Parallel machine scheduling by column generation

Eindhoven University
of Technology

Department of Mathematics and Computing Science

Memorandum COSOR 97-06

Parallel machine scheduling by column generation

J.M. van den Akker
J.A. Hoogeveen
S.L. van de Velde

Eindhoven, March 1997
The Netherlands

Parallel machine scheduling by column generation

J.M. van den Akker*
Department of Mathematical Models and Methods
National Aerospace Laboratory NLR
P.O. Box 90502, 1006 BM Amsterdam, The Netherlands
Email address: vdakker@nlr.nl

J.A. Hoogeveen
Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
Email address: slam@win.tue.nl

S.L. van de Velde
Faculty of Mechanical Engineering
University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
Email address: s.l.vandevelde@wb.utwente.nl

September, 1995 — Revision: March 20, 1997

*Supported by Human Capital and Mobility (HCM), grant number: ERBCHBGCT940513

Abstract

Parallel machine scheduling problems concern the scheduling of n jobs on m machines to minimize some function of the job completion times. If preemption is not allowed, then most problems are not only $\mathcal{NP}$-hard, but also very hard from a practical point of view. In this paper, we show that strong and fast linear programming lower bounds can be computed for an important class of machine scheduling problems with additive objective functions. Characteristic of these problems is that on each machine the order of the jobs in the relevant part of the schedule is obtained through some priority rule. To that end, we formulate these parallel machine scheduling problems as a set covering problem with an exponential number of binary variables, n covering constraints, and a single side constraint. We show that the linear programming relaxation can be solved efficiently by column generation, since the pricing problem is solvable in pseudo-polynomial time. We display this approach on the problem of minimizing total weighted completion time on m identical machines. Our computational results show that the lower bound is singularly strong and that the outcome of the linear program is often integral. Moreover, they show that our branch-and-bound algorithm that uses the linear programming lower bound outperforms the previously best algorithm.

1980 Mathematics Subject Classification (Revision 1991): 90B35.
Keywords and Phrases: parallel machine scheduling, set covering formulation, linear programming, column generation, dynamic programming, total weighted completion time.

1 Introduction

Parallel machine scheduling problems concern the scheduling of n jobs on m parallel machines to minimize some function of the job completion times. Problems in which preemption of jobs is not allowed decompose into two subproblems: *assigning* jobs to machines and then *sequencing* the jobs on each machine. We consider the class of problems with additive objective functions that have all jobs in the relevant part of the schedule sequenced according to some priority rule — we refer to it as class (A). The difficult part lies then mainly in the assignment of jobs to machines, because for a given assignment we can find the optimal schedule by sequencing the jobs in the relevant part of the schedule according to the priority rule, after which the non-relevant jobs are scheduled after the other jobs.

Class (A) contains important objective functions like total weighted completion time, for which the whole schedule is relevant, and objective functions like the weighted number of tardy jobs and total weighted late work, for which only the on-time part of the schedule is relevant. All these problems are unary $\mathcal{NP}$-hard but solvable in pseudo-polynomial time for a *fixed* number of machines m by applying the dynamic programming techniques of Rothkopf (1966) and Lawler and Moore (1969). These pseudo-polynomial algorithms, however, are impractical unless $m = 2$ or the processing times are (very) small.

Additive objective functions pose a computational challenge, since it is difficult to compute strong lower bounds. This is nicely witnessed by the research effort that the problem of minimizing the total weighted completion time on m identical parallel machines has attracted since the early days of machine scheduling research; see for instance Eastman, Evan, and Issacs (1964), Elmaghraby and Park (1974), Barnes and Brennan (1977), Sarin, Ahn, and Bishop (1988). Using the notation scheme of Graham, Lawler, Lenstra, and Rinnooy Kan (1979), we refer to this problem as $P||\sum_{j=1}^{n} w_j C_j$ or as $Pm||\sum_{j=1}^{n} w_j C_j$ when the number of machines m is fixed. The lower bounds developed by Webster (1992, 1995), which require pseudo-polynomial time, and Belouadah and Potts (1994) are a big leap forward. Belouadah and Potts also report on the performance of a branch-and-bound algorithm that uses their bound; it is capable of solving instances with up to 20 jobs and 5 machines and 30 jobs on 4 machines.

It is also remarkable that other parallel machine scheduling problems

with additive objective functions have not received any attention yet. For instance, no one ventured at the parallel machine problem of minimizing the weighted number of tardy jobs or total late work, although their single-machine counterparts, which are binary $\mathcal{NP}$-hard, attracted considerable interest; see for instance Potts and Van Wassenhove (1988, 1992) and Hariri, Potts, and Van Wassenhove (1995).

In this paper, we present a methodology that can be used to deal with the problem $P||\sum_{j=1}^{n} w_j C_j$ and the other problems in class (A); we describe the approach in detail for the former problem and indicate how it can be modified to deal with the latter problems. The approach is based on formulating the problem $P||\sum_{j=1}^{n} w_j C_j$ as a set covering problem with an exponential number of binary variables, n covering constraints, and a single side constraint. We then solve the linear programming relaxation of this formulation by a column generation approach that uses an $O(n\sum_{j=1}^{n} p_j)$ algorithm to solve the corresponding pricing problem, where p_j is the processing time of job J_j $(j = 1, \ldots, n)$. Obviously, if the optimal solution for the linear programming relaxation happens to be integral, then we have identified an optimal solution for the problem $P||\sum_{j=1}^{n} w_j C_j$. If not, then we apply a branch-and-bound algorithm to determine an optimal solution. Our computational results show the compelling quality of the linear programming bound, which makes branching often unnecessary, and the superiority of our branch-and-bound algorithm to the algorithms presented before.

When we were conducting this research, Chan, Kaminsky, Muriel, and Simchi-Levi (1995) as well as Chen and Powell (1995) independently proposed and analyzed the column generation approach to this formulation of the problem $P||\sum_{j=1}^{n} w_j C_j$. Chan, Kaminsky, Muriel, and Simchi-Levi emphasized on worst-case performance analysis and probabilistic analysis. They have established two main results. The first one is that the linear programming bound is asymptotically optimal for any number of machines; if $m = 2$, then these values always coincide. Their second main result is that the value of the optimal solution is at most equal to $(1 + \sqrt{2})/2$ times the value of the linear programming bound; this bound is strengthened to 1.04 in case $w_j = p_j$ for all $j = 1, \ldots, n$. Chen and Powell show how the formulation can be obtained by Dantzig Wolfe decomposition. They also propose a branch-and-bound algorithm in which lower bounds are computed by solving the linear programming relaxation through column generation. They do not branch on the completion times, but on the original x_{ij} variables that indi-

cate that job i is processed immediately before job j on some machine. As a consequence, they cannot use the $O(n\sum p_j)$ algorithm to solve the pricing problem after the branching has started, but have to resort to an $O(n^2\sum p_j)$ time algorithm.

This paper is organized as follows. In Section 2, we present the column generation approach for the problem $P||\sum_{j=1}^{n} w_jC_j$. In Section 3, we describe our branch-and-bound algorithm. In Section 4, we report on our computational experiments for randomly generated instances of this problem. In Section 5, we discuss the adaptations of the formulation and the pricing algorithm necessary to apply them to other prominent problems in the class (A). We draw conclusions and point out directions for future research in Section 6.

2 Column generation for $P||\sum_{j=1}^{n} w_jC_j$

2.1 Problem description

There are m identical machines, $M_1,\ldots,M_m$, available for processing n independent jobs, $J_1,\ldots,J_n$. Job J_j $(j=1,\ldots,n)$ has a processing requirement of length p_j and a weight w_j. Each machine is available from time zero onwards and can handle no more than one job at a time. Preemption of jobs is not allowed. A *feasible schedule* is a specification of the job completion times $C_1,\ldots,C_n$ such that no machine processes more than one job at a time and $C_j-p_j\geq 0$. The objective is to find a schedule with minimum total weighted completion time $\sum_{j=1}^{n} w_jC_j$. The problem is $\mathcal{NP}$-hard in the strong sense when the number of machines is part of the problem instance.

We review various basic properties of an optimal schedule that are useful in our column generation approach. First of all, we have the following.

Theorem 1 (Elmaghraby and Park, 1974) *There exists an optimal schedule with the following properties:*

(i) The jobs are processed contiguously from time zero onwards, and no machine is idle before all jobs have been started.

(ii) The last job on any machine is completed between time $H_{\min}=\sum_{j=1}^{n} p_j/m-(m-1)p_{\max}/m$ *and* $H_{\max}=\sum_{j=1}^{n} p_j/m+(m-1)p_{\max}/m$, *where* $p_{\max}=\max_{1\leq j\leq n} p_j$.

(iii) On each machine the jobs are sequenced in order of non-increasing ratios w_j/p_j.

(iv) If $w_j \geq w_k$ and $p_j \leq p_k$, then there exists an optimal schedule in which job J_j is started no later than job J_k.

There exists a dynamic programming algorithm based on the first three observations runs in $O(n(\sum_{j=1}^{n} p_j)^{m-1})$ time and space. Especially the space requirement becomes unmanageable when m increases.

In the remainder of this subsection, we derive a time slot for each job in which it needs to be executed. The time slot of each J_j is specified by a release date r_j, before which it cannot be started, and a deadline $\bar{d}_j$, at which it has to be finished. Initially, we have that $r_j = 0$ and $\bar{d}_j = \sum_{k=1}^{n} p_k/m + (m-1)p_j/m$ — the deadlines follow from property (ii) (Belouadah and Potts, 1994). Using property (iv), we try to derive tighter release dates and deadlines. Reindex the jobs in order of non-increasing ratios w_j/p_j; to avoid trivialities, we assume that $n > m$. We define $\mathcal{P}_j$ as

$$\mathcal{P}_j = \{J_k \,|k < j,\, w_k \geq w_j, p_k \leq p_j\}$$

and $\mathcal{S}_j$ as

$$\mathcal{S}_j = \{J_k \,|k > j,\, w_k \leq w_j, p_k \geq p_j\}.$$

Accordingly, there is an optimal schedule in which all jobs in the set $\mathcal{P}_j$ start no later than job J_j. If $|\mathcal{P}_j| > m-1$, then we may conclude that at least $|\mathcal{P}_j| - m + 1$ jobs belonging to $\mathcal{P}_j$ are completed before the starting time of J_j. Hence, if ρ_j is the sum of processing times of the $|\mathcal{P}_j| - m + 1$ smallest jobs in the set $\mathcal{P}_j$, then the earliest start time of J_j is given by

$$r_j = \lceil \rho_j/m \rceil, \tag{1}$$

where $\lceil x \rceil$ is the smallest integer larger than or equal to x.

In a similar spirit, we try to tighten the deadline of J_j. Since there is an optimal schedule in which J_j starts no later than any job in $\mathcal{S}_j$, we conclude that the amount of work that needs to be done between the start time of J_j and time $H_{\max}$ amounts to at least $\sum_{J_k \in \mathcal{S}_j} p_k + p_j$. Accordingly, we have that J_j can start no later than

$$\delta_j = H_{\max} - \lceil (\sum_{J_k \in \mathcal{S}_j} p_k + p_j)/m \rceil. \tag{2}$$

If $\delta_j + p_j < \bar{d}_j$, then we let $\bar{d}_j = \delta_j + p_j$ be the new deadline of J_j.

2.2 Mathematical formulation and column generation

The $P||\sum_{j=1}^{n} w_j C_j$ problem belongs to the class (A) because of property (iii), which follows directly from Smith's rule for the single-machine version (Smith, 1956). We formulate it as a set covering problem with an exponential number of binary variables, n covering constraints, and a single side constraint. Define a *machine schedule* as a string of jobs that can be assigned together to any single machine. 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. Let $C_j(s)$ be the completion time of job J_j in s; $C_j(s)$ is defined only if $a_{js} = 1$. Note that since the jobs in s appear in order of their indices without any idle time between their execution, we have that $C_j(s) = \sum_{k=1}^{j} a_{ks} p_k$ — remember that we have reindexed the jobs in order of non-increasing ratios w_j/p_j. Hence, the cost c_s of machine schedule s is readily computed as

$$c_s = \sum_{j=1}^{n} w_j C_j(s) = \sum_{j=1}^{n} w_j a_{js} \left[\sum_{k=1}^{j} a_{ks} p_k \right].$$

We call a machine schedule s *feasible* if $r_j + p_j \leq C_j(s) \leq \bar{d}_j$ for each job J_j included in s. Let S be the set containing all feasible 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 m machine schedules, one for each machine, such that together they contain each job exactly once and minimize total cost. Mathematically, the problem is then to determine values x_s that minimize

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

subject to

$$\sum_{s \in S} x_s = m, \tag{3}$$

$$\sum_{s \in S} a_{js} x_s = 1, \text{ for each } j = 1, \ldots, n, \tag{4}$$

$$x_s \in \{0, 1\}, \text{ for each } s \in S. \tag{5}$$

Condition (3) and the integrality conditions (5) ensure that exactly m machine schedules are selected. Conditions (4) ensure that each job is executed exactly once. Note that the equality sign in the conditions (3) and (4) can be changed into 'smaller than' and 'larger than', respectively, without loosing the validity of the formulation.

The number of columns involved in this formulation is equal to $\sum_{k=1}^{n-m+1} \binom{n}{k}$. Neither the set covering problem, nor its linear programming relaxation, which is obtained by replacing conditions (5) by the conditions $x_s \geq 0$ for all $s \in S$, can therefore be solved by a method that first generates all feasible columns explicitly. Instead, we resort to an iterative method that considers the feasible columns implicitly: column generation. 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 linear programming relaxation of the set covering formulation by adding new columns that may decrease the solution value, if the optimal solution has not been determined yet; these new columns are not obtained through enumeration, but through a *pricing algorithm* that solves an optimization problem, which is called the *pricing problem.*

The primary issue is the design of the pricing algorithm, which we discuss in the next subsection. Our pricing algorithm is a dynamic programming algorithm that usually generates more than one column with negative reduced cost. We also discuss the issue of which and how many such columns should be added to $\bar{S}$ per iteration.

In Section 2.4, we design a heuristic to generate the initial set $\bar{S}$. We also discuss an implementation issue like the proper size of the initial set $\bar{S}$. In Section 2.5, we give a step-wise description of our implementation of the column generation algorithm. In Section 2.6, we discuss a special type of fractional solution to the linear programming problem.

2.3 The pricing algorithm

From the theory of linear programming, we know that a solution to a minimization problem is optimal if the *reduced cost* of each variable is non-negative. In our problem, the reduced cost c'_s of any machine schedule s

is given by

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

where λ_0 is the given value of the dual variable corresponding to condition (3) and $\lambda_1, \ldots, \lambda_n$ are the given values of the dual variables corresponding to conditions (4). To test whether the current solution is optimal, we determine if there exists a machine schedule $s \in S$ with negative reduced cost. To that end, we solve the *pricing problem* of finding the machine schedule in S with minimum reduced cost. Since λ_0 is a constant that is included in the reduced cost of each machine schedule, we essentially have to minimize

$$c_s - \sum_{j=1}^{n} \lambda_j a_{js} = \sum_{j=1}^{n} [w_j (\sum_{k=1}^{j} a_{ks} p_k) - \lambda_j] a_{js}$$

subject to the release dates and the deadlines of the jobs. Our algorithm, which we call the pricing algorithm, tests whether a feasible solution to the linear programming relaxation is optimal; if the outcome is negative, then it outputs a set of feasible machine schedules s with $c'_s < 0$ among which the machine schedule with minimum reduced cost.

Our pricing algorithm is based on dynamic programming and uses a forward recursion that exploits the property that on each machine the jobs are sequenced in order of increasing indices; recall that we indexed the jobs in order of non-increasing ratios w_j/p_j. Let $F_j(t)$ denote the minimum reduced cost for all feasible machine schedules that consist of jobs from the set $\{J_1, \ldots, J_j\}$ in which the last job is completed at time t. Furthermore, let $P(j) = \sum_{k=1}^{j} p_k$. For the machine schedule that realizes $F_j(t)$, there are two possibilities: either J_j is not part of it, or the machine schedule contains J_j. As to the first possibility, we must select the best machine schedule with respect to the first $j-1$ jobs that finishes at time t; the value of this solution is $F_{j-1}(t)$. The second possibility is feasible only if $r_j + p_j \leq t \leq \bar{d}_j$. If it is, then we add J_j to the best machine schedule for the first $j-1$ jobs that finishes at time $t - p_j$; the value of this solution is $F_{j-1}(t - p_j) + w_j t - \lambda_j$. The initialization is then

$$F_j(t) = \begin{cases} -\lambda_0, & \text{if } j = 0 \text{ and } t = 0, \\ \infty, & \text{otherwise.} \end{cases}$$

The recursion is then for $j = 1, \ldots, n$, $t = 0, \ldots, \min\{P(j), \max_{1 \leq k \leq j} \bar{d}_k\}$

$$F_j(t) = \begin{cases} \min\{F_{j-1}(t), F_{j-1}(t - p_j) + w_j t - \lambda_j\}, & \text{if } r_j + p_j \leq t \leq \bar{d}_j, \\ F_{j-1}(t), & \text{otherwise.} \end{cases} \quad (6)$$

The optimal solution value is then found as

$$F^* = \min_{H_{\min} \leq t \leq H_{\max}} F_n(t).$$

Accordingly, if $F^* \geq 0$, then the current linear programming solution is optimal. If $F^* < 0$, then it is not, and we need to introduce new columns to the problem. Candidates are associated with those t for which $F_n(t) < 0$; they can be found by backtracing.

An important implementation issue is the number of columns to add to the linear program after having solved the pricing algorithm. The more columns we add per iteration, the fewer linear programs we need to solve — but the more columns we add per iteration, the bigger the linear programs become. An empirically good choice appeared to be adding those three columns that correspond to those three t for which $F_n(t)$ is most negative.

Note that the pricing algorithm requires $O(n \sum_{j=1}^{n} p_j)$ time and space. This means that our column generation approach is not sensible if $m = 2$: the $P2||\sum_{j=1}^{n} w_j C_j$ problem is better solved directly through dynamic programming.

2.4 The randomized list scheduling heuristic

We need a set $\bar{S}$ of initial columns to compute the initial dual variables. For this purpose, we use a simple but fast randomized list scheduling heuristic to generate many different solutions for the $P||\sum_{j=1}^{n} w_j C_j$ problem, followed by an iterative local improvement method for the ten best solutions.

This randomized list scheduling algorithm generates a schedule by assigning the first $n - 1$ jobs, in order of non-increasing ratios w_j/p_j, randomly to the machines, where an earlier available machine has a larger probability of getting the next job on the list. Specifically, the first available machine has a probability of 80% of getting the next job, the second available machine has a probability of 15%, and the third available machine has a probability of 5%. The last job is always assigned to the earliest available machine.

Such a heuristic works very fast, is expected to give a reasonable solution that satisfies property (iii) of Theorem 1, and is likely to give an alternative solution if it is run again. We therefore run the heuristic between 2,000 and 5,000 times, depending on the size of the instance, thus obtaining many alternative solutions. We store the ten best solutions and apply neighborhood search to try and improve on each of these solutions. The neighborhood of a feasible schedule consists of all schedules in which the jobs on each machine are sequenced in order of non-increasing ratios w_j/p_j that can be obtained by the following two types of *changing operations*: moving a job from one machine to another; and swapping two jobs scheduled on two different machines. If there is a better schedule in the neighborhood, then we adopt it as the new schedule. This process is repeated and terminates when no further improvement can be found.

Using this procedure, we obtain a set $\bar{S}$ consisting of $10m$ columns. Note that we cannot guarantee that that $r_j + p_j \leq C_j(s) \leq \bar{d}_j$ for each J_j and $s \in \bar{S}$. This is no problem at all — these time slots are derived only to speed up the pricing algorithm.

2.5 Solving the linear programming relaxation

In this subsection, we give a step-wise description of our iterative column generation algorithm to solve the linear programming relaxation.

COLUMN GENERATION ALGORITHM
Step 1. Run the randomized list scheduling heuristic 2,000 – 5,000 times, store the 10 best solutions, and try to improve these solutions by iterative local improvement. The resulting solutions constitute the initial variable set $\bar{S}$.
Step 2. Solve the linear programming relaxation to obtain the vector of current dual multipliers λ.
Step 3. Use the pricing algorithm to determine if there are any machine schedules s with negative reduced cost.
Step 4. If such machine schedules exist, then add three machine schedules with most negative reduced cost to the set $\bar{S}$ — go to *Step 2.*
Step 5. If no such machine schedule exists, then stop: we have solved the linear programming relaxation to optimality.

Let now x^* denote the optimal solution to the linear programming relax-

ation of the set covering formulation and let S^* denote the set containing all columns s for which $x_s^* > 0$. If x^* is integral, then x^* constitutes an optimal solution for $P||\sum_{j=1}^{n} w_j C_j$ and we are done. If x^* is fractional, then we need in principle to apply branch-and-bound to close the integrality gap and find an optimal integral solution for our problem, unless x^* is a special type of fractional solution that can be converted without much effort into an integral solution of the same objective value. This special type is discussed in the next subsection.

2.6 A special type of fractional solution

In this section, we discuss a special type of fractional solution to the linear programming relaxation, namely the case in which for each job the completion time is equal in each machine schedule in S^* in which it occurs. This special case is less esoteric than it may seem on first sight — it occurred quite often in our computational experiments. The next result is then a powerful tool to convert such a fractional solution of the linear programming relaxation into an integral solution with the same objective solution. The resulting integral solution is then optimal for the original problem.

Theorem 2 *If $C_j(s) = C_j$ for each job J_j $(j = 1, \ldots, n)$ and for each s with $x_s^* > 0$, then the schedule obtained by processing J_j in the time interval $[C_j - p_j, C_j]$ $(j = 1, \ldots, n)$ is feasible and has minimum cost.*

Proof. The schedule in which job J_j $(j = 1, \ldots, n)$ 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. 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

$\sum_{s \in B(t)} x_s^*$. Because of condition (3), we know that at most m jobs are started at time zero. 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. It is readily checked that the condition $C_j(s) = C_j$ implies that the cost of this schedule is equal to the cost of the fractional solution, and hence minimal. □

If x^* is fractional and does not satisfy the conditions of Theorem 2, then we need a branch-and-bound algorithm to find an optimal solution. This algorithm is presented in the next section.

3 The branch-and-bound algorithm

If the optimal solution to the linear program neither is integral, nor satisfies the conditions of Theorem 2, then a branch-and-bound algorithm is required to find an optimal solution. From other applications, we know that the branching strategy of fixing a variable 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 partitioning strategy is based upon splitting the set of possible completion times.

If we have a fractional optimal solution that does not satisfy the conditions of Theorem 2, then there is at least one job J_j for which

$$\sum_{s \in S^*} C_j(s) x_s^* > \min\{C_j(s) \mid x_s^* > 0\};$$

we call such a job J_j a *fractional job.* Our partitioning strategy reflects this property. 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 $C_j \leq \min\{C_j(s) \mid x_s^* > 0\}$ and one for the condition that $C_j \geq \min\{C_j(s) \mid x_s^* > 0\} + 1$. The first condition essentially specifies a new deadline for J_j by which it must be completed, which is smaller than its current deadline $\bar{d}_j$. For the problem corresponding to this descendant node, we have therefore that

$$\bar{d}_j = \min\{C_j(s) \mid x_s^* > 0\}.$$

Since the jobs in $\mathcal{P}_j$ start no later than J_j and $\bar{d}_j - p_j$ is the latest possible start time of J_j, we must have that each $J_k \in \mathcal{P}_j$ is completed by time $\bar{d}_j - p_j + p_k$, which may be smaller than its current deadline $\bar{d}_k$. Hence, we let

$$\bar{d}_k \leftarrow \min\{\bar{d}_k, \bar{d}_j - p_j + p_k\}$$

for each $J_k \in \mathcal{P}_j$.

The second condition specifies a release date $\min\{C_j(s) \mid x_s^* > 0\} + 1 - p_j$ before which J_j cannot be started, which is larger than its current release date r_j. For the problem corresponding to this descendant node, we have therefore that

$$r_j = \min\{C_j(s) \mid x_s^* > 0\} + 1 - p_j.$$

Since the jobs in $\mathcal{S}_j$ start no earlier than J_j, we can possibly increase the release dates of these jobs as well. For each $J_k \in \mathcal{S}_j$, we let

$$r_k \leftarrow \max\{r_k, r_j\}.$$

Hence, this partitioning strategy not only reduces the feasible scheduling interval of J_j — it may also reduce the feasible scheduling intervals of the jobs in $\mathcal{P}_j$ and $\mathcal{S}_j$.

The nice thing of this partitioning strategy is that either type of condition can easily be incorporated in the pricing algorithm without increasing its time or space requirement: we can use exactly the same recursion as before.

It may be so that the instance that corresponds to our current node does not have a feasible solution, that is, there exists no solution for which $r_j + p_j \leq C_j \leq \bar{d}_j$ for each $j = 1, \ldots, n$. The problem of finding out if there exists a feasible solution is of course NP-complete. We therefore check a necessary condition for the existence of a feasible solution — if it fails this check, then we may prune the corresponding node and backtrack.

This necessary condition proceeds by treating each deadline $\bar{d}_j$ as a due date d_j by which J_j *should* be completed (and hence it may be exceeded) and considering the problem of scheduling the current set of jobs with their processing times, release dates and due dates, so as to minimize the maximum lateness $L_{\max} = \max_{1 \leq j \leq n}(C_j - d_j)$. We then compute a lower bound on the optimal solution value of this NP-hard problem. Various lower bound procedure procedures exist, but we use the so-called *set-based* lower bound

by Vandevelde, Hoogeveen, Hurkens, and Lenstra [1997], which can be computed in $O(n^2)$ time.

It may also be that the current set of columns $\bar{S}$, which was formed while solving the linear programming relaxation in the previous node, does not constitute a feasible solution, due to the new release dates and deadlines. We might use a heuristic to try and generate additional columns that satisfy all release dates and deadlines of the current instance — but there is no guarantee of success, since finding even a feasible solution to the problem is an NP-hard problem. If we would pursue this venue, then, if a greedy heuristic does not succeed in finding a feasible solution, we would need to apply an enumerative algorithm.

We work around this problem in the following way. We first remove the infeasible columns that are not part of the current solution to the linear programming relaxation — these columns can be deleted with impunity. As to the infeasible columns that are currently part of the linear programming solution, we increase their costs to some big value, say M, such that the current solution value increases to a value larger than the incumbent upper bound. 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.

If we backtrack, then we reduce the costs of these columns to their true values.

4 Computational experiments

In this section, we report on our computational results for the problem $P||\sum_{j=1}^{n} w_j C_j$. The algorithms were coded in the computer language C; the experiments were conducted on an HP9000/710, which is a 55 mips machine. To solve the linear programs, we used the package CPLEX, version 2.1.

We incorporated some speed-ups in the pricing algorithm to reduce the empirical running time. Since these speed-ups are not of interest for the red line of the paper, we have included the details in Appendix A.

We tested our algorithm on three classes of randomly generated instances:

(i) instances with processing times drawn from the uniform distribution $[1, 10]$ and weights from the uniform distribution $[10, 100]$;

(ii) instances with processing times and weights both drawn from the uniform distribution $[1, 100]$;

(iii) instances with processing times and weights both drawn from the uniform distribution $[10, 20]$.

First, we report on the performance of our algorithm for instances with $n = 20, 30, 40, 50$ jobs and $m = 3, 4, 5$ machines. Later on, we report our experience with larger instances with up 100 jobs and 10 machines. For each combination of n, m, and instance class, we have generated 25 instances. As we will see, the performance of our algorithm increases with the number of machines *for fixed* n, which contrasts with other branch-and-bound algorithms, including the one by Belouadah and Potts (1994). Problems with three machines are the hardest to solve for our algorithm. Recall that problems with $m = 2$ are better solved directly by dynamic programming than by our algorithm.

We divide these instances into 'easy' instances, for which no branching was required, and 'hard' instances, for which the branch-and-bound algorithm needed to be invoked. For each combination of n and m, we report on the number of 'easy' instances out of 25 and the average time to solve the linear programming problem for them. For the 'hard' instances, we report on the *maximum* number of nodes in the branch-and-bound tree, the *maximum* computation time to solve a hard instance, and the *maximum gap* between linear programming solution value and optimal solution value in absolute terms. The maximum percent excess of the optimal solution value over the linear programming solution was always smaller than 0.05% and goes unreported. Finally, we report the number of instances for which the gap is zero.

Tables 1-3 summarize our computational results for the randomly generated instances with up to 50 jobs and 5 machines. The headers of the

columns are:

n	=	number of jobs;
m	=	number of machines;
NB	=	number of instances out of 25 for which branching was not required;
ACT	=	average computation time in seconds for the 'easy' instances;
MNN	=	maximum number of search tree nodes;
MCT	=	maximum computation time in seconds for the 'hard' instances;
MGAP	=	maximum gap between optimal solution value and lower bound;
ILP=LP	=	number of instances out of 25 for which the optimal solution value and lower bound concur.

Table 1 displays our results for instances belonging to class (i), which is used by Belouadah and Potts (1994) as well to test the performance of their branch-and-bound algorithm. They report that their algorithm fails to solve instances within one minute of computation time on a CDC 7600 computer, which is about twice as slow as our machine, if $n = 30$ and $m = 3$ or $m = 4$, and if $n = 20$ and $m = 5$. Note that instances of this size are a piece of cake for our algorithm.

First of all, Table 1 shows the quality of the linear programming lower bound: branching is often not required, particularly for the instances with $n \leq 30$, and the lower bound is tight for 281 instances out of 300. Accordingly, if the linear programming solution is not integral, then it is often a question of finding an integral solution of the same value; this is an important reason that the rule stipulated in Theorem 1 is so useful. If branching is required, then we need few nodes only to find and verify an optimal solution. Table 1 confirms the claim that we made earlier: for *fixed* n, the instances grow easier with increasing m. This is plausible: the more machines involved, the fewer jobs we may expect to appear in the optimal machine schedules, and accordingly the smaller the relevant solution space will be. Also note that for $n = 40$ and $n = 50$ the linear programming solution is more often fractional than for $n \leq 30$, although the corresponding value is as often tight. A likely reason for this phenomenon is that the number of optimal fractional

		'easy' instances		'hard' instances			
n	m	NB	ACT	MNN	MCT	MGAP	ILP=LP
20	3	24	0.57	2	1.17	0	25
20	4	23	0.41	4	0.78	2	24
20	5	24	0.37	2	0.53	0	25
30	3	17	4.26	2	8.72	1	24
30	4	13	2.52	2	5.73	1	24
30	5	19	1.82	3	2.72	0	25
40	3	7	25.90	10	96.04	1	23
40	4	4	10.41	12	24.83	5	20
40	5	10	6.24	10	14.37	2	22
50	3	2	1.88	14	434.34	1	24
50	4	2	36.83	14	138.61	3	21
50	5	2	14.98	6	46.04	2	24

Table 1: Results for randomly generated instances.

solutions grows with n and the chances of 'hitting' an integral one decreases accordingly.

Since the pricing algorithm requires pseudo-polynomial time, we may expect that the performance of our algorithm deteriorates with the size of the processing times of the jobs. Table 2 displays our results for instances belonging to class (ii), where the processing times are drawn from the uniform distribution $[1, 100]$.

Table 2 indicates that these instances are indeed harder to solve. Not only do we need more time to solve the linear programs, which is entirely attributable to the pricing algorithm, but also slightly more search nodes. The extra time effort is modest, however. Note that the maximum gap is slightly bigger in comparison with Table 1. As a whole, the results remain satisfactory.

Finally, we may also expect that instances with fairly homogeneous ratios w_j/p_j are hard to solve as well. After all, if all ratios w_j/p_j are close to each other, then the number of relevant columns will be large. This intuition is confirmed by our computational results for the instances belonging to class (iii).

Indeed, Table 3 shows that the solution to the linear programming prob-

		'easy' instances		'hard' instances			
n	m	NB	ACT	MNN	MCT	MGAP	ILP=LP
20	3	25	0.76	-	-	-	25
20	4	24	0.54	1	0.72	0	25
20	5	24	0.45	4	0.94	4	24
30	3	21	5.22	6	13.07	4	22
30	4	22	3.12	8	6.38	9	24
30	5	22	2.23	8	5.25	10	23
40	3	22	25.71	8	61.26	11	23
40	4	12	10.81	18	55.47	40	16
40	5	17	7.62	12	20.39	14	20
50	3	17	115.48	10	614.05	11	19
50	4	16	44.23	10	90.89	29	20
50	5	14	23.68	16	99.14	15	18

Table 2: Results for instances with large processing times.

lem is seldom integral, although it is more often tight. Apparently, many optimal solutions are fractional. At the same time, it is more difficult to find an integral solution: we need more search nodes than before. This is quite plausible: since the ratios w_j/p_j are close to each other, the position of a job in the final schedule is not so predetermined, and accordingly our branching rule will be less effective.

Finally, we note that our algorithm is able to deal with instances with $n > 50$ as well, as long as the number of machines is not too small. To give an indication of the performance of our algorithm for such instances, we also report on experiments for instances with 60 jobs on 6 machines, 70 jobs on 7 machines, and so on, up to 100 jobs on 10 machines. The results can be found in Table 4. We see that branching is required for more than 90% of these large instances. In contrast, it is required for less than 50% of the smaller instances. Nonetheless, the lower bound value is still tight for a significant majority of instances. Of course, we need more nodes, the maximum gap between the optimal solution value and the linear programming bound is larger, and we need more computing time — but still, instances of even this size are solvable by our algorithm.

It is a subjective feeling, but the column generation approach gives us

		'easy' instances		'hard' instances			
n	m	NB	ACT	MNN	MCT	MGAP	ILP=LP
20	3	20	0.54	4	1.45	0	25
20	4	23	0.33	2	0.66	0	25
20	5	19	0.33	2	0.55	0	25
30	3	12	2.95	7	15.77	0	25
30	4	4	1.79	10	10.20	0	25
30	5	10	1.12	18	6.91	2	23
40	3	1	10.39	20	188.28	1	24
40	4	4	4.07	16	64.07	0	25
40	5	3	2.00	18	35.89	0	25
50	3	3	42.35	34	742.82	3	24
50	4	0	-	48	342.82	1	22
50	5	1	6.23	22	184.72	3	24

Table 3: Results for instances with homogeneous w_j/p_j ratios.

the idea that solving the parallel machine scheduling problem comes down to cracking the linear programming relaxation: once you have done that, you only need a relatively small number of search nodes to find an optimal solution. This would imply that you can really benefit from faster computers. This is opposite to our experience with most standard branch-and-bound algorithms, which may require an enormous number of nodes.

Finally, we report on the performance of our heuristic. We do not differentiate between instance classes. Table 5 reports for each combination of n and m the average gap ($AGAP$) between the best solution value found by our heuristic and the optimal solution value — this gap is expressed as a percentage of the latter. Furthermore, it reports for each combination the number of instances out of 75 for which an optimal solution was found. The computing time of the heuristic is negligible.

Our conclusion is that the heuristic performs quite well — as could be expected actually, since the problem has a lot of structure and we used this in the design of the heuristic. The average gap $AGAP$ is never more than 0.06% — the maximum gap that we found was no more than 0.08%. Note that the performance of the heuristic deteriorates if the size of the instances, measured either by the number of jobs or the number of machines, increases.

		'easy' instances		'hard' instances			
n	m	NB	ACT	MNN	MCT	MGAP	ILP=LP
60	6	1	35.90	30	128.82	1	20
70	7	1	55.35	22	164.68	2	18
80	8	0	-	38	445.76	7	19
90	9	0	-	26	397.86	3	22
100	10	0	-	30	535.91	3	21
60	6	6	52.65	24	214.62	56	11
70	7	4	110.51	32	321.16	39	9
80	8	7	166.88	32	709.89	38	14
90	9	9	277.26	38	891.25	33	14
100	10	1	490.16	62	1842.66	41	7
60	6	1	13.45	78	482.19	2	21
70	7	0	-	68	663.80	4	19
80	8	0	-	112	1036.77	2	19
90	9	0	-	132	1698.96	5	21
100	10	0	-	142	2211.98	4	14

Table 4: Results for large instances from all three instance classes. The top section concerns instance class (i), the middle section instance class (ii), and the bottom section class (iii).

5 Extensions

In this section, we briefly discuss two other types of problems in the class (A) to which the column generation approach applies:

- problems with additive objective functions for which only a part of the schedule is relevant, like the weighted number of tardy jobs and total weighted late work;
- parallel machine problems with non-identical machines.

We indicate the major differences with the approach for the $P||\sum_{j=1}^{n} w_j C_j$ problem only.

n	m	$AGAP$	$\#opt$
20	3	0.018	36
20	4	0.037	25
20	5	0.042	20
30	3	0.013	17
30	4	0.038	4
30	5	0.045	11
40	3	0.017	1
40	4	0.029	4
40	5	0.044	3
50	3	0.016	5
50	4	0.028	0
50	5	0.040	1
60	6	0.044	1
70	7	0.051	0
80	8	0.053	0
90	9	0.058	0
100	10	0.059	0

Table 5: Performance of the heuristic.

5.1 Complementary objectives

The weighted number of late jobs and total weighted late work are two important objective functions in which jobs are not penalized as long as they are completed at or before their due dates. These objective functions are defined as follows. A job is called *late* if $C_j > d_j$, where d_j is the due date of J_j. The weighted number of late jobs is denoted by $\sum_{j=1}^{n} w_j U_j$, where $U_j = 1$ if J_j is late, and $U_j = 0$ otherwise. Total weighted late work is denoted by $\sum_{j=1}^{n} w_j V_j$ where V_j, the late work of J_j, is defined as the portion of work of J_j that is performed after its due date d_j. Accordingly, we have that $V_j = \min\{p_j, \max\{0, C_j - d_j\}\}$. In both cases, the objective functions are to be minimized.

For either problem, there is an optimal schedule in which each machine first performs the on-time jobs in order of non-decreasing due dates and then the late jobs in any sequence (Potts and Van Wassenhove, 1992). The late jobs appear thus in the irrelevant part of the schedule, and in fact it

does not matter if, when, and by what machine the late jobs are executed. These problems are therefore equivalent to *maximizing* $\sum_{j=1}^{n} w_j(1-U_j)$, the weighted number of on-time jobs, and $\sum_{j=1}^{n} w_j(p_j - V_j)$, total weighted on-time work. These problems lend themselves much better for the column generation approach, since the pricing algorithm needs to focus then only on the on-time jobs. These complementary problems can then be formulated as maximizing

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

subject to

$$\sum_{s \in S} a_{js} x_s \leq 1, \text{ for each } j = 1, \ldots, n, \tag{7}$$

and conditions (3) and (5). Note the sense of condition (7): machine schedules contain on-time jobs only. The pricing algorithm maximizes the reduced cost, which means that the initialization of the recursion is different; apart from that, the recursion is essentially the same as the one described in Section 2.3, but a job will be included in a machine schedule only if it is (partially) on-time.

5.2 Non-identical machines

If the machines are not identical, then the processing time of J_j on M_i is p_{ij}, not p_j, for each i and j. Hence, the cost of a machine schedule then depends on the choice of the machine as well, and we may even have that the priority rule is not equal for all machines. This can be easily overcome by associating a different set of machine schedules to each machine. Let $S(i)$ denote the set of feasible machine schedules for machine M_i $(i = 1, \ldots, n)$. We need to adjust the formulation given in Section 2.2 only slightly to accommodate non-identical machine problems. The problem is formulated as minimizing

$$\sum_{i=1}^{m} \sum_{s \in S(i)} c_s x_s$$

subject to

$$\sum_{s \in S(i)} x_s = 1, \text{ for each } i = 1, \ldots, m, \tag{8}$$

$$\sum_{i=1}^{m} \sum_{s \in S(i)} a_{js} x_s = 1, \text{ for each } j = 1, \ldots, n, \tag{9}$$

$$x_s \in \{0, 1\}, \text{ for each } s \in S(i), i = 1, \ldots, m. \tag{10}$$

For the pricing algorithm, we need to perform the recursion m times, one time for each machine separately. Accordingly, the pricing algorithm runs in $O(n \sum_{i=1}^{m} \sum_{j=1}^{n} p_{ij})$ time.

In addition, the partitioning strategy that we proposed in Section 3 does not apply in case of non-identical machines. An effective alternative is available though: simply use a forward partitioning strategy where jobs are assigned to machines; this partitioning strategy can easily be combined with column generation.

6 Conclusion and future research

Column generation algorithms have been shown to be useful for many intractable combinatorial optimization problems; see for an overview Barnhart, Johnson, Nemhauser, Savelsbergh, and Vance (1994). This paper shows that a column generation algorithm is computationally attractive for a certain class of NP-hard parallel machine scheduling problems as well, including such an important problem as minimizing total weighted completion time.

Our column generation algorithm for this problem proceeds by formulating the parallel machine scheduling problem as an integer linear programming problem with $n + 1$ constraints but an exponential number of variables. We solve the linear programming relaxation by column generation, where we use a fast pseudopolynomial algorithm to compute columns with negative reduced cost, if there are any. Our computational results show that the solution value is at least a singularly strong lower bound on the optimal solution value. Furthermore, the solution is often integral, or fractional but transformable into an equivalent integral solution — in both cases, we have solved the scheduling problem to optimality. If it is not integral, then we need a branch-and-bound algorithm to find an optimal solution. In this case, we have a powerful partitioning strategy to close the gap between lower bound and optimal solution value.

Our computational results show that our algorithm is superior to existing algorithms, which run already into trouble solving randomly generated

instances with 20 jobs and 5 machines or 30 jobs and 4 machines. Our algorithm solves instances of this size in no more than 15 seconds on a 55 mips computer. In fact, we have that the smaller the ratio n/m, the easier the instance is for our algorithm. For instance, we can solve instances up to no more than 50 jobs if $m = 3$ — but we can solve instances with up to 100 jobs if $m = 10$.

In conclusion, the column generation approach is an effective and powerful solution methodology for parallel machine scheduling problems. We are now investigating if there are any other classes of NP-hard machine scheduling problems for which this statement holds. Another issue, connected to the problem of minimizing total weighted completion time, concerns the design of a good algorithm for detecting whether a node in the branch-and-bound tree corresponds to a feasible schedule.

Acknowledgement
Part of the research was conducted when the first author visited the Center for Operations Research and Econometrics at the Catholic University of Louvain. The authors thank an anonymous referee for helpful comments on an earlier draft.

References

[1] J.W. Barnes and J.J. Brennan (1977). An improved algorithm for scheduling jobs on identical machines. *AIIE Transactions 9*, 25-31.

[2] C. Barnhart, E.L. Johnson, G.L. Nemhauser, M.W.P. Savelsbergh, and P.H. Vance (1994). *Branch-and-price: column generation for solving huge integer programs*, Report COC-9403, Georgia Institute of Technology, Atlanta.

[3] H. Belouadah and C.N. Potts (1994). Scheduling identical parallel machines to minimize total weighted completion time. *Discrete Applied Mathematics 48*, 201-218.

[4] L.M.A. Chan, P. Kaminsky, A. Muriel, and D. Simchi-Levi (1995). *Machine scheduling, linear programming and list scheduling heuristics*, Working paper, Northwestern University, Chicago.

[5] Z.L. CHEN AND W.B. POWELL (1995). *Solving Parallel Machine Total Weighted Completion Time Problems by Column Generation*, Working paper, Princeton University.

[6] W.L. EASTMAN, S. EVAN, AND I.M. ISSACS (1964). Bounds for the optimal scheduling of N jobs on M processors. *Management Science 11*, 268-279.

[7] S.E. ELMAGHRABY AND S.H. PARK (1974). Scheduling jobs on a number of identical machines. *AIIE Transactions 6*, 1-13.

[8] 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.

[9] A.M.A. HARIRI, C.N. POTTS AND L.N. VAN WASSENHOVE (1995). Single machine scheduling to minimize total weighted late work. *ORSA Journal on Computing 7*, 232-242.

[10] 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.

[11] C.N. POTTS AND L.N. VAN WASSENHOVE (1988). Algorithms for scheduling a single machine to minimize the weighted number of late jobs. *Management Science 34*, 843-858.

[12] C.N. POTTS AND L.N. VAN WASSENHOVE (1992). Single machine scheduling to minimize total late work. *Operations Research 40*, 586-595.

[13] M.H. ROTHKOPF (1966). Scheduling independent tasks on parallel processors. *Management Science 12*, 437-447.

[14] S.C. SARIN, S. AHN, AND A.B. BISHOP (1988). An improved branching scheme for the branch-and-bound procedure of scheduling n jobs on m parallel machines to minimize total weighted flowtime. *International Journal of Production Research 26*, 1183-1191.

[15] W.E. SMITH (1956). Various optimizers for single-stage production. *Naval Research Logistics Quarterly 31*, 325-333.

[16] A.M.G. VANDEVELDE, J.A. HOOGEVEEN, C.A.J. HURKENS, AND J.K. LENSTRA (1997). *Lower bounds for the multiprocessor flow shop,* Manuscript, Department of Mathematics and Computing Science, Eindhoven University of Technology.

[17] S.T. WEBSTER (1992). New bounds for the identical parallel processor weighted flow time problem. *Management Science 38*, 124-136.

[18] S.T. WEBSTER (1995). Weighted flow time bounds for scheduling identical processors. *European Journal of Operational Research 80*, 103-111.

A Implementation of the pricing algorithm

In the previous, we took advantage of the property that there is an optimal schedule in which no machine schedule finishes its last job before time $H_{\min}$. Accordingly, we solved the pricing problem by choosing the machine schedule s with smallest $F_n(t)$ value, where $H_{\min} \leq t \leq H_{\max}$. To satisfy this lower bound on t, we may need to add a job J_j to s, although $w_j C_j(s) - \lambda_j > 0$ and we are minimizing.

If we ignore the above property, then we know that there is a machine schedule s with minimum cost in which all jobs J_j have $w_j C_j(s) < \lambda_j$. We exploit this observation to reduce the empirical running time of the pricing algorithm. Define $\Delta_j = \lceil \lambda_j / w_j \rceil$ for each j $(j = 1, \ldots, n)$; we have that $\Delta_j \geq 1$, since we can discard all jobs J_j with $\lambda_j = 0$. Accordingly, if $\Delta_j \leq \min\{\min_{1 \leq k \leq j} \bar{d}_k, P(j)\}$, then equation (6) can be replaced by

$$F_j(t) = \begin{cases} \min\{F_{j-1}(t), F_{j-1}(t - p_j) + w_j t - \lambda_j\}, & \text{if } r_j + p_j \leq t < \Delta_j - 1, \\ F_{j-1}(t), & \text{otherwise.} \end{cases}$$

Moreover, the optimal solution value is now found as

$$F^* = \min_{0 \leq t \leq H_{\max}} F_n(t).$$

We like to avoid the explicit computation and storage of all values $F_j(t)$ for $t \geq \Delta_j$, since they are all the same. On the other hand, the recurrence relation needs the value $F_j(t)$ when computing $F_{j+1}(t)$ and $F_{j+1}(t+p_{j+1})$, so we need a procedure to retrieve the proper value of $F_j(t)$ when it is needed. Note now that for any $t \geq \Delta_j$ we have that $F_j(t) = F_{j-a(j,t)}(t)$, where $j - a(j,t)$ is the index of the last job before J_j that cannot be discarded from the machine schedule with maximum value beforehand, that is, $a(j,t)$ is equal to the smallest value such that $t \leq \Delta_{j-a(j,t)} - 1$. Hence, we know that we did not exclude $F_{j-a(j,t)}(t)$ from the computation. In the same fashion, we have for any $t \geq \Delta_j + p_j$ that $F_j(t-p_j) = F_{j-b(j,t)}(t-p_j)$, where $j - b(j,t)$ is the index of the last job before J_j that may be included in the machine schedule with maximum value. Therefore, $b(j,t)$ is the smallest value such that $t - p_j \leq \Delta_{j-b(j,t)} - 1$. Accordingly, the recurrence relation is then for $j = 1, \ldots, n$, $t = 0, \ldots, \min\{\Delta_j - 1, P(j), H_{\max}\}$

$$F_j(t) = \min\{F_{j-a(j,t)}(t), F_{j-b(j,t)}(t-p_j) + w_j t - \lambda_j\}. \tag{11}$$

In order to make it work and to gain from this adjustment, we need to establish an efficient procedure to find the $a(j,t)$ and $b(j,t)$ values for all j and the appropriate times t. Note we must have $a(j,0) = 1$ for any j $(j = 1, \ldots, n)$ and $a(j,t) \geq a(j,t-1)$. Hence, when computing $F_j(t)$ we check first if $a(j,t) = a(j,t-1)$; if it is not, then we increase the value $a(j,t)$ in steps of size one until $t \leq \Delta_{j-a(j,t)}$. Accordingly, the computation of $a(j,t)$ requires one check each time we perform computation (11) plus $O(n^2)$ operations altogether to find the values $a(j,t)$ if $a(j,t) \neq a(j,t-1)$. We can design a similar procedure to compute the values $b(j,t)$. Hence, the worst-case running time of the pricing algorithm remains the same. The average running time has been reduced, however, since we have restricted the range of the state variable t for which the recursion needs to be performed.