum solution for the jobs $J_1, \ldots, J_j$ subject to the constraint that the total processing time of the on time jobs amounts to t [1]. The initialization is standard: We put $f_j(t) = 0$ if $j = t = 0$ and ∞, otherwise. Given $f_j(t)$ for $t = 0, 1, \ldots, \sum_{i=1}^{j} p_i$, we compute $f_{j+1}(t)$ $(t = 0, 1, \ldots, \sum_{i=1}^{j+1} p_i)$ through the following recurrence relation

$$f_{j+1}(t) = \begin{cases} \infty & \text{if } t > d_{j+1} \\ \min\{f_j(t) + w_{j+1}, f_j(t - p_{j+1})\} & \text{otherwise.} \end{cases}$$

Assuming that this procedure returns the correct values for the state-variables, which we will show below, we find that $f_n(t)$ $(t = 0, \ldots, \sum_{i=1}^{n} p_i)$ gives the value of the optimum solution subject to the constraint that the total processing time of the on time jobs amounts to t. Therefore, we can find the optimum by determining the minimum of the $f_n(t)$ values over all values of t. Since each of the $O(n \sum p_j)$ state-variables is computed in constant time, the algorithm can be implemented to run in $O(n \sum p_j)$ time and space.

Lemma 3.1 *The state-variables $f_j(t)$ $(j = 0, \ldots, n; t = 0, \ldots, \sum_{i=1}^{j} p_i)$ get the right value.*

Proof. We prove this lemma by induction. Suppose that the values $f_j(t)$ are correct for $t = 0, 1, \ldots, \sum_{i=1}^{j} p_i$; this is obviously true for $j = 0$. We will now show that the recurrence relation assigns the correct values to $f_{j+1}(t)$. First, suppose that $t > d_{j+1}$. In that case, the last job in the on time set completes after its due date, which implies infeasibility, and hence $+\infty$ is the correct value then. If $t \leq d_{j+1}$, then one possibility is to add J_{j+1} to the tardy set, which implies that the interval $[0, t]$ must be filled with jobs from $\{J_1, \ldots, J_j\}$; this gives the first term in the minimand. The other possibility is to put J_{j+1} in the on time set, which implies that only the interval $[0, t - p_{j+1}]$ is left for the jobs in $\{J_1, \ldots, J_j\}$; this leads to the second term in the minimand. Choosing the better one of these two options yields the correct value of $f(j, t)$. ■

[1] In the paper by Lawler and Moore, the total processing time should amount to *no more than* t; the current definition makes the recurrence relation simpler.

The implementation of the dynamic programming algorithm can obviously be improved by restricting the domain of t in $f_j(t)$ to be no more than $\min\{\sum_{i=1}^{j} p_i, d_j\}$, since the remaining $f_j(t)$ values have been initialized at $+\infty$ already.

The dynamic programming algorithm described for the problem $1||\sum U_j$ can be adapted to solve the problem $1||\sum w_j U_j$ in a straightforward manner. Recall that the state-variable $f_j(k)$ $(j = 0, \ldots, n; k = 0, \ldots, j)$ was defined as the minimum total processing time over all feasible subsets of $\{J_1, \ldots, J_j\}$ consisting of exactly k elements. When we change this such that $f_j(k)$ is equal to the minimum total processing time over all feasible subsets of $\{J_1, \ldots, J_j\}$ that have total weight equal to k (which is equal to the cardinality in the unit-weight case), then we can use it to solve the problem $1||\sum w_j U_j$, since the whole analysis of Section 2 goes through. We only need to change the recurrence relation to

$$f_{j+1}(k) = \min\{f_j(k), f_j(k - w_{j+1}) + p_{j+1}\};$$

through this, we can compute the correct values of the state-variables $f_j(k)$ $(j = 0, \ldots, n; k = 0, 1, \ldots, \sum_{i=1}^{j} w_i)$ in constant time. Hence, this dynamic programming algorithm can be implemented to run in $O(n \sum w_i)$ time and space. The value of the optimum solution is then equal to $\sum_{i=1}^{n} w_i - k^*$, where k^* corresponds to the maximum value of k such that $f_n(k) < +\infty$; the corresponding schedule is determined by backtracking.

Both dynamic programming algorithms above have a pseudo-polynomial running time. Karp (1972) showed that this is inevitable, unless $\mathcal{P} = \mathcal{NP}$, by showing that the problem $1||\sum w_j U_j$ is $\mathcal{NP}$-hard in the ordinary sense, even if all due dates are equal; this was presumably the first $\mathcal{NP}$-hardness proof formulated for a scheduling problem. The reduction is from the Partition problem, which is known to be $\mathcal{NP}$-hard in the ordinary sense. Partition is defined as follows:

Partition
Given n positive integers $a_1, a_2, \ldots, a_n$ with sum equal to $2A$, does there exist a subset S of the index set $\{1, \ldots, n\}$ such that

$$\sum_{i \in S} a_i = A?$$

Given any instance of Partition, we construct the following instance of the decision variant of our problem. We let each integer a_i $(i = 1, \ldots, n)$ correspond to a job J_i with processing time $p_i = a_i$, weight $w_i = a_i$, and due date $d_i = A$. The decision variant of the problem of $1||\sum w_j U_j$ is: Does the instance defined above have a solution with value no more than A? We denote this problem by DVWNLJ (Decision Variant of Weighted Number of Late Jobs problem).

Lemma 3.2 *The answer to the instance of* Partition *is 'yes' if and only if the answer to* DVWNLJ *is 'yes'.*

Proof. First suppose that the answer to Partition is 'yes'; let S be a subset of $\{1, \ldots, n\}$ such that $\sum_{i \in S} a_i = A$. It is easily verified that we then can construct a 'yes' solution of DVWNLJ by putting job J_i $(i = 1, \ldots, n)$ in the on time set if $i \in S$ and in the tardy set if $i \notin S$.

Conversely, suppose that we have a 'yes' solution of DVWNLJ; let S be the index set containing the indices of the jobs in the on time set. Since each job has due date equal to

A, we know that $\sum_{i\in S} p_i = \sum_{i\in S} a_i \le A$. We further know that, since the total weight of all jobs equals $2A$ and the total weight of the jobs in the tardy set is no more than A, the total weight of the jobs in the on time set is at least equal to A, that is, $\sum_{i\in S} w_i = \sum_{i\in S} a_i \ge A$. Combining these two inequalities implies that $\sum_{i\in S} a_i = A$, and hence S constitutes a 'yes' solution to Partition. ∎

Theorem 3.3 *The problem $1||\sum w_j U_j$ is $\mathcal{NP}$-hard in the ordinary sense.* ∎

4 Scheduling with release dates

In this section we assume that each job has a *release date*, which we denote by r_j $(j = 1, \dots, n)$; this implies that job J_j $(j = 1, \dots, n)$ cannot be started before time r_j. Note that if no release date has been specified for some job J_j, then we can simply put $r_j = 0$, as the machine is not assumed to be available before this time. Using standard notation, we denote this problem by $1|r_j|\sum U_j$.

Unfortunately, the problem of deciding whether there exists a solution of $1|r_j|\sum U_j$ without tardy jobs is identical to the problem of deciding whether there exists a schedule for this instance with $L_{\max} \le 0$. Since the latter problem is known to be $\mathcal{NP}$-hard in the strong sense (Lenstra, Rinnooy Kan, and Brucker (1977)), the general $1|r_j|\sum U_j$ problem is $\mathcal{NP}$-hard in the strong sense as well.

Kise, Iberaki, and Mine (1978) consider the special case in which the release dates and due dates are similarly ordered, that is, the jobs can be renumbered such that

$$r_1 \le r_2 \le \ldots \le r_n \quad \text{and} \quad d_1 \le d_2 \le \ldots \le d_n.$$

Since this order is essential, we assume that the jobs have been renumbered according to this order. Kise et al. show that this problem can be solved in $O(n^2)$ time. This was improved by Lawler (-), who presented an $O(n \log n)$ algorithm. Here we discuss Lawler's algorithm, which is based on the dynamic programming formulation of Section 2.

Since the release dates and due dates are ordered similarly, Observation 2.1 is still valid. Therefore, given a subset of jobs, we can easily check if this subset is feasible, that is, if all jobs in it can be on time, by putting the jobs in EDD-order. Analogous to our definition of the state-variable $f_j(k)$ in Section 2, we define $f_j(k)$ here as the earliest completion time of the last job in a feasible subset of $\{J_1, \dots, J_j\}$ in which exactly k jobs are on time; because of the release dates, $f_j(k)$ is not necessarily equal to the total processing time of the jobs in this subset. Since the dominance rule of Section 2 remains valid, we can solve this special case of $1|r_j|\sum U_j$ through dynamic programming. The initialization is standard: $f_j(k) = 0$ if $j = k = 0$ and $+\infty$, otherwise. Denoting the maximum cardinality of the on time set of $\{J_1, \dots, J_j\}$ by mc_j again, we find that

$$\begin{array}{ll} mc_{j+1} = mc_j + 1 & \text{if } \max\{f_j(mc_j), r_{j+1}\} + p_{j+1} \le d_{j+1} \\ mc_{j+1} = mc_j & \text{otherwise,} \end{array}$$

where we use $mc_0 = 0$. The process of updating the state-variables is essentially the same as in Section 2: We simply put $f_{j+1}(0) = 0$ and determine $f_{j+1}(k)$ $(k = 1, \dots, mc_{j+1})$ through the recurrence relation

$$f_{j+1}(k) = \min\{f_j(k), \max\{f_j(k-1), r_{j+1}\} + p_{j+1}\}.$$

The correctness proof is straightforward. This algorithm can be implemented to run in $O(n^2)$. To speed it up to run in $O(n \log n)$ time, we have to establish the relation between $f_j(k)$ and $f_j(k-1)$. Let $S_j(k)$ be the set that corresponds to $f_j(k)$. Because of the release dates, we cannot simply remove the longest job from $S_j(k)$ anymore to determine $S_j(k-1)$, but we need a more involved approach. We start with the following well-known lemma.

Lemma 4.1 *For any subset S of $\{J_1, \ldots, J_n\}$ we find the minimum times at which the first j ($j = 1, \ldots, |S|$) jobs have been completed through the preemptive shortest remaining processing time (PSRPT) rule, which is defined as follows: at any time, execute the available job with shortest remaining processing time, where ties are broken by choosing the job with smallest index.*

Proof. This follows by applying an easy interchange argument. ∎

We will need the PSRPT-rule in the following situation. Suppose that we are given two sets of jobs $S \cup \{J_0\}$ and $S \cup \{J_1\}$, where in both schedules determined by the PSRPT-rule for these two job sets all jobs from S are completed before J_0 and J_1, respectively. If we want to choose which job of J_0 and J_1 to add to S such that the resulting set of jobs is completed as soon as possible, then we can decide this by computing the PSRPT-schedule for the set $S \cup \{J_0, J_1\}$ and choosing the job that is not completed last. The correctness of this statement follows from the observation that the jobs in S are completed before the other two jobs and from the fact that the PSRPT-rule completes the first j ($j = 1, \ldots, |S| + 2$) jobs as soon as possible, and hence also the first $|S| + 1$ jobs.

Lemma 4.2 *Let $S_j(k)$ denote the subset of $\{J_1, \ldots, J_j\}$ that corresponds to the state-variable $f_j(k)$ ($j = 1, \ldots, n; k = 0, \ldots, mc_j$). We have that*

$$\emptyset = S_j(0) \subset S_j(1) \subset \ldots \subset S_j(mc_j) \qquad \textit{for } j = 1, \ldots, n.$$

Proof. We prove the lemma by induction. For $j = 1$ it is clearly true. Suppose that it holds for $j = l - 1$; we will now show that it also holds for $j = l$. Consider the dynamic programming algorithm when computing state-variables $f_l(k)$ ($k = 0, \ldots, mc_{l-1}$) in relation to the corresponding sets $S_l(k)$: we have to choose between the sets $S_{l-1}(k)$ and $S_{l-1}(k-1) \cup J_l$. Suppose that k_0 is the smallest value of k for which we decide to include J_l. If we choose to include J_l in all of the sets $S_l(k)$, for $k = k_0 + 1, k_0 + 2, \ldots, mc_l$, then we are done, since in that case we have:

$$S_l(k-1) = S_{l-1}(k-1) \subset S_{l-1}(k) = S_l(k), \quad \forall k = 0, \ldots, k_0 - 1;$$

$$S_l(k_0 - 1) = S_{l-1}(k_0 - 1) \subset S_{l-1}(k_0 - 1) \cup \{J_l\} = S_l(k_0);$$

$$S_l(k-1) = S_{l-1}(k-2) \cup \{J_l\} \subset S_{l-1}(k-1) \cup \{J_l\} = S_l(k), \quad \forall k = k_0 + 1, \ldots, mc_{l-1},$$

and finally, if $mc_l = mc_{l-1} + 1$, then we find that $S_l(mc_l) = S_l(mc_l - 1) \cup \{J_l\}$. The crucial part is hence to show that J_l is put into all sets $S_l(k)$, for $k = k_0 + 1, k_0 + 2, \ldots, mc_l$. Here, we need Lemma 4.1. As remarked above, the choice between the sets $S_{l-1}(k)$ and $S_{l-1}(k-1) \cup J_l$ for any relevant value of k can be made by applying the PSRPT-rule to the set $S_{l-1}(k) \cup J_l$ and remove the job that is completed last (or, equivalently, keep the k jobs that are completed first). Since all jobs are present at time r_l, the job that will be completed last is the job with largest remaining processing time at time r_l (where ties are broken in favor of a smaller index). When determining $S_l(k_0)$, we see that J_l is not removed, but some other job, say J_h, which implies that the processing time of job J_h remaining at time r_l was more than p_l, because of the tie-breaker. If we subsequently determine $S_l(k_0+1)$ by applying the PSRPT-rule to the set $S_{l-1}(k_0+1) \cup J_l$, then we see that the processing time of J_h remaining at time r_l is the same as it was in the PSRPT-schedule for the set $S_{l-1}(k_0)$, and hence, J_l is preferred to J_h, which implies that J_l will be included (remark that J_h will be completed at the (k_0+1)'th place and the job belonging to $S_{l-1}(k_0+1)$ but not to $S_{l-1}(k_0)$ will be removed). This argument can be repeated to show that J_l will get included in all sets $S_l(k)$, $k = k_0, k_0+1, \ldots, mc_l$. ■

The proof in combination with the PSRPT-rule leads to the following algorithm, which can easily be implemented in $O(n \log n)$ time. We first give the algorithm in pseudo-code, and we discuss it afterwards.

LAWLER'S ALGORITHM
STEP 1. $j \leftarrow 0$; $S \leftarrow \emptyset$.
STEP 2. $j \leftarrow j+1$. If $j = n+1$, then go to STEP 6.
STEP 3. Compute the PSRPT-schedule for the jobs J_i with $i \in S$ for the interval $[0, r_j]$. Determine the processing time p'_i remaining at time r_j for each job J_i with $i \in S$.
STEP 4. Check if $r_j + \sum_{i \in S} p'_i + p_j \leq d_j$. If yes, then $S \leftarrow S \cup \{j\}$, and go to STEP 2.
STEP 5. Determine $h \leftarrow \arg\max_{i \in S} p'_i$; in case of a tie, choose h as large as possible. If $p'_h > p_j$, then $S \leftarrow S \cup \{j\} \backslash \{h\}$. Go to STEP 2.
STEP 6. S contains the set of on time jobs.

The idea behind the algorithm is the following. We add the jobs in EDD-order. The set S denotes the current set of on time jobs. When adding J_j, we check whether we can add it to the on time set; this is done in STEP 4. If not, then we find the job with the largest processing time remaining at time r_j (job J_h in the algorithm) and compare p'_h to p_j. If $p'_h > p_j$, then we replace h by j in the set S; otherwise, we keep S the same (STEP 5). The process of determining the p'_i values for all $i \in S$ takes place in STEP 3. Obviously, it is not necessary to compute the PSRPT-schedule for the jobs in S over and over again, since the part until r_{j-1} is still the same as before (if job J_h is processed somewhere in this schedule and h gets removed from S, then we simply replace it with idle time, since clearly none of the other jobs in S was then available; this does not alter the values of the processing time remaining at time r_{j-1} of the other jobs). If we compute the remaining processing times p'_i ($i \in S$) in an incremental way, then we can implement the algorithm to run in $O(n)$ time after the jobs have been sorted.

We conclude this section by applying the above algorithm to the following 7-job instance, which is due to Lawler (-). The data are shown in Table 3. We use here the incremental way to determine the p'_i values for all jobs J_i with $i \in S$.

	J_1	J_2	J_3	J_4	J_5	J_6	J_7
r_j	0	2	4	5	6	7	8
d_j	4	4	9	9	12	12	13
p_j	3	2	4	1	3	2	1

Table 3: Release dates, due dates, and processing times

In iteration 1, we put $S \leftarrow \{1\}$; $p'_1 = 3$.

In iteration 2, we see that J_1 is executed in the interval $[0, 2]$; hence, p'_1 is reduced to 1. We cannot add J_2 to S, and hence we compare p_2 to p'_1. Since p'_1 is smaller, we keep $S = \{1\}$;

In iteration 3, we compute the PSRPT-schedule for the interval $[2, 4]$. We see that J_1 is completed at time 3, and there are no more jobs in S. Hence, we find $p'_1 \leftarrow 0$. When we add J_3 to S, then it is on time; therefore, we put $S \leftarrow \{1, 3\}$.

In iteration 4, we compute the PSRPT-schedule for the interval $[4, 5]$; we put $p'_3 \leftarrow 3$. There is room to add J_4 to the on time set, and therefore we put $S \leftarrow \{1, 3, 4\}$.

In iteration 5, we compute the PSRPT-schedule for the interval $[5, 6]$; we put $p'_4 \leftarrow 0$. There is room to add J_5 to the on time set; hence, we put $S \leftarrow \{1, 3, 4, 5\}$.

In iteration 6, we compute the PSRPT-schedule for the interval $[6, 7]$; we put $p'_3 \leftarrow 2$ (the tie-breaker in the PSRPT-schedule chooses the job with smallest index). Adding J_6 to the on time set is not feasible. We see that $p'_5 = 3$ is the largest remaining processing time in S, which is more than p_6. Hence, we put $S \leftarrow \{1, 3, 4, 6\}$.

In iteration 7, we compute the PSRPT-schedule for the interval $[7, 8]$; we put $p'_3 \leftarrow 1$. There is room to add J_7 to the on time set; hence, we put $S \leftarrow \{1, 3, 4, 6, 7\}$, which is the solution to this instance.

5 Scheduling with deadlines

In this section we discuss the effect of adding *deadlines* to the problem $1||\sum U_j$. The deadline of job J_j is denoted by $\bar{d}_j$ ($j = 1, \ldots, n$), and it indicates that job J_j must be completed by time $\bar{d}_j$. Obviously, this only makes a difference if we are forced to execute the tardy jobs as well; otherwise, we can just adjust the due date of J_j such that it becomes equal to $\min\{d_j, \bar{d}_j\}$. Using standard notation, we denote this problem by $1|\bar{d}_j|\sum U_j$.

Lawler (-) shows that this problem is $\mathcal{NP}$-hard in the ordinary sense by a reduction from PARTITION; we present this proof below. Since no pseudo-polynomial algorithm is known, this $\mathcal{NP}$-hardness proof does not fully fix the computational complexity of the problem. Another open problem is the complexity of the special case in which the due dates and deadlines are related; an example of such a special case is the problem of minimizing the number of tardy jobs subject to the constraint that the lateness of a tardy job should not exceed a given threshold.

Consider the general instance of PARTITION defined in Section 3. We define the following special instance of $1|\bar{d}_j|\sum U_j$. There are $4n$ jobs, the data of which are found in Table 3. Here $M = 3A$ and

$$P_{i-1} = \sum_{k=1}^{n}(2^{n+k-1} + 2^{k-1}) + \sum_{k=1}^{i-1}(2^{n+k-1} + 2^{k-1}) =$$
$$2^n(2^n - 1) + 2^n - 1 + 2^n(2^{i-1} - 1) + 2^{i-1} - 1$$

The decision variant that we will show $\mathcal{NP}$-hardness of is defined as: Does there exist a feasible schedule for the instance of the problem $1|\bar{d}_j|\sum U_j$ defined above in which there are no more than $2n$ tardy jobs? We will denote this problem as DVNLJWD (Decision Variant of the Number of Late Jobs problem With Deadlines).

The reduction defined above looks rather complicated. The idea behind it is that, for each $i = 1, \ldots, n$, the jobs $\{i, 3n + i\}$ and $\{n + i, 2n + i\}$ will form two pairs, one pair of

j	p_j	d_j	$\bar{d}_j$
i $(i = 1, \ldots, n)$	$2^{i-1}M + a_i$	$(2^i - 1)M + A$	$(P_{i-1} + 2^{i-1})M - A$
$n + i$ $(i = 1, \ldots, n)$	$2^{i-1}M$	$(2^i - 1)M + A$	$(P_{i-1} + 2^{i-1} + 2^{n+i-1})M - A$
$2n + i$ $(i = 1, \ldots, n-1)$	$2^{n+i-1}M$	$(2^{n+i} - 1)M + A$	$(P_{i-1} + 2^{n+i-1})M - A$
$3n$	$2^{2n-1}M$	$(2^{2n} - 1)M - A$	$(P_{n-1} + 2^{2n-1})M - A$
$3n + i$ $(i = 1, \ldots, n-1)$	$2^{n+i-1}M - 2a_i$	$(2^{n+i} - 1)M + A$	$(P_{i-1} + 2^{i-1} + 2^{n+i-1})M - A$
$4n$	$2^{2n-1}M - 2a_n$	$(2^{2n} - 1)M - A$	$(P_{n-1} + 2^{n-1} + 2^{2n-1})M - A$

Table 4: Data for the reduction

which goes in the on time set, whereas the other pair goes into the tardy set. We determine the set $S \subset \{1, \ldots, n\}$ by putting $i \in S$ if the pair $\{i, 3n+i\}$ goes into the on time set, for $i = 1, \ldots, n$. Because of the structure of the due dates and deadlines, a schedule leading to 'yes' for DVNLJWD will then have the following form: first the on time job from the pair $\{J_i, J_{n+i}\}$ $(i = 1, \ldots, n)$ followed by the on time job from the pair $\{J_{2n+i}, J_{3n+i}\}$ $(i = 1, \ldots, n)$; these jobs are followed by the tardy jobs in order of earliest deadline. There are two critical points in this schedule. The first one is the completion time of the on time job from the pair $\{J_n, J_{2n}\}$, which is equal to $(2^n - 1)M + \sum_{i \in S} a_i$, whereas its due date is equal to $(2^n - 1)M + A$; hence, it is on time only if $\sum_{i \in S} a_i \leq A$. The second critical point is the completion time of the on time job from the pair $\{J_{3n}, J_{4n}\}$, which is equal to $(2^{2n} - 1)M - \sum_{i \in S} a_i$. Since the due date of jobs $3n$ and $4n$ is equal to $(2^{2n} - 1)M - A$, and one of these two jobs must be on time, we need $\sum_{i \in S} a_i \geq A$. Combining both inequalities, we find that $\sum_{i \in S} a_i = A$.

Using the description of a 'yes' solution above, we can easily verify that if the subset S of $\{1, \ldots, n\}$ corresponds to a 'yes' solution of PARTITION, then the answer to DVNLJWD is 'yes' as well.

Conversely, suppose that the answer to DVNLJWD is 'yes'. We are done if we are able to show the special structure of this solution, which is that, for each $i = 1, \ldots, n$, the on time set contains, either the jobs $\{i, 3n+i\}$, or the jobs $\{n+i, 2n+i\}$. We start with the following observations.

Observation 5.1 *Any feasible schedule for* DVNLJWD *with no more than* $2n$ *tardy jobs possesses the following properties:*

1. *At most one of the jobs from* $\{J_i, J_{n+i}\}$ *is on time, for each* $i = 1, \ldots, n$.
2. *At most one of the jobs from* $\{J_{2n+i}, J_{3n+i}\}$ *is on time, for each* $i = 1, \ldots, n$.
3. *There are exactly* $2n$ *on time jobs.*
4. *At most one of the jobs from* $\{J_{n+i}, J_{3n+i}\}$ *is on time, for each* $i = 1, \ldots, n$.

Proof. The first property holds, since $p_i + p_{n+i} > 2^i M > (2^i - 1)M + A = d_i = d_{n+i}$ for $i = 1, \ldots, n$, where $M = 3A$. A similar proof shows the second property, and the third property follows immediately from the combination of the first two properties and the assumption that there are at least $2n$ on time jobs. Hence, we have exactly one job from $\{J_i, J_{n+i}\}$ and one job from $\{J_{2n+i}, J_{3n+i}\}$ in the on time set, for each $i = 1, \ldots, n$.

The proof of the fourth property follows from a contradiction. Suppose that for some i both J_{n+i} and J_{3n+i} belong to the on time set. A quick computation shows that in any feasible schedule with exactly $2n$ jobs on time we must first execute the on time jobs, and then the remaining jobs in order of earliest deadline. If we compute the completion time of job J_{2n+i} in such a schedule, then we find that it is more than $P_{i-1}M - 4A + 2^{i-1}M + 2^{n+i-1}M \geq (P_{i-1} + 2^{n+i-1})M + (M - 4A) \geq \bar{d}_{2n+i}$, since $M = 3A$. ■

Property 4 implies that if J_{3n+i} $(i = 1, \ldots, n)$ belongs to the on time set, then J_{n+i} does not belong to the on time set, and hence Properties 1 and 3 imply that J_i belongs to the on time set. What is left to show is that if J_i $(i = 1, \ldots, n)$ belongs to the on time set, then J_{3n+i} belongs to the on time set as well. This we prove in the next lemma.

Lemma 5.2 *Consider any feasible schedule for* DVNLJWD *with* $2n$ *tardy jobs. If* J_i $(i = 1, \ldots, n)$ *belongs to the on time set, then* J_{3n+i} *belongs to the on time set, too.*

Proof. Determine the index set S_0 as the subset of $\{1, \ldots, n\}$ that contains the indices of the jobs J_i $(i = 1, \ldots, n)$ that belong to the on time set. Similarly, determine S_3 such that it contains the indices of the jobs J_{3n+i} $(i = 1, \ldots, n)$ in the on time set. We have shown above that $S_3 \subset S_0$.

Recall that in any 'yes' schedule for DVNLJWD we have to execute first the on time job from the pair $\{J_i, J_{n+i}\}$ $(i = 1, \ldots, n)$, then the on time job from the pair $\{J_{2n+i}, J_{3n+i}\}$ $(i = 1, \ldots, n)$, and finally the tardy jobs in order of earliest deadline. As observed in the description of the idea behind the reduction, we need that $\sum_{i \in S_0} a_i \leq A$ to be sure that the job from the pair $\{J_n, J_{2n}\}$ that belongs to the on time set meets its due date. Similarly, we need that

$$\sum_{i \in S_0} a_i - 2 \sum_{i \in S_3} a_i \leq -A.$$

Adding up the inequality $\sum_{i \in S_0} a_i \leq A$ yields the inequality $\sum_{i \in S_0} a_i \leq \sum_{i \in S_3} a_i$. Since $S_3 \subset S_0$ and all a_i values are nonnegative, this can occur only if $S_0 = S_3$. ■

By now, we have shown that the structure claimed in the sketch of the idea behind the reduction is necessary for a feasible schedule with no more than $2n$ tardy jobs. Therefore, we have proven the following theorem.

Theorem 5.3 *The problem* $1|\bar{d}_j|\sum U_j$ *is* $\mathcal{NP}$*-hard in the ordinary sense.* ■

6 Scheduling on m parallel machines

In this section we consider the problem of minimizing the weighted number of tardy jobs, where we have m parallel, identical machines available for executing the jobs. This implies that we have the additional freedom of deciding which machine is going to execute a job; since the machines are identical, the processing time is independent of the choice of machine. We denote machine i by M_i $(i = 1, \ldots, m)$; as before, each machine is continuously available from time zero onwards and can handle no more than one job at a time. Using standard notation, we denote the problem by $P||\sum w_j U_j$.

This problem can be solved through a generalization of the dynamic programming algorithm of Section 3, since the properties stated in Observation 2.1 remain true for each machine. We need state-variables $f_j(t_1, t_2, \ldots, t_m)$, which have value equal to the value of the optimum solution for the jobs $J_1, \ldots, J_j$ subject to the constraint that the total processing time of the on time jobs on machine M_i amounts to t_i, for $i = 1, \ldots, m$. This leads to an algorithm that is easily implemented to run in time $O(n(\sum p_j)^m)$ and space. Especially the space requirement becomes a problem when m increases.

We describe in this section how this problem can alternatively be solved through the method of *column generation*. It was first applied by Van den Akker, Hoogeveen, and Van de Velde (1999) and Chen and Powell (1999); we describe the method by Van den Akker et al., which is more efficient in the branch-and-bound phase.

The crucial step is to view the problem $P||\sum w_j U_j$ as a *partitioning* problem. Looking at any feasible schedule, we see that each of the m machines executes a subset of the on time jobs in EDD-order, possibly followed by some tardy jobs. Hence, we need to partition the jobs into m subsets of on time jobs, and one subset containing the tardy jobs; we need only one subset of tardy jobs, since these can be divided over the machines in any way, if we need to execute these at all. Obviously, we require each of the m subsets of on time jobs to be *feasible*, that is, all of the jobs in a subset should be on time when executed in EDD-order.

We apply this partitioning idea by formulating the problem as a set covering problem with an exponential number of binary variables, n covering constraints, and a single side constraint. We let each feasible subset correspond to a *machine schedule*, which we define as a string of jobs that can be assigned together to any single machine, such that each job in it is on time when this string of jobs is scheduled in EDD-order. Let a_{js} be a constant that is equal to 1 if job J_j is included in machine schedule s and 0 otherwise. Accordingly, the column $(a_{1s}, \ldots, a_{ns})^T$ represents the jobs in machine schedule s. Instead of minimizing the weighted number of tardy jobs, we look at the equivalent problem of maximizing the weighted number of on time jobs. Accordingly, we define the value c_s of machine scheduling s as the sum of the weights of the jobs included in it, that is, $c_s = \sum_{j=1}^{n} a_{js} w_j$. Let S be the set containing all machine schedules. We introduce variables x_s ($s = 1, \ldots, |S|$) that assume value 1 if machine schedule s is selected and 0 otherwise. The problem is then to select at most m machine schedules, such that each job occurs in at most one machine schedule and such that the total value is maximum. Mathematically, the problem is then to determine values x_s that maximize

$$\sum_{s \in S} c_s x_s$$

subject to

$$\sum_{s \in S} x_s \leq m, \tag{2}$$

$$\sum_{s \in S} a_{js} x_s \leq 1, \text{ for each } j = 1, \ldots, n, \tag{3}$$

$$x_s \in \{0, 1\}, \text{ for each } s \in S. \tag{4}$$

Condition (2) and the integrality conditions (4) ensure that at most m machine schedules are selected. Conditions (3) ensure that each job is executed at most once.

To obtain an upper bound that we can use in a branch-and-bound procedure, we solve the linear programming relaxation of this integer linear programming formulation. We obtain the LP-relaxation by replacing the constraints $x_s \in \{0, 1\}$ with $x_s \geq 0$, as the conditions 3 prohibit x_s from exceeding the value of 1. Because of the large number of eligible machine schedules, which in general is too big to enumerate them all, we use the technique of column generation to solve the LP-relaxation, which method considers the feasible columns implicitly. Starting with a restricted linear programming problem in which only a subset $\bar{S}$ of the variables is available, the column generation method solves the LP-relaxation by adding new columns that may improve the solution value, if the optimal solution has not been determined yet; these new columns are obtained by solving the so-called *pricing problem*.

When we have solved the LP-relaxation for a restricted set of variables, we find the values of the dual variables λ_0, which corresponds to conditions 2, and λ_j $(j = 1, \ldots, n)$, where λ_j corresponds to the jth one of the constraints 3. It is well known from the theory of linear programming (see for example Bazaraa, Jarvis, and Sherali (1990)) that the *reduced cost* of any variable x_s is then equal to

$$c'_s = c_s - \lambda_0 - \sum_{j=1}^{n} \lambda_j a_{js} = \sum_{j=1}^{n} w_j a_{js} - \sum_{j=1}^{n} \lambda_j a_{js} - \lambda_0 = \sum_{j=1}^{n} (w_j - \lambda_j) a_{js} - \lambda_0,$$

and the current solution, in which only a subset of the variables have been taken into account, is guaranteedly optimal only if $c'_s \leq 0$ for each $s \in S$. The pricing problem now boils down to finding the variable x_s with corresponding column $(a_{1s}, \ldots, a_{ns})^T$ for which c'_s is maximum. If this maximum has value smaller than or equal to zero, then we have solved the LP-relaxation to optimality; if it is positive, then we add this variable to the LP-relaxation. Since the term $-\lambda_0$ is an unavoidable constant, we ignore it, and hence we want to find the column $(a_{1s}, \ldots, a_{ns})^T$ that maximizes

$$\sum_{j=1}^{n} (w_j - \lambda_j) a_{js},$$

subject to the constraint that it corresponds to a feasible machine schedule. But this results in a familiar problem: we have to maximize the weighted number of on time jobs, where the weight of job J_j $(j = 1, \ldots, n)$ is equal to $(w_j - \lambda_j)$. We solve this problem through the algorithm of Lawler and Moore (1969) described in Section 3; the alternative algorithm described in Section 3 is not well-suited for this purpose here, since the weights are not integral anymore.

After we have solved the LP-relaxation, we have found a strong upper bound that can be applied in a branch-and-bound procedure if necessary. We have to be careful when choosing a branching strategy in case of column generation: it is well-known that the branching strategy of fixing a variable x_s at either zero or one does not work in combination with column generation, as the pricing algorithm may come up with this column again, even though we fixed the variable at zero. Our initial branching strategy is based upon splitting the set of possible completion times.

Suppose that we have found an optimal solution to the LP-relaxation; let S^* denote the set of columns that have been (partially) selected, that is, for each $s \in S^*$ we have $x_s > 0$. For each machine schedule $s \in S^*$ we compute $C_j(s)$ as the completion time of job J_j $(j = 1, \ldots, n)$ if $a_{js} = 1$. We distinguish between the case that for each job J_j $(j = 1, \ldots, n)$ the completion time $C_j(s)$ is the same for each machine schedule $s \in S^*$ it occurs in, and the opposite case. We first address the latter case. Suppose that job J_j is a so-called *fractional* job, that is, a job for which $C_j(s)$ is not the same for each $s \in S^*$. We further define

$$\bar{C}_j \leftarrow \min\{C_j(s) \,|\, s \in S^*\}.$$

We design a binary branch-and-bound tree for which in each node we first identify the fractional job with smallest index, and, if any, then create two descendant nodes: one for the condition that $C_j \leq \bar{C}_j$ and one for the condition that $C_j \geq \bar{C}_j + 1$. The first condition essentially specifies a deadline for J_j by which it must be completed. The second condition specifies a release date $\bar{C}_j + 1 - p_j$ before which J_j cannot be started. Given such a deadline or release date, we first separate the current set of columns accordingly. When we need to solve the pricing problem in a node for which deadlines and/or release dates have been specified, we can incorporate these bounds in the pricing algorithm. For example, when we are given the additional constraint that $C_j \leq \bar{C}_j$, then we have to find the machine schedule that maximizes the total weight (based on weights $w'_j = (w_j - \lambda_j)$ of the set of on time jobs, where we can only select feasible machine schedules that possess the properties of Observation 2.1 in which job J_j can be on time only if it is completed by time $\bar{C}_j$. As this problem is equivalent to minimizing the total weight of the tardy jobs, we can solve it through Lawler and Moore's algorithm by adjusting the recurrence relation to

$$f_j(t) = \begin{cases} \infty & \text{if } t > d_j \\ \min\{f_{j-1}(t) + w'_j, f_{j-1}(t - p_j)\} & \text{if } t \leq \bar{C}_j \\ f_{j-1}(t) + w'_j & \text{if } t > \bar{C}_j \end{cases}$$

Hence, this branching strategy does not complicate the pricing algorithm, which is a major advantage.

After having solved this problem through branch-and-bound, we find the optimal solution of the LP-relaxation in which the completion time of job J_j $(j = 1, \ldots, n)$ is the same in each column $s \in S^*$ that contains job J_j; since this is an essential property of any integral solution, the value of the LP-relaxation is still an upper bound, and if this solution is integral, then we have found an optimum solution. If it is not integral, but we have that

$$\sum_{s \in S^*} a_{js} x_s \in \{0, 1\}, \text{ for each } j = 1, \ldots, n,$$

then we can find an optimal solution in the following way. We put all jobs J_j with $\sum_{s \in S^*} a_{js} x_s = 1$ in the on time set, which we denote by E and all the remaining jobs in the tardy set T. We construct a schedule σ for the on time jobs by executing each of the jobs $J_j \in E$ in the interval $[C_j - p_j, C_j]$, where C_j is the common completion time in the columns in S^* containing J_j. This leads to the following theorem.

Theorem 6.1 *Suppose that the fractional solution satisfies the following properties*

1. *For each job j $(j = 1, \ldots, n)$, we either have $\sum_{s \in S^*} a_{js} x_s = 1$, in which case we put $j \in E$, or $\sum_{s \in S^*} a_{js} x_s = 0$, in which case we put $j \in T$;*

2. *For each $j \in E$, we have that the completion time of job J_j is equal to C_j in each machine schedule $s \in S^*$ containing J_j.*

Then the schedule σ obtained by processing J_j in the time interval $[C_j - p_j, C_j]$ for all $j \in E$ is feasible and has value equal to the value of the fraction solution.

Proof. The schedule σ in which job J_j $(j \in E)$ is processed from time $C_j - p_j$ to time C_j is feasible if and only if at most m jobs are processed at the same time and no job starts before time zero. The second condition is obviously satisfied, since the C_j values originate from feasible machine schedules. The first constraint is satisfied if we show that at most m jobs are started at time zero and that the number of jobs started at any point in time $t \in [1, T]$ is no more than the number of jobs completed at that point in time, where T denotes the latest point in time that a job is started. We first prove that at most m jobs are started at time 0. We define the set Z such that it contains the indices of all jobs in σ that are started at time zero; we need to prove that $|Z| \leq m$. Since each job that starts at time zero in any column $s \in S^*$ is included in Z, we know that $\sum_{j \in Z} a_{js} = 1$ for each $s \in S^*$. Hence, we find that

$$|Z| = \sum_{j \in Z} 1 = \sum_{j \in Z} (\sum_{s \in S^*} a_{js} x_s) = \sum_{s \in S^*} (\sum_{j \in Z} a_{js}) x_s = \sum_{s \in S^*} x_s \leq m,$$

because of condition (2). Hence, we know that at most m jobs are started at time zero. As to the final part of the feasibility proof, let $A(t) \subseteq S^*$ be the set of all machine schedules in which at least one job starts at time t; similarly, let $B(t) \subseteq S^*$ be the set of all machine schedules in which at least one job completes at time t. As $C_j(s) = C_j$, for any machine schedule containing J_j, the number of jobs started at time t is equal to $\sum_{s \in A(t)} x_s^*$; similarly, the number of jobs completed at time t is equal to $\sum_{s \in B(t)} x_s^*$. Since each machine schedule s is constructed such that there is no idle time between the jobs, a job in s can start at time t only if some other job in s is completed at time t. Hence, $A(t) \subset B(t)$, which means that the indicated schedule is feasible.

The value of the fractional solution is equal to

$$\sum_{s \in S^*} c_s x_s = \sum_{s \in S^*} (\sum_{j=1}^{n} a_{js} w_j) x_s = \sum_{j=1}^{n} w_j \sum_{s \in S^*} a_{js} x_s = \sum_{j \in E} w_j,$$

where the last inequality follows from the definition of E and T. But this is equal to the weighted number of on time jobs in σ. ■

If for some j we have that $0 < \sum_{s \in S^*} a_{js} x_s < 1$, then we cannot apply the theorem, and we have to branch further. In this case, we create two nodes, where we assume in node 1 that $\sum_{s \in S^*} a_{js} x_s = 0$ and in node 2 we assume that $\sum_{s \in S^*} a_{js} x_s = 1$; this is equivalent to putting J_j tardy (node 1) or on time (node 2). Note that putting J_j into the tardy set simply means that we ignore J_j in this node; putting J_j into the on time set does not burden the pricing problem, and hence the whole procedure goes through. There is one major difference, though, in comparison with the situation before: due to our second branching strategy, we require some jobs to be present in the on time schedule, in which they have to meet a given deadline and possibly a release date (because of the first part of the branching strategy). Therefore, we may run the risk that in our current node no feasible schedule exists for the jobs that have been put in the on time set by the second part of the branching procedure in which they meet the release dates and the deadlines that have been issued by the first part of the branch-and-bound procedure. The problem of finding out if there exists a feasible solution is of course $\mathcal{NP}$-complete. We therefore check a necessary condition for the existence of a feasible solution, where we only need to consider the jobs — if it fails this check, then we may prune the corresponding node and backtrack. This necessary condition proceeds by transforming the feasibility problem of determining whether there exists a feasible schedule that meets all release dates and deadlines into an optimization problem. We replace each deadline $\bar{d}_j$ by a due date d_j of equal value, and consider the problem of minimizing the maximum lateness $L_{\max}$, which is defined as $\max_{1 \leq j \leq n}(C_j - d_j)$. Because $d_j = \bar{d}_j$, the constraint $C_j \leq \bar{d}_j$ is equivalent to the constraint $C_j - d_j \leq 0$. Hence, the answer to the feasibility problem is 'yes' if and only if there exists a feasible schedule with $L_{\max} \leq 0$. The optimization problem is $\mathcal{NP}$-hard, but several good lower bounds that are quickly computed are known (see for instance Vandevelde, Hoogeveen, Hurkens, and Lenstra (2003)).

A second problem that can occur when we include the constraint $\sum_{s \in S^*} a_{js} x_s = 1$ is that we must be more careful about having a subset $\bar{S}$ of the columns that constitutes a feasible solution (we did not have this problem before, since we could just put $x_s = 0$ to find a feasible solution). One way is to work around this problem in the following. We first remove the infeasible columns that are not part of the current solution to the linear programming relaxation. As to the infeasible columns that are currently part of the linear programming solution, we decrease their costs with some big value, say M, such that the current solution value decreases to a value smaller than the incumbent lower bound (which can be the value of any feasible solution, even zero). Using this trick, we can proceed with the column generation algorithm to solve the current instance of the problem, because we have ensured that there exists at least one feasible solution.

References

[1] J.M. van den Akker, J.A. Hoogeveen, and S.L. van de Velde (1999). Parallel machine schedule by column generation. *Operations Research 47*, 862-872.

[2] M.S. Bazaraa, J.J. Jarvis, and H.D. Sherali (1990). *Linear Programming and Network Flows*. Wiley, New York.

[3] Z.-L. Chen and W.B. Powell (1999). Solving parallel machine scheduling problems by column generation. *INFORMS Journal on Computing 11*, 78-94.

[4] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics 5*, 287-326.

[5] J.R. Jackson (1955). Scheduling a production line to minimize maximum tardiness. Research Report 43, Management Science Research Project, University of California, Los Angeles.

[6] H. Kise, T. Iberaki, and H. Mine (1978). A solvable case of the one-machine scheduling problem with ready and due times. *Operations Research 26*, 121-126.

[7] R.M. Karp (1972). Reducibility among combinatorial problems, in: R.E. Miller and J.W. Thatcher (eds.) (1972). *Complexity of Computer Computations*, Plenum Press, New York, pp. 85-103.

[8] E.L. Lawler (-). Scheduling a single machine to minimize the number of late jobs. Unpublished manuscript.

[9] E.L. Lawler and J.M. Moore (1969). A functional equation and its application to resource allocation and sequencing problems. *Management Science 16*, 77-84.

[10] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker (1977). Complexity of machine scheduling problems. *Annals of Discrete Mathematics 1*, 343-362.

[11] J.M. Moore (1968). An n job, one machine sequencing algorithm for minimizing the number of late jobs. *Management Science 15*, 102-109.

[12] A.M.G. Vandevelde, J.A. Hoogeveen, C.A.J. Hurkens, and J.K. Lenstra (2003). Lower bounds for the head-body-tail problem on parallel machines: a computational study of the multiprocessor flow shop. Submitted to *Informs Journal on Computing*.