Discrete Optimization 6 (2009) 10–22

Contents lists available at ScienceDirect

Discrete Optimization

journal homepage: www.elsevier.com/locate/disopt

The one-machine just-in-time scheduling problem with preemption

Yann Hendel [a], Nina Runge [a], Francis Sourd [b,*]

[a] Université Pierre et Marie Curie - LIP6 - 4, place Jussieu - 75252 PARIS CEDEX 05, France
[b] SNCF - Innovation & Recherche - 45, rue de Londres - 75379 PARIS CEDEX 08, France

ARTICLE INFO

Article history:
Received 12 October 2006
Received in revised form 11 August 2008
Accepted 12 August 2008
Available online 6 September 2008

MSC:
90B35
90B30

Keywords:
Single-machine scheduling
Just-in-time
Preemption

ABSTRACT

This paper investigates the notion of preemption in scheduling, with earliness and tardiness penalties. Starting from the observation that the classical cost model where penalties only depend on completion times does not capture the just-in-time philosophy, we introduce a new model where the earliness costs depend on the start times of the jobs. To solve this problem, we propose an efficient representation of dominant schedules, and a polynomial algorithm to compute the best schedule for a given representation. Both a local search algorithm and a branch-and-bound procedure are then derived. Experiments finally show that the gap between our upper bound and the optimum is very small.

1. Introduction

Researchers and practitioners have shown interest in just-in-time scheduling for about two decades. A common idea is to notice that a job that completes either tardily or early in a schedule induces extra costs. We can find in the literature the study of many scheduling problems that aim to minimize an earliness–tardiness criterion. However, while preemptive problems are an important part of the scheduling theory, preemptive earliness–tardiness scheduling problems seem somewhat neglected. This paper focuses on the one-machine problem.

In non-preemptive problems, earliness–tardiness costs are function of the job completion time: a tardiness penalty is due if the job completes after its due date, conversely an earliness penalty is due if it completes before the due date. However, in a scheduling environment where preemption is allowed, such functions may not bring the desired results: indeed, when the job has been started, the goal is to complete it as soon as possible, so that it can be removed from the production line. We also say that the *work-in-process* has to be minimized. If the costs only depend on the completion time of the job, idle time within the execution of the job might not be penalized, and the work-in-process will be large. In order to avoid this problem, we are going to propose a model that attaches the earliness costs to the job start, while tardiness costs remain tied to the completion of the jobs. The idea of having the optimization criterion depending on the job start has been introduced by Hoogeveen and Van De Velde [12]: they tackle a bicriteria problem where one wants to minimize the maximum *promptness*, i.e. the difference between the start of the job and a given *target start time*, and the mean flow time.

In this paper, we adapt the classical one-machine earliness–tardiness problem, to allow preemption and, in order to determine earliness–tardiness costs, we consider that each job has two due dates instead of one: one tied to the start time of the job and the other tied to its completion time. Formally, we consider a set of jobs $\mathcal{J} = \{J_1, \ldots, J_n\}$ which has to be scheduled on a single machine. Each job $J_j \in \mathcal{J}$ has a processing time p_j. We define tardiness costs as $T_j = \max(0,\ C_j - d_j^c)$,

* Corresponding author.
E-mail addresses: Yann.Hendel@gmail.com (Y. Hendel), Nina.Runge@lip6.fr (N. Runge), Francis.Sourd@sncf.fr (F. Sourd).

1572-5286/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.disopt.2008.08.001

where C_j is the completion time of job J_i and d_j^c is the *ideal completion time* (or *due-date*) of job J_j, and earliness costs as $E_j = \max(0,\ d_j^s - S_j)$ where S_j is the start time of job J_j and $d_j^s = d_j^c - p_j$ is the *ideal start time* (that is the target start time in [12]) of job J_j. We want to minimize $\sum_{j=1}^{n}(\alpha_j E_j + \beta_j T_j)$. We call this problem JIT-POMP which stands for *Just-In-Time Preemptive One-Machine Problem*. This problem is NP-hard: if all earliness penalties are nil, there is an optimal schedule without job interruption, so that we have an instance of $1 \parallel \sum w_i T_i$ which is strongly NP-hard [14].

We first make two remarks about this preemptive model. First, the earliness of a job is a function of its *start* time. Indeed, if earliness classically is estimated by the expression $\max(0,\ d_j^c - C_j)$, we can show that there is an optimal schedule in which no job completes early. Indeed, if a job is early, an infinitesimal piece of length ϵ of it can be moved so that it becomes scheduled in the time interval $[d_j^c - \epsilon, d_j^c]$. Therefore, it has no cost at all, which means that the problem is equivalent to the single machine preemptive problem with only weighted tardiness costs. Second, if a job is scheduled without interruption, its cost is equal to its classical earliness–tardiness non-preemptive cost. Therefore, for a given instance, any feasible schedule of the non-preemptive problem (and especially an optimal one) gives an upper bound for JIT-POMP. The non-preemptive problem, namely $1 \parallel \sum_i f_i(C_i)$ with $f_i(C_i) = \max(\alpha_i(d_i - C_i),\ \beta_i(C_i - d_i))$ has been extensively studied. One way to approach this NP-complete problem [8], is to consider a particular polynomial case, the so-called *timing* problem in which the order of the jobs has already been determined. This subproblem is often used as a core sub-routine in a branch and bound procedure [11,16] or in a local search procedure [9,21,7]. Garey et al. [8] have proposed an $O(n \log n)$ algorithm to solve the timing subproblem in the case of unary earliness–tardiness penalties. Several papers [6,19,5,17,15,10] have extended this algorithm, in order to improve its practical efficiency and to adapt it for problems with non-symmetric and even non convex penalties. One generalization that we will use in this paper, is to consider that the cost functions f_i are convex and piecewise linear. From here onwards, the number of segments of f_i will be denoted by $\|f_i\|$ and $s = \sum_i \|f_i\|$. Hendel and Sourd [10] have proposed an $O(s \log n)$ algorithm for this criterion.

In this paper, we follow a similar approach for JIT-POMP. The first point is to find how to represent the set of solutions. A *representation* is a set R of constraints that are to be satisfied by the represented schedule. Therefore, when we consider the constraints of JIT-POMP plus the constraints of R, we may have a set of several possible schedules. We will say that a representation is *efficient* if $|R|$ is polynomial in n and there is a polynomial algorithm (the so-called timing algorithm) that finds the best schedule in JIT-POMP subject to R. A possible representation would be to fix all the start times (that is R contains n constraints of the form "$S_i = a_i$"). However, the resulting problem JIT-POMP subject to R is equivalent to $1|pmtn, r_i| \sum w_j T_j$, which is NP-hard [14]. Therefore, this representation is not efficient. We show in the next section that an order for the start and completion times of the jobs is sufficient to get a polynomial timing problem.

In Section 2, we give a first efficient representation in which the schedule is defined by a sequence of the $2n$ start and completion times, and the timing problem is solved by linear programming. In Section 3, we propose some dominance properties on the sequences and the schedules in order to get a better representation for which the timing algorithm is strongly polynomial. In Section 4, a local search procedure based on our efficient representation is proposed. Finally, Section 5 is devoted to a branch-and-bound procedure that uses both dominant sequences and a dedicated lower bound.

2. A linear programming-based representation

From here onwards, we denote by σ an order for $\{S_1, \ldots, S_n, C_1, \ldots, C_n\}$. S_i denotes the start time of J_i, that is, according to the context, either the start event or the variable of the proposed linear program. C_i similarly denotes the completion time of J_i. The sequence σ is given as a list and corresponds to a representation, as defined in the introduction, by adding the corresponding inequalities between the start and completion time variables. For example, the representation of the sequence $\sigma = (S_1, S_2, S_3, C_1, C_3, C_2)$ is the series of inequalities $S_1 \le S_2 \le S_3 \le C_1 \le C_3 \le C_2$. For each $i \in \{1, \ldots, n\}$, we are going to assume that S_i is before C_i in the sequence.

We now introduce the values $P_\sigma(X, Y)$ that represent the mandatory processing time that occurs between two distinct events $X, Y \in \{S_1, \ldots, S_n, C_1, \ldots, C_n\}$. If Y precedes X in the sequence, we simply set $P_\sigma(X, Y) = -\infty$. Otherwise, $P_\sigma(X, Y)$ is the sum of the processing times of the jobs that must start after X and that must complete before Y. For example, in the previous example, $P_\sigma(S_1, C_3)$ is equal to $p_1 + p_3$ because both J_1 and J_3 must be processed between S_1 and C_3, while J_2 can be processed after C_3. $P_\sigma(X, Y)$ is equal to 0 if no job is constrained to be processed between X and Y. Clearly, all these values only depend on σ, so that they are constant values in the following linear program that solve JIT-POMP subject to σ:

$$(\text{LP}) : \min \sum_{i=1}^{n}(\alpha_i E_i + \beta_i T_i) \tag{1}$$

$$\text{s.t.} \quad E_i \ge d_i^s - S_i, \quad i = 1, \ldots, n \tag{2}$$

$$T_i \ge C_i - d_i^c, \quad i = 1, \ldots, n \tag{3}$$

$$Y - X \ge P_\sigma(X, Y), \quad X, Y \in \{S_1, \ldots, S_n, C_1, \ldots, C_n\} \tag{4}$$

$$E_i, T_i \ge 0 \quad i = 1, \ldots, n. \tag{5}$$

This linear program provides the start and completion times for each job. Inequalities (1)–(4) are clearly necessary conditions that must be satisfied by any preemptive schedule. In order to show that any solution to (LP) corresponds to a feasible

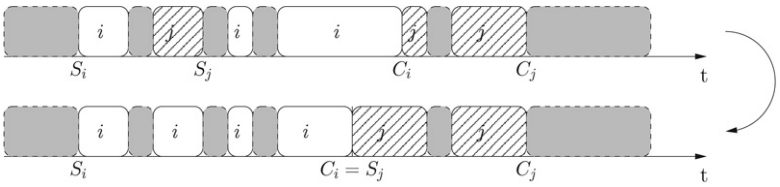

Fig. 1. Proof of Property 1.

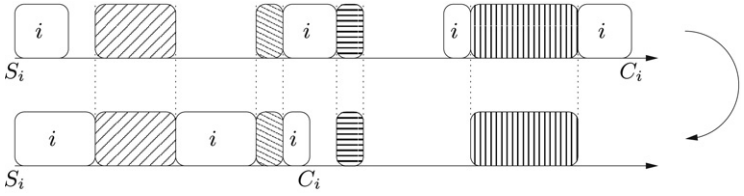

Fig. 2. Proof of Property 2.

schedule, we first observe that the objective function implies $E_i = \max(0, d_i^s - S_i)$ and $T_i = \max(0, C_i - d_i^c)$ in any solution of (LP). The existence of a feasible schedule such that the processing of J_i is done in $[S_i, C_i]$, directly derives from Hall's theorem [3]. To build the schedule, we observe that the start and completion times respectively induce release dates and deadlines; by Jackson's rule we get a feasible schedule with less than $2n$ preemptions.

As for the complexity, (LP) has $O(n)$ variables but $O(n^2)$ constraints. It is therefore polynomial and it can even be solved in strongly polynomial time as Tardos' method [20] can be applied. It can also be solved as special cases of the network problems with separable convex cost functions solved by Karzanov and McCormick [13] and Ahuja et al. [2]. However, in order to pratically solve JIT-POMP, the timing procedure has to be called very often. In the next section, we present a second efficient representation which takes advantage of some dominance rules. The benefits of this second approach are twofold. First, we will show in Section 3.3 that the resulting timing algorithm is faster. Second, we show in the beginning of Section 4 that the size of the solution space is smaller.

3. A combinatorial algorithm for the timing problem

The timing algorithm we propose in this section does not work for all the sequences that satisfy $S_i \leq C_i$, but only for a subclass of sequences which are dominant. This class is defined in Section 3.1 and the algorithm is then presented in Section 3.2. In Section 3.3, we experimentally compare our algorithm to solving (LP) with ILOG CPLEX.

3.1. Valid sequences

We will denote by $P(\sigma)$, the timing problem corresponding to the sequence σ and by $\mathrm{Opt}(\sigma)$ the optimal cost for $P(\sigma)$.

Property 1. *For every sequence σ_1, such that for a pair of jobs (J_i, J_j), $S_i \leq S_j \leq C_i \leq C_j$, then there is a sequence σ_2 such that $S_i \leq C_i \leq S_j \leq C_j$ and $\mathrm{Opt}(\sigma_2) \leq \mathrm{Opt}(\sigma_1)$.*

Proof. We consider the execution intervals of J_i and J_j in a schedule whose cost is $\mathrm{Opt}(\sigma_1)$. The execution intervals of J_i and J_j can be rearranged in such a way that J_i is totally processed before the start of J_j. To do so, we proceed iteratively: we consider the earliest piece of J_j which is executed before the last pieces of J_i. We swap the largest possible amount of the piece of J_j with the latest possible amounts of pieces of J_i (by doing so, another preemption may be added). We perform the same operations on the current schedule until J_i is entirely executed before J_j. Consequently the completion time of J_i is moved backward and the start time of J_j is postponed. Thus the costs induced by the two jobs can only decrease. Fig. 1 illustrates the transformation (the gray rectangles represent other jobs). □

Property 2. *Let us consider a schedule where, for some job J_i, there is some idle time between S_i and C_i, then we can build a schedule with a lower or equal cost such that there is no idle time between S_i and C_i.*

Proof. Consider a schedule where there is some idle time between the start of job J_i and its completion. As illustrated by Fig. 2, the execution intervals of job J_i can be rearranged in order to fill the gaps between the start and the completion of J_i. In that manner, C_i and the cost of the schedule can only decrease. □

For any pair of jobs (J_i, J_j), according to Property 1, we have four possibilities: $S_i \leq S_j \leq C_j \leq C_i$, $S_j \leq S_i \leq C_i \leq C_j$, $S_i \leq C_i \leq S_j \leq C_j$ or $S_j \leq C_j \leq S_i \leq C_i$. In the first two cases, we say that J_j (resp. J_i) is *nested* in J_i (resp. J_j). In the other cases, we say that J_i and J_j are *separated*. Thus, the first property ensures a structure which is analogous to the nested parenthesis

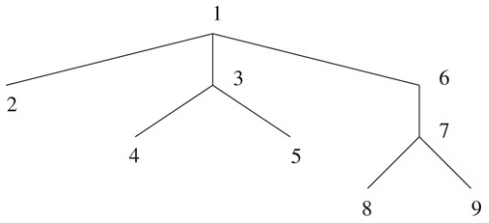

Fig. 3. Tree associated to the first main block of σ_{11}.

structure [1]: for a given job J_j, each job started between S_j and C_j, must be finished before C_j, and is therefore nested in J_j. We call the subsequence associated with each job, a block: we call B_j the block associated with job J_j, it contains all the jobs that are executed within the interval $[S_j, C_j]$. Moreover, according to the second property, there is no idle time in B_j, therefore, we denote by P_j the constant length of this block which is $p_j + \sum_{J_k \in B_j} p_k$. A maximum block which is not nested in any other block is called a *main* block. A main block is a connected component of the inclusion graph of the blocks. From Property 2, there is an optimal schedule where idle time is only between the main blocks.

A sequence with this nested parenthesis structure is said to be *valid*. Below is an example of a valid sequence of 11 jobs:

$$\sigma_{11} = (S_1(S_2, C_2)(S_3(S_4, C_4)(S_5, C_5)C_3)(S_6(S_7(S_8, C_8)(S_9, C_9)C_7)C_6)C_1)(S_{10}(S_{11}, C_{11})C_{10})$$

σ_{11} has two main blocks, B_1 and B_{10}.

A nested parenthesis structure can be represented by a forest. Each main block is represented by a tree. A leaf represents a stand alone job J_j i.e. when C_j is the immediate successor of S_j in the sequence (and then, $C_j - S_j = p_j$). A node r with k sons in the left-right order, represents a job J_r such that S_r (resp. C_r) precedes (resp. follows) k sub-sequences of the original sequence in their left-right order. The number of nodes in the tree corresponds to the number of jobs in the corresponding main block. The tree represented in Fig. 3 corresponds to the first main block of σ_{11}.

3.2. Solving the timing problem

In this section, we solve the timing problem corresponding to a valid sequence. For the sake of simplicity, a dummy job J_0 is introduced in order to represent the idle time. This idle time is lower than $\max_i d_i^c$ (for there is no idle time after the last due date). A new sequence σ_0 is derived from σ by adding S_0 and C_0 respectively before and after the other dates of σ. The processing time of J_0 is $p_0 = \max_i d_i^c$. This job comes at no cost, and it is easy to see that $\mathrm{Opt}(\sigma_0) = \mathrm{Opt}(\sigma)$. This way, σ_0 is made up of a single main block B_0 that starts at $S_0 = 0$ and completes at $C_0 = \sum_{i=0}^{n} p_i$.

Since there is a single main block, a single tree is used to represent σ_0. Our data structure is based on this tree structure and each node r contains the following data:

- P_r the total processing time between S_r and C_r
- $f_r(t)$ the cost function of scheduling block B_r such that $C_r = t$ (and $S_r = t - P_r$).

We are going to prove by induction, that the information stored in each node can be derived from the information stored in the child nodes. Moreover, we also prove by induction, that the cost functions f_i are piecewise linear and convex. First, since a leaf represents a stand-alone job that is executed without preemption, the cost function stored is the classical earliness–tardiness function $f_i(C_i) = \max(\alpha_i(d_i - C_i), \beta_i(C_i - d_i))$, which is piecewise linear and convex. Let us now consider an inner node r and its associated job J_r. We assume that k blocks, denoted by $B_{[1]}, \ldots, B_{[k]}$ are nested in B_r and we denote by $\mathbb{T}_1, \ldots, \mathbb{T}_k$ the subtrees in the left-right order. The tree $\mathbb{T}_i$ holds the cost function of $B_{[i]}$, which is piecewise linear and convex by the induction hypothesis, and the length $P_{[i]}$ of block $B_{[i]}$.

We first have $P_r = \sum_{i=1}^{k} P_{[i]} + p_r$. If we fix C_r, S_r is also fixed because there is no idle time inside the block ($S_r = C_r - P_r$). The blocks $B_{[1]}, \ldots, B_{[k]}$, in this order, have to be scheduled in an optimal manner within the time interval $[S_r, C_r]$, the sum of the idle periods in this interval being p_r. However, to build $f_r(C_r)$, we need to compute the cost of block B_r for every C_r. To achieve this goal, we first compute the optimal schedule when the sub-blocks are not constrained by S_r and C_r (see Fig. 4(a)), then we show that the constrained optimal schedule is derived from this latter schedule.

By the induction hypothesis, the blocks $B_{[1]}, \ldots, B_{[k]}$ have constant execution times and have convex and piecewise linear cost functions. Therefore, they can be viewed as jobs which are executed without preemption, and have convex piecewise linear cost functions. Thus, in order to compute the optimal schedule of these blocks without the constraint that they are between S_r and C_r, we can apply the timing algorithm proposed by Hendel and Sourd [10]. This algorithm solves the timing problem for a sequence of non-interruptible jobs with a convex piecewise linear cost function. The obtained schedule provides a new block decomposition: we denote by $B'_{[1]}, \ldots, B'_{[l]}$ the maximum sets of blocks without idle time. Every block $B'_{[i]}$ has a convex and piecewise linear cost function $f'_{[i]}$ (as a sum of convex piecewise linear cost functions) and a processing time $P'_{[i]}$ (which is the sum of the processing times of the original blocks contained in this new block). We denote by $C'_{[i]}$ its

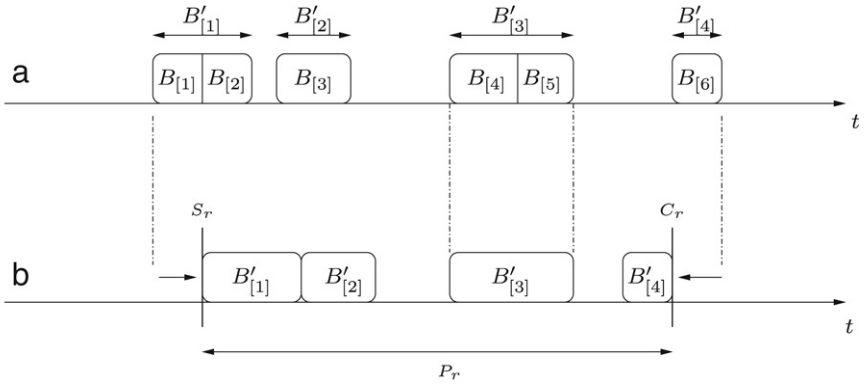

Fig. 4. Optimal schedules without a completion time constraint, next constrained by S_r and C_r.

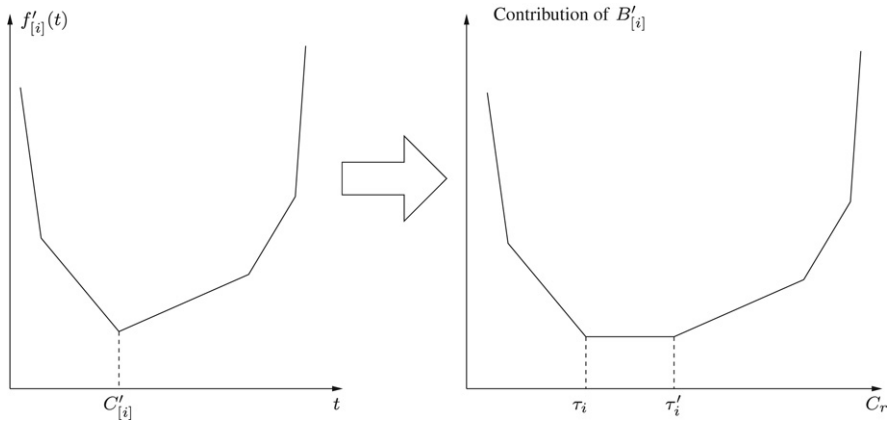

Fig. 5. Cost function of a block $B'_{[i]}$ and its contribution to f_r.

completion time in this schedule (which corresponds to the minimum of $f'_{[i]}$). In Fig. 4(a), $B_{[1]}$ and $B_{[2]}$ have merged into $B'_{[1]}$, $B_{[4]}$ and $B_{[5]}$ into $B'_{[3]}$.

We next have to add the two constraints which force the jobs to be executed after S_r and before C_r: setting S_r forces the leftmost blocks to be right-shifted and setting C_r forces the rightmost blocks to be left-shifted (see Fig. 4(b)). We then say a block is *critical* when it is constrained by the time window $[S_r, C_r]$: a block $B'_{[i]}$ is said to be *right-critical* (resp. *left-critical*), when there is no idle time between $B'_{[i]}$ and C_r (resp. when there is no idle time between S_r and $B'_{[i]}$). If a block $B'_{[i]}$ is not critical, it is said to be *on time* and its completion time remains $C'_{[i]}$. On Fig. 4(b), $B'_{[1]}$ and $B'_{[2]}$ are left-critical, $B'_{[3]}$ is on time and $B'_{[4]}$ is right-critical.

We are now able to calculate the cost of $B'_{[i]}$ in function of the completion time C_r of B_r. Formally, let $\tau_i = C'_{[i]} + \sum_{j>i}^{l} P'_{[j]}$ and $\tau'_i = C'_{[i]} + p_r + \sum_{j>i}^{l} P'_{[j]}$. Block $B'_{[i]}$ is *on time* when $\tau_i \le C_r \le \tau'_i$ and its cost is $f'_{[i]}(C'_{[i]})$. $B'_{[i]}$ is *right-critical* when $P_r \le C_r < \tau_i$ and its cost is given by $f'_{[i]}(C_r - \sum_{j>i}^{l} P'_{[j]})$. $B'_{[i]}$ is *left-critical* when $C_r > \tau'_i$ and its cost is given by $f'_{[i]}(C_r - p_r - \sum_{j>i}^{l} P'_{[j]})$. Fig. 5 shows how the cost of $B'_{[i]}$ in function of C_r is derived from the cost function $f'_{[i]}(t)$. Roughly speaking, it simply consists of inserting a horizontal segment of length p_r at the minimum of $f'_{[i]}$ and translating the function horizontally. Since $C'_{[i]}$ is the time at which $f'_{[i]}$ is minimum, the contribution of block $B'_{[i]}$ to the computation of f_r is convex and piecewise linear.

Eventually, we obtain f_r by adding the contributions of all the blocks plus the earliness and tardiness costs of J_r. This sum is therefore piecewise linear and convex. We have thus proved the induction hypothesis.

The cost functions $f_{[1]}, \ldots, f_{[k]}$ are piecewise linear and convex and have respectively $\|f_{[1]}\|, \ldots, \|f_{[k]}\|$ segments. In Hendel and Sourd [10], it is proved that the non constrained schedule can be obtained in $O(\sum_{j=1}^{k} \|f_{[j]}\| \log n)$. Then the contribution of each block $B'_{[i]}$ has to be computed: this contribution is derived from $f'_{[i]}$ in $O(\|f'_{[i]}\|)$ and has at most $\|f'_{[i]}\| + 1$ segments. Finally, $f_r(t)$ is obtained by adding these contributions and by taking into account the earliness and tardiness costs of job J_r. On the whole, $f_r(t)$ is computed in $O(\sum_{j=1}^{k} \|f_{[j]}\| \log n)$ time and has at most $\sum_{j=1}^{k} \|f_{[j]}\| + k + 2$ segments.

We now want to estimate the complexity to compute $f_0(t)$. First, all the inner nodes of the tree have to be treated.

We now show by induction that the cost function stored at each node has at most $3n' - 1$ segments where n' represents the total number of descendant nodes: each cost function stored at a leaf has $(3 \times 1 - 1)$ segments. Suppose that a node

Table 1
Comparaison Timing (top)—CPLEX (middle)—Ratio (bottom)

n	$\rho = 0.0$	$\rho = 0.2$	$\rho = 0.4$	$\rho = 0.6$	$\rho = 0.8$	$\rho = 1.0$
200	9.3 ms	6.4 ms	9.4 ms	6.2 ms	6.3 ms	7.9 ms
	900.1 ms	912.3 ms	903.1 ms	915.7 ms	871.8 ms	890.7 ms
	96.78	*142.55*	*96.07*	*147.69*	*138.38*	*112.75*
300	11.0 ms	28.2 ms	23.5 ms	29.9 ms	29.6 ms	20.2 ms
	2562.2 ms	2632.9 ms	2587.5 ms	2531.3 ms	2448.6 ms	2489.2 ms
	232.93	*93.37*	*110.11*	*84.66*	*82.72*	*123.23*
400	7.8 ms	34.3 ms	48.5 ms	45.3 ms	32.9 ms	40.9 ms
	4970.1 ms	4976.6 ms	5059.4 ms	4973.3 ms	4903.1 ms	4915.6 ms
	637.19	*145.09*	*104.32*	*109.79*	*149.03*	*120.19*
500	12.6 ms	56.4 ms	56.3 ms	53.5 ms	54.7 ms	54.6 ms
	9148.3 ms	9298.5 ms	9020.4 ms	8835.9 ms	8824.9 ms	9008.1 ms
	726.06	*164.87*	*160.22*	*164.85*	*161.33*	*164.98*
600	17.2 ms	106.3 ms	93.7 ms	93.5 ms	107.7 ms	110.8 ms
	14914.1 ms	15257.8 ms	14818.7 ms	14082.7 ms	13632.7 ms	14096.7 ms
	867.10	*143.54*	*158.15*	*150.62*	*126.58*	*127.23*

which has k sub-trees representing $n'_1, \ldots, n'_k$ nodes such that $\sum_{i=1}^{k} n'_i = n' - 1$ and each sub-tree has less than $3n'_i - 1$ segments. According to the previous section, it is clear that the number of segments of f_r is $\sum_{i=1}^{k} 3(n'_i - 1) + 2 + k$ which is lower than $3n' - 1$, which means that the computation of f_r is done in $O(n' \log n')$ time. Therefore, the complexity for computing all the nodes, and especially f_0 is in $O(n^2 \log n)$ time.

3.3. Experimental results

In this section, we compare the efficiency of the linear program (LP) and the timing algorithm presented in Section 3.2.

3.3.1. Instances

In order to solve (LP), we use CPLEX 9.1 on a 3.6 GHz PC with 3.5 Go RAM. We have randomly generated instances with 200, 300, 400, 500 and 600 jobs. The processing times are generated from the uniform distribution [1, 10]. The due dates are generated from the uniform distribution $[\max(0, P(1 - 3\rho/2)), P(1 + \rho/2)]$, where $P = \sum_{j \in \mathcal{J}} p_j$ and ρ is a parameter. When ρ is large, the due dates are scattered between 0 and $P(1 + \rho/2)$. When $\rho = 0.0$, all jobs have a common due date. ρ has the values 0.0, 0.2, 0.4, 0.6, 0.8 and 1.0. Earliness and tardiness penalties are generated from the uniform distribution [1, 5]. For each value of ρ and n, we have generated 10 instances.

The execution of the timing algorithm requires a fixed sequence. These sequences are generated randomly by inserting one job at a time at a randomly determined position in the forest associated to the sequence.

3.3.2. Results

The timing algorithm is implemented in Java. The timing algorithm and CPLEX are executed with the same instances and the same fixed sequences. Results are reported in Table 1. In each cell of the table, the first value is the execution time of the timing algorithm. The second value is the execution time of CPLEX. The execution times are in milliseconds. The third value indicates the ratio between the second and first values that indicates the speed improvement. As illustrated by Fig. 6 for instances with $\rho = 0.2$, we observe that the timing algorithm generally runs more than 100 times faster than CPLEX.

An interesting observation is the fact that the execution time of CPLEX for a given number of jobs is quite constant. But executing the timing algorithm on instances with $\rho = 0.0$, that means instances with a common due date, the timing algorithm runs faster than when $\rho > 0.0$. This can be explained by the fact that the optimal schedule contains no idle time.

4. Neighborhood search

We now consider the general problem and the aim of this section is to find good sequences for $S_1, C_1, \ldots, S_n, C_n$ that lead to near optimal solutions. The solution space corresponds to the set of valid sequences defined in Section 3.1. We first observe that if the set of feasible solutions is defined as in Section 2, its size would be $\frac{(2n)!}{2^n}$. With the dominance rules of Properties 1 and 2, the size is significantly reduced: since there are $C_n = \frac{\binom{2n}{n}}{n+1}$ well parenthesized words (C_n is the number of Catalan) and $n!$ ways to assign the n jobs to the n pairs of parentheses, the size of the solution set is "only" $n!C_n = \frac{(2n)!}{(n+1)!}$. However, this set is still too large to be completely enumerated, and thus this section is devoted to a local search.

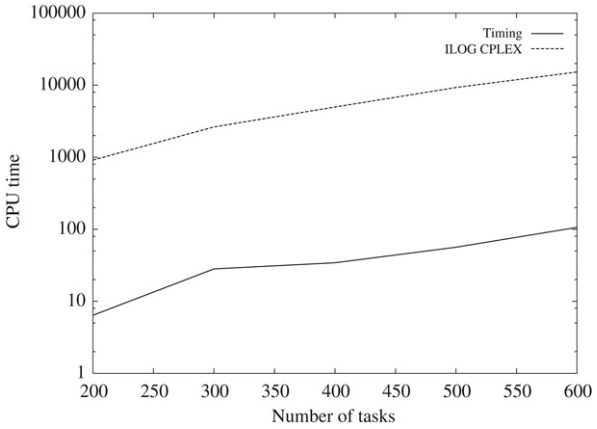

Fig. 6. Comparison of the Timing algorithm with CPLEX ($\rho = 0.2$).

4.1. Neighborhood definition

Local Search methods are known to provide good results for just-in-time scheduling problems [18]. In this section, we propose a neighborhood search for the preemptive problem. The considered neighborhood is based on the data structure introduced in Section 3.2. An iterated improvement (or *descent*) method is then implemented and the experimental results are finally reported.

We only define this neighborhood on valid sequences. For a given solution, we execute two operations to construct the neighborhood. The first operation is a SWAP operation. This operation consists of exchanging two jobs J_i and J_j in σ, that is S_i is swapped with S_j and C_i is swapped with C_j. In the forest associated to σ, it means that node i is swapped with node j. The second operation is an INSERT operation. The basic idea is to delete, for each job J_i, the corresponding events S_i and C_i from σ and to insert them again at new positions. These new positions are well defined in the tree representation: 4 positions are considered for each job J_i:

- around job J_j : $\ldots S_i\, S_j \ldots C_j\, C_i \ldots$
- within job J_j : $\ldots S_j\, S_i \ldots C_i\, C_j \ldots$
- before job J_j : $\ldots S_i\, C_i\, S_j \ldots C_j \ldots$
- after job J_j : $\ldots S_j \ldots C_j\, S_i\, C_i \ldots .$

The described modifications can easily be interpreted in the tree representation. Node i which corresponds to job J_i is deleted from the tree. Then, for each job J_j, node i is reinserted as follows:

- node i is inserted between j and its father
- node i is inserted as son of node j and the sons if j become the son of node i
- node i is inserted as left brother of node j
- node i is inserted as right brother of node j.

The definition of the neighborhood ensures that the parenthesis structure is preserved. For each sequence of the neighborhood, we apply the timing algorithm presented in Section 3.2. The size of the neighborhood is at most $5n^2$. If we apply the timing algorithm of Section 3.2 to each sequence in the neighborhood, the overall complexity is in $O(n^4 \log(n))$. It should be noted that some methods may be developed to avoid computing from scratch the timing of each sequence in the neighborhood. We refer to Hendel and Sourd [10] for efficient neighborhood search for the one-machine earliness–tardiness scheduling problem without preemption.

4.2. Experimental results

4.2.1. Algorithms

The goal of this section is to test the efficiency of our neighborhood, and to detect the classes of instances that are harder to solve. Therefore, we limit our study to an iterated improvement procedure. Clearly, more elaborated meta-heuristics would improve the quality of the solutions at the price of longer computation times but experiments in Section 4.2.3 show that our simple procedure gives satisfactory results.

Our algorithm has two steps. First, an initial non-preemptive solution is obtained by executing an iterated descent for the non-preemptive problem, based on the neighborhood defined in Hendel and Sourd [9]. This choice is motivated by the fact that this heuristic is fast, and we observed that there are in general few preemptions in optimal solutions. Then, a descent based on the neighborhood defined in Section 4.1 is launched.

Table 2
Quality of solution and execution times after five descents

n	30	60	90	Average
Best fit	8.19% 6.1 s	5.46% 94.6 s	4.81% 554.6 s	6.15%
First fit	8.46% 3.3 s	5.49% 38.6 s	4.69% 181.5 s	6.22%

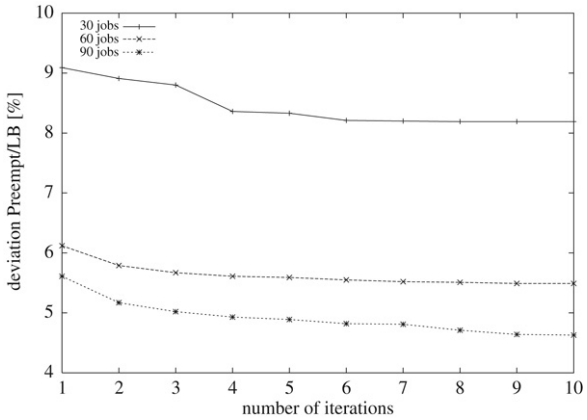

Fig. 7. Improvement of the quality of the solution with the number of iterations.

In our experimental results, we compare the "best-fit" and the "first-fit" implementations of the neighborhood search. In the best-fit method, the best sequence is chosen once the entire neighborhood is explored. This best sequence will be used for the next iteration. In the first-fit method, a new iteration is started as soon as an improving solution is found.

To improve the execution time for the "first fit" implementation, the order in which neighbors are considered is important. The INSERT neighborhood is explored first. We define a heuristic distance between each pair of jobs J_i and J_j and, in the nondecreasing order of these distances, we try to reinsert J_i into the four positions around J_j (and of course, in the same time, to reinsert J_j around J_i). The SWAP neighborhood is then similarly explored.

4.2.2. Instances

The generation scheme of these instances is based on those of the literature [11,16]. There are no release dates specified. We tested instances with $n = 30, 60, 90$ jobs. The processing times are generated from the uniform distribution $[10, 100)$ and the earliness–tardiness penalties are drawn from the uniform distribution $[1, 5]$. The due date of each job is drawn from the uniform distribution $[d_{\min}, d_{\min} + \rho P]$ where $d_{\min} = \max(0, P(\tau - \rho/2))$ and $P = \sum_{j=1}^{n} p_j$. The two parameters τ and ρ are, respectively, the tardiness and range parameters. These instances are available online.[1]

4.2.3. Results

The neighborhood search was implemented in Java, and the code was run on a 3.6 GHz PC with 3.5Gb RAM. The main conclusion is that execution times are satisfactory, since instances with up to 90 jobs can be solved in a reasonable time. Table 2 presents the average execution times for five executions in seconds ordered by the size of the instances. This execution time includes the computation of the initial sequence. The best-fit and first-fit algorithms are compared. We observe the first-fit version is at least 50% faster than the best-fit version.

To evaluate the quality of a solution, we compare its value to the lower bound presented in Section 5.2. We run the descent five times for each instance. The deviations are presented in Table 2. For the best fit version, we observe an average deviation of 6.15%. The first fit version provides an average deviation of 6.22%. We observe that the two versions provide results of equivalent quality, while the first fit method is significantly faster. Since the first step of the local search starts with a random sequence, multiple executions of the neighborhood search can improve the upper bound (and therefore the deviation). We executed the first fit version ten times for each instance and observed the deviation between the best known preemptive solution and the lower bound. For example, this deviation improves from 5.61% to 4.63% for instances with 90 jobs. Fig. 7 shows the improvements for jobs with $n = 30, 60, 90$ jobs and with up to ten iterations.

To determine the instances that are difficult to solve, that means the deviation between the lower bound and the result of the local search algorithm is high, the instances are sorted according to τ and ρ. Table 3 shows results of instances with

[1] http://www-poleia.lip6.fr/~sourd/project/et/sks30-90.zip.

Table 3
Variation of quality for 60 job instances for five executions executed with the first fit version

τ	$\tau = 0.2$	$\tau = 0.5$	$\tau = 0.8$
Dev Heur/LB	4.52%	6.28%	5.69%
Dev NP/Heur	0.6%	1.15%	1.12%
Dev NP/LB	5.45%	8.05%	7.34%
CPU Time	5.14 s	7.9 s	10.14 s
ρ	$\rho = 0.2$	$\rho = 0.5$	$\rho = 0.8$
Dev Heur/LB	3.09%	4.72%	8.67%
Dev NP/Heur	0.17%	0.59%	2.11%
Dev NP/LB	3.37%	5.58%	11.89%
CPU Time	3.09 s	4.72 s	8.67 s

$n = 60$ jobs obtained with the first fit version. Each instance is executed five times. For each value of τ and ρ four values are displayed:

(1) the deviation between the lower bound and the result of the local search heuristic **(Dev Heur/LB)**,
(2) the deviation between the non-preemptive solution and the local search heuristic which expresses a possible improvement of the solution by allowing preemption **(Dev NP/Heur)**,
(3) the deviation between the non-preemptive solution and the lower bound **(Dev NP/LB)**,
(4) the average computation time of the local search algorithm **(CPU Time)**.

The value of τ does not influence the quality of the solution, but the value of ρ does. The deviation between the non-preemptive solution and the lower bound is tighter for smaller ρ, and we observe a less important improvement between the non-preemptive upper bound and the preemptive solution. On the other hand, when ρ is large, the difference between the upper bound and the lower bound is more important and preemption is able to significantly improve the solution. On average, preemption improves non-preemptive solution by 2%. When comparing the computation times, we remark that instances with larger τ and ρ lead to longer computation times.

5. Branch and bound algorithm

In this section, we present a branch and bound algorithm, which allows one to compute exact solutions for simple instances. First we present the branching scheme and briefly discuss the upper bound and the dominance rules. In Section 5.2, we give special attention to the lower bound. In the last subsection we report experimental results.

5.1. Branching scheme

Let σ be a partial sequence where k jobs have already been inserted, that is σ is a valid sequence for these k jobs. At each node, an unscheduled job is selected, and its start and completion times are inserted at every possible positions into the existing sequence σ. Only those insertions are considered which satisfy the Property 1 of nested sequences. Heuristically, the unscheduled job – say $J_{k^\star}$ – with a minimal due date is selected and we first insert $C_{k^\star}$ at the rightmost position in σ and then we insert $S_{k^\star}$ at the rightmost position such that the new sequence is valid. Clearly, $O(k^2)$ branches are generated (see Fig. 8).

The efficiency of the branch-and-bound algorithm mainly relies on the lower bound presented in the next section. The upper bound used with the lower bound to cut nodes is the one provided by the solution of the neighborhood search presented in Section 4. We only compute this upper bound at the root node but it is update when a better feasible schedule is found during the search.

5.2. A lower bound

In this section, we propose a lower bound for JIT-POMP which is an adaptation of the ones proposed by Sourd and Kedad-Sidhoum [16] and independently by Bülbül et al. [4] for the non-preemptive case. The main idea of the lower bound is to compute a minimum cost assignment: each job J_i is divided into p_i operations $o_{i1}, o_{i2}, \ldots, o_{ip_i}$ which have unit execution times. These operations are to be assigned to T distinct time slots $[t-1, t)$ where $1 \leq t \leq T$ and T is the horizon of the schedule. The assignment costs are assumed to be operation-independent: we introduce for each job and each time slot an assignment cost c_{it}, whose value will be defined later. The problem can then be expressed as a transportation problem:

$$(\text{TP}) : \min \sum_{i=1}^{n} \sum_{t=1}^{T} c_{it} x_{it}$$

$$\text{s.t.} \quad \sum_{t=1}^{T} x_{it} = p_i, \quad \forall i = 1, \ldots, n$$

$$
\begin{array}{l}
S_1C_1S_2S_3C_3S_4C_4C_2\mathbf{S_5C_5}\\
S_1C_1\mathbf{S_5}S_2S_3C_3S_4C_4C_2\mathbf{C_5}\\
\mathbf{S_5}S_1C_1S_2S_3C_3S_4C_4C_2\mathbf{C_5}\\
S_1C_1S_2S_3C_3S_4C_4\mathbf{S_5C_5}C_2\\
S_1C_1S_2S_3C_3\mathbf{S_5}S_4C_4\mathbf{C_5}C_2\\
S_1C_1S_2\mathbf{S_5}S_3C_3S_4C_4\mathbf{C_5}C_2\\
S_1C_1S_2S_3C_3S_4\mathbf{S_5C_5}C_4C_2\\
S_1C_1S_2S_3C_3\mathbf{S_5C_5}S_4C_4C_2\\
S_1C_1S_2\mathbf{S_5}S_3C_3\mathbf{C_5}S_4C_4C_2\\
S_1C_1S_2S_3\mathbf{S_5C_5}C_3S_4C_4C_2\\
S_1C_1S_2\mathbf{S_5C_5}S_3C_3S_4C_4C_2\\
S_1C_1\mathbf{S_5C_5}S_2S_3C_3S_4C_4C_2\\
\mathbf{S_5}S_1C_1\mathbf{C_5}S_2S_3C_3S_4C_4C_2\\
S_1\mathbf{S_5C_5}C_1S_2S_3C_3S_4C_4C_2\\
\mathbf{S_5C_5}S_1C_1S_2S_3C_3S_4C_4C_2
\end{array}
$$

Fig. 8. Descendant partial sequences of $\sigma = S_1C_1S_2S_3C_3S_4C_4C_2$.

$$\sum_{j=1}^{n} x_{it} \leq 1, \quad \forall t = 1, \ldots, T$$
$$x_{it} \geq 0, \quad \forall i = 1, \ldots, n \text{ and } \forall t = 1, \ldots, T.$$

Since the time complexity depends on the scheduling horizon T, it has to be finite and as small as possible. A valid value for T is the makespan of the optimal schedule of $1|r_j, pmtn|C_{\max}$ where $r_j = d_j^s$. This problem is solved in polynomial time using Jackson's rule. We can easily prove that an optimal schedule for JIT-POMP completes before T.

To produce a valid lower bound, the assignment costs c_{it} of the above described transportation problem have to verify some additional sufficient conditions.

Property 3. *A sufficient condition for* (TP) *to provide a lower bound for* JIT-POMP *is to satisfy the following conditions:*

(1) *For each job J_i, the assignment costs of the operations are non-increasing between the interval $[1, d_i^c]$ and non-decreasing between the interval $[d_i^c + 1, T]$.*
(2) *The costs c_{it} are nonnegative.*
(3) *For each job J_i and for each time point t, we have*

$$\sum_{t<t'\leq t+p_i} c_{it'} \leq \alpha_i \max(0, (d_i^s - t)) + \beta_i \max(0, ((t + p_i) - d_i^c)).$$

Proof. For any feasible preemptive schedule, we can define a feasible (non-optimal) solution to (TP) by setting $x_{it} = 1$ if and only if J_i is in process in the time slot $[t-1, t)$. Proving that $\sum_{it} c_{it}x_{it}$ is not greater than the cost of the corresponding schedule is sufficient to prove the validity of the lower bound. We prove a slightly stronger result that is, for any job J_i, $\sum_{t=1}^{d_i^c} c_{it}x_{it} \leq \alpha_i E_i$ and $\sum_{t=d_i^c+1}^{T} c_{it}x_{it} \leq \beta_i T_i$.

To prove the first inequality, we first observe that condition (3) with $t = d_i^s$ gives $\sum_{t'=d_i^s+1}^{d_i^c} c_{it'} \leq 0$ and since condition (2) prohibits negative values, the strict equality $\sum_{t'=d_i^s+1}^{d_i^c} c_{it'} = 0$ holds. From condition (2), we also have that $c_{it} = 0$ for $t = d_i^s + 1, \ldots, d_i^c$. Therefore, we only have to prove that $\sum_{t=1}^{d_i^s} c_{it}x_{it} \leq \alpha_i E_i$. We define $n_i = \sum_{t=1}^{d_i^s} x_{it}$ be the number of operations assigned before d_i^s in (TP). If $n_i = 0$, then the inequality is obviously satisfied. Otherwise, we have $S_i < d_i^s$. We have $\sum_{t=1}^{d_i^s} c_{it}x_{it} = \sum_{t=S_i+1}^{d_i^s} c_{it}x_{it} \leq \sum_{t=S_i+1}^{S_i+n_i} c_{it} \leq \sum_{t=S_i+1}^{S_i+p_i} c_{it}$. The first inequality comes from condition (1) and the second inequality comes from condition (2). Finally, deriving condition (3) with $t = S_i$ gives that $\sum_{t=S_i+1}^{S_i+p_i} c_{it} \leq \alpha_i E_i$, which proves that $\sum_{t=1}^{d_i^c} c_{it}x_{it} \leq \alpha_i E_i$. The proof of the other inequality is symmetrical. □

For the non-preemptive problem, condition (3) alone is sufficient (and necessary) for the validity of the (TP) lower bound. The following example shows that conditions (1) and (2) are indeed necessary in the preemptive case. Let us consider two jobs J_1 and J_2 such that ($p_1 = 2, d_1 = 5, \alpha_1 = 2, \beta_1 = 100$) and ($p_2 = 3, d_2 = 4, \alpha_2 = 100, \beta_2 = 100$). The assignment costs of job J_1 satisfy condition (3) but violate (1) and (2). They are defined as follows:

$c_{11} = 6$	$c_{12} = 0$	$c_{13} = 4$	$c_{14} = -2$	$c_{15} = 2$	$c_{1i} = 98, i > 5$
$c_{21} = 100$	$c_{22} = 0$	$c_{23} = 0$	$c_{24} = 0$	$c_{25} = 100$	$c_{1i} = 100, i > 5$

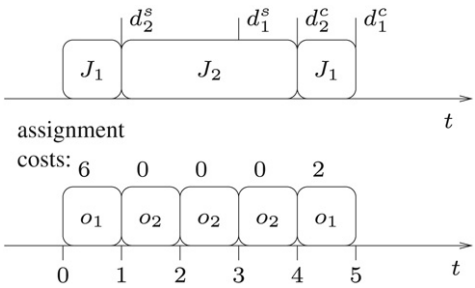

Fig. 9. The value of the schedule is 6, but the minimal matching has a value of 8.

As illustrated by Fig. 9, the value of the minimal matching is 8, whereas the value of the optimal schedule is 6, which clearly proves that such assignment costs do not give a valid lower bound. The assignment costs proposed by Sourd and Kedad-Sidhoum [16] and Bülbül et al. [4] both satisfy conditions (1) and (3) but the cost proposed in the latter paper may be negative. Therefore, we will consider the costs of Sourd and Kedad-Sidhoum which are given by

$$c'_{jt} = \begin{cases} \alpha_j \left\lfloor \dfrac{d_j^c - t}{p_j} \right\rfloor & \text{if } t \le d_j^c, \\ \beta_j \left\lceil \dfrac{t - d_j^c}{p_j} \right\rceil & \text{if } t > d_j^c. \end{cases} \tag{6}$$

This lower bound provides good results for the non-preemptive case. Since the optimum of JIT-POMP is smaller than the optimum of the non-preemptive case, the gap between the lower bound and the optimum is even smaller.

To compute the lower bound, Sourd and Kedad-Sidhoum [16] proposed an $O(n^2T)$ algorithm which is based on the Hungarian algorithm.

This lower bound can be incorporated into the branch and bound algorithm to compute a lower bound for a partially fixed sequence. The cost of the partial sequence can be computed with the timing algorithm presented in Section 3. This is, of course, a lower bound for any extended sequence derived from this partial sequence. But we can reinforce it by adding the value of (TP) for all the unscheduled jobs. As jobs are inserted in a fixed order (the non-decreasing order of their due dates), we know that nodes that are at the same height in the branching tree always contain the same subset of jobs. Therefore, only n instances of (TP) are useful, namely the instances (TP_i) ($1 \le i \le n$) that are the restrictions of (TP) for the $n - i$ jobs with largest due dates. Thus we can pre-solve them before the start of the branch-and-bound search.

This lower bound independently considers scheduled and non-scheduled jobs, which is a very strong relaxation. We have tried to adapt the assignment costs to take into account the partial sequence. In such an approach, the transportation problem has to be solved at each node, which is very time consuming. Unfortunately, our experimental tests showed that the improvement of the lower bound was not good enough with respect to the computation times.

5.3. Experimental results

This branch and bound algorithm has been implemented in Java. Instances with $n = 10, 12, 14, 16, 18, 20$ jobs were generated. The processing times are drawn from the uniform distribution [1, 20] and the earliness and tardiness penalties are in [1, 5]. The due dates are chosen from the uniform distribution $[P - 3\rho P/2, P + \rho P/2]$ where $P = \sum_{i=1}^{n} p_i$ and $\rho \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$. Five instances were generated for each value of n and ρ.

Table 4 shows the details of the execution. For each pair of ρ and n, four values are displayed:

(1) the deviation between the upper bound and the optimal preemptive solution **(Dev UpperNP/OptP)**,
(2) the deviation between the optimal non-preemptive solution and the optimal preemptive solution **(Dev OptNP/OptP)**,
(3) the deviation between the optimal preemptive solution and the lower bound **(Dev OptP/LB)**,
(4) the average computation time of the branching procedure in seconds (computation of the upper bound is not taken into account) **(CPU Time)**.

The possibility of solving instances in a reasonable time clearly depends on the difficulty of the instance. When the due dates are tighter, the algorithm takes more time. On the one hand, when $\rho = 1.0$, instances with up to 20 jobs can be solved. On the other hand, when $\rho = 0.2$, then even small instances with $n = 12$ jobs need more than 100 min to get solved. We remark that the variance of the CPU times are large even for the same class of instances. For example, the average CPU time for $\rho = 1.0$ and $n = 20$ is 6246.58 s. But four of the five instances have been solved in less than 130 s.

93% of the tested instances admit an optimal preemptive solution that is strictly better than the non-preemptive optimum. The overall improvement between the optimal non preemptive solution and the optimal preemptive solution is 9.21%. For 68% of the tested instances the upper bound provides the optimal solution. The overall deviation between the preemptive upper bound and the optimal preemptive solution is 1.48%. It shows that our simple iterative improvement procedure is very efficient.

Table 4
Results for branch and bound algorithm

Number of jobs		10	12	14	16	18	20
$\rho = 0.2$	**Dev UpperP/OptP**	1.65%	1.84%				
	Dev OptNP/OptP	2.26%	2.6%				
	Dev OptP/LB	5.95%	6.67%				
	CPU Time	279.59 s	6151.85 s				
$\rho = 0.4$	**Dev UpperP/OptP**	3.68%	1.6%	1.33%			
	Dev OptNP/OptP	7.29%	4.25%	12.12%			
	Dev OptP/LB	7.02%	7.03%	6.82%			
	CPU Time	2.74 s	121.08 s	139.73 s			
$\rho = 0.6$	**Dev UpperP/OptP**	0%	0%	1.32%	0.77%	0%	0%
	Dev OptNP/OptP	12.02%	10.61%	13.07%	8.63%	6.97%	7.67%
	Dev OptP/LB	10.93%	9.41%	11.63%	12.02%	6.68%	7.74%
	CPU Time	2.61 s	0.6 s	2.84 s	4.89 s	271.18 s	56.61 s
$\rho = 0.8$	**Dev UpperP/OptP**	1.25%	0%	2.34%	0.74%	1.47%	4.11%
	Dev OptNP/OptP	10.68%	14.59%	6.13%	8.03%	8.53%	8.78%
	Dev OptP/LB	6.82%	5.87%	1.62%	8.8%	9.98%	9.34%
	CPU Time	3.75 s	6.61 s	3.09 s	6.08 s	66.12 s	29.34 s
$\rho = 1.0$	**Dev UpperP/OptP**	0%	4.19%	3.39%	0%	4.02%	0.59%
	Dev OptNP/OptP	14.23%	11.63%	10.83%	11.71%	9.31%	7.44%
	Dev OptP/LB	7.4%	10.72%	12.85%	8.19%	8.46%	6.63%
	CPU Time	0.03 s	0.35 s	1.54 s	2.06 s	269.02 s	6246.58 s

6. Conclusion

We have introduced a new earliness–tardiness problem, in order to deal with preemption. We have proposed a polynomial algorithm to solve the essential timing sub-problem. Then, we have shown the interest of this algorithm in solving the general problem: indeed, our descent algorithm finds near-optimal feasible schedules.

An interesting perspective would consist in finding a more compact encoding of the feasible schedule, in order to obtain both a faster timing problem and a smaller solution set. Such an approach would require new dominance properties on the sequences. In view of practical applications, we should consider a more precise model for preemption in order to penalize the number of job interruptions and to introduce additional "launch" times when a job is restarted.

Acknowledgment

The authors would like to thank the anonymous referees for their helpful comments.

References

[1] A.V. Aho, J.D. Ullman, Foundation of Computer Science, W.H. Freeman, 1994.
[2] R.K. Ahuja, D.S. Hochbaum, J.B. Orlin, Solving the convex cost integer dual network flow problem, Management Science 49 (2003) 950–964.
[3] J. Bang-Jensen, G. Gutin, Digraphs: Theory, Algorithms and Applications, Springer-Verlag, 2000.
[4] K. Bülbül, P. Kaminsky, C. Yano, Preemption in single machine earliness/tardiness scheduling, Journal of Scheduling 10 (2007) 271–292.
[5] Ph. Chrétienne, F. Sourd, Scheduling with convex cost functions, Theoretical Computer Science 292 (2003) 145–164.
[6] J.S. Davis, J.J. Kanet, Single-machine scheduling with early and tardy completion costs, Naval Research Logistics 40 (1993) 85–101.
[7] B. Esteve, C. Aubijoux, A. Chartier, V. T'kindt, A recovering beam search algorithm for the single machine Just-in-time scheduling problem, European Journal of Operational Research 172 (2006) 798–813.
[8] M.R. Garey, R.E. Tarjan, G.T. Wilfong, One-processor scheduling with symmetric earliness and tardiness penalties, Mathematics of Operations Research 13 (1988) 330–348.
[9] Y. Hendel, F. Sourd, Efficient neighborhood search for the one-machine earliness–tardiness scheduling problem, European Journal of Operational Research 173 (2006) 108–119.
[10] Y. Hendel, F. Sourd, An improved earliness–tardiness timing algorithm, Computers & Operations research 34 (2007) 2931–2938.
[11] J.A. Hoogeveen, S.L. van de Velde, A branch-and-bound algorithm for single-machine earliness–tardiness scheduling with idle time, INFORMS Journal on Computing 8 (1996) 402–412.
[12] J.A. Hoogeveen, S.L. van de Velde, Scheduling with target start times, European Journal of Operational Research 129 (2001) 87–94.
[13] A.V. Karzanov, S.T. McCormick, Polynomial methods for separable convex optimization in unimodular linear spaces with applications, SIAM Journal on Computing 26 (1997) 1245–1275.
[14] J.K. Lenstra, A.H.G. Rinnooy Kan, P. Brucker, Complexity of machine scheduling problems, Annals of Discrete Mathematics 1 (1977) 343–362.
[15] Y. Pan, L. Shi, Dual Constrained single machine sequencing to minimize total weighted completion time, IEEE Transactions on Automation Science and Engineering 2 (2005) 344–357.
[16] F. Sourd, S. Kedad-Sidhoum, The one-machine scheduling with earliness and tardiness penalties, Journal of Scheduling 6 (2003) 533–549.
[17] F. Sourd, Optimal Timing of a sequence of tasks with general completion costs, European Journal of Operational Research 165 (2005) 82–96.
[18] F. Sourd, S. Kedad-Sidhoum, A faster branch-and-bound algorithm for the earliness–tardiness scheduling problem, Journal of Scheduling 11 (2008) 49–58.
[19] W. Szwarc, S.K. Mukhopadhyay, Optimal timing schedules in earliness–tardiness single machine sequencing, Naval Research Logistics 42 (1995) 1109–1114.

[20] É. Tardos, A strongly polynomial algorithm to solve combinatorial linear programs, Operations Research 34 (1986) 250–256.
[21] G. Wan, B.P.C. Yen, Tabu search for single machine with distinct due windows and weighted earliness/tardiness penalties, European Journal of Operational Research 142 (2002) 271–281.