Scheduling a Single Machine to Minimize a Regular Objective Function under Setup Constraints

Philippe Baptiste [a] Claude Le Pape [b]

[a] *CNRS LIX, Ecole Polytechnique, F-91128, Palaiseau, Philippe.Baptiste@Polytechnique.fr*

[b] *ILOG, Gentilly, France, clepape@ilog.fr*

Abstract

Motivated by industrial applications, we study the scheduling situation in which a set of jobs subjected to release dates and deadlines are to be performed on a single machine. The objective is to minimize a regular sum objective function $\sum_i f_i$ where $f_i(C_i)$ corresponds to the cost of the completion of job J_i at time C_i. On top of this, we also take into account setup times and setup costs between families of jobs as well as the fact that some jobs can be "unperformed" to reduce the load of the machine. We introduce lower bounds and dominance properties for this problem and we describe a Branch and Bound procedure with constraint propagation. Experimental results are reported.

Key words: Single Machine Scheduling; Sequencing; Setup Time; Setup Cost; Unperformed Jobs

1 Introduction

Real manufacturing scheduling problems exhibit a number of difficult features that are often ignored in the literature. Motivated by a new testbed inspired by industrial real life situations [29], we study in this paper the one machine scheduling problem with:

- Release dates and deadlines.
- Costs dependent on the completion times of activities.
- Possibilities of leaving some jobs "unperformed."
- Setup times and costs.

In this problem, a set of jobs $\{J_1, ..., J_n\}$ subjected to release dates r_i and deadlines d_i are to be performed on a single machine. The processing time of J_i is p_i. The objective is to minimize a regular (*i.e.*, non-decreasing) sum objective function $\sum_i f_i$ where $f_i(C_i)$ corresponds to the cost of completing J_i at time $C_i \in [r_i + p_i, d_i]$. Two extensions of this core problem are considered:

- When the machine is overloaded some jobs can be *"unperformed"* to reduce the machine load. In such a case, an "unperformance" cost u_i is associated to each job J_i. When the job is not scheduled in its time window $[r_i, d_i]$, the cost u_i is added to the objective function. Note that if the f_i functions are null, the problem reduces to minimizing $\sum w_i U_i$, a well-known objective function in scheduling theory (see for instance, [10]). This additional cost can be integrated in the cost functions f_i. Note however than when non-performance costs are used, f_i is constant over the interval (d_i, ∞) and the exact completion time of job J_i is no longer relevant once it is known to exceed d_i.
- Due to manufacturing constraints, *setups* must be performed between jobs with different machine feature requirements. We rely on the following model: there are q families of jobs and $\phi(J_i) \in \{1, ..., q\}$ denotes the family of the job J_i. Within the same family, there is no transition time nor cost. But between consecutive jobs of families ϕ_1 and ϕ_2, at least $\delta(\phi_1, \phi_2)$ units of time must elapse. Moreover, a cost $f(\phi_1, \phi_2)$ is associated to this setup.

We describe a new lower bound for this very general problem and a Branch and Bound procedure with constraint propagation. Experimental results are reported.

1.1 Literature Review

A lot of research has been carried on the unweighted total tardiness problem with equal release dates $\mathbf{1}||\sum \mathbf{T_i}$. Powerful dominance rules have been introduced by Emmons [23]. Lawler [27] has proposed a dynamic programming algorithm that solves the problem in pseudo-polynomial time. Finally, Du and Leung have shown that the problem is NP-Hard [22]. Most of the exact methods for solving $1||\sum T_i$ strongly rely on Emmons' dominance rules. Potts and Van Wassenhove [30], Chang et al.[14] and Szwarc et al.[40], have developed Branch and Bound methods using the Emmons rules coupled with the decomposition rule of Lawler [27] together with some other elimination rules. The best results have been obtained by Szwarc, Della Croce and Grosso [40, 41] with a Branch and Bound method that efficiently handles instances with up to 500 jobs. The weighted problem $\mathbf{1}||\sum \mathbf{w_i T_i}$ is strongly NP-Hard [27]. For this problem, Rinnooy Kan et al.[35] and Rachamadugu [31] have extended the Emmons Rules [23]. Exact approaches based on Dynamic Programing and

Branch and Bound have been tested and compared by Abdul-Razacq, Potts and Van Wassenhove [1].

There are less results on the problem with arbitrary release dates $\mathbf{1}|\mathbf{r_i}|\sum \mathbf{T_i}$. Chu and Portmann [18] have introduced a sufficient condition for local optimality which allows them to build a dominant subset of schedules. Chu [16] has also proposed a Branch and Bound method using efficient dominance rules. This method handles instances with up to 30 jobs for the hardest instances and with up to 230 jobs for the easiest ones. More recently, Baptiste, Carlier and Jouglet [4] have described a new lower bound and some dominance rules which are used in a Branch and Bound procedure which handles instances with up to 50 jobs for the hardest instances and 500 jobs for the easiest ones. Let us also mention that exact Branch and Bound procedures have been proposed for the same problem with setup times [32, 38]. For the $\mathbf{1}|\mathbf{r_i}|\sum \mathbf{w_i}\mathbf{T_i}$ problem, Akturk and Ozdemir [3] have proposed a sufficient condition for local optimality which improves heuristic algorithms. This rule is then used with a generalization of Chu's dominance rules to the weighted case in a Branch and Bound algorithm [2]. This Branch and Bound method handles instances with up to 20 jobs. Recently Jouglet et al. [25] have proposed a new Branch and Bound that solves all instances with up to 35 jobs.

For the total completion time problem, in the case of identical release dates, both the unweighted and the weighted problems $\mathbf{1}||\sum \mathbf{w_i}\mathbf{C_i}$ can easily be solved polynomially in $O(n \log n)$ by applying the Shortest Weighted Processing Time priority rule, also called Smith's rule [37]. For the unweighted problem with release dates, several researchers have introduced dominance properties and proposed a number of algorithms [13, 21, 20]. Chu [15, 17] has proved several dominance properties and has provided a Branch and Bound algorithm. Chand, Traub and Uzsoy used a decomposition approach to improve Branch and Bound algorithms [12]. Among the exact methods, the most efficient algorithms [15, 12] can handle instances with up to 100 jobs. The weighted case with release dates $\mathbf{1}|\mathbf{r_i}|\sum \mathbf{w_i}\mathbf{C_i}$ is NP-Hard in the strong sense [34] even when the preemption is allowed [26]. Several dominance rules and Branch and Bound algorithms have been proposed [8, 9, 24, 33]. To our knwoledge, the best results are obtained by Belouadah, Posner and Potts with a Branch and Bound algorithm which has been tested on instances involving up to 50 jobs.

Many exact methods have been proposed for the problem $\mathbf{1}|\mathbf{r_i}|\sum \mathbf{U_i}$ [5, 19, 7]. More recently, Ruslan Sadykov [36] has proposed a very efficient Branch and Cut algorithm for this problem.

Our Branch and Bound has been implemented in a Constraint Programming framework. Constraint Programming is a paradigm aimed at solving combinatorial optimization problems. Often these combinatorial optimization problems are solved by defining them as one or several instances of the *Constraint Satisfaction Problem* (CSP). Informally speaking, an instance of the CSP is described by a set of *variables*, a set of possible values for each variable, and a set of *constraints* between the variables. The set of possible values of a variable is called the variable's *domain*. A constraint between variables expresses which combinations of values for the variables are allowed. Constraints can be stated either implicitly (intentionally), *e.g.*, an arithmetic formula, or explicitly (extensionally), where each constraint is expressed as a set of tuples of values that satisfy the constraint. The question to be answered for an instance of the CSP is whether there exists an assignment of values to variables, such that all constraints are satisfied. Such an assignment is called a *solution* of the CSP.

One of the key ideas of constraint programming is that constraints can be used "actively" to reduce the computational effort needed to solve combinatorial problems. Constraints are thus not only used to test the validity of a solution, as in conventional programming languages, but also in an active mode to remove values from the domains, deduce new constraints, and detect inconsistencies. This process of actively using constraints to come to certain deductions is called *constraint propagation*. The specific deductions that result in the removal of values from the domains are called *domain reductions*. The set of values in the domain of a variable that are not invalidated by constraint propagation is called the *current domain* of that variable.

As the general CSP is NP-complete constraint propagation is usually incomplete. This means that some but not all the consequences of the set of constraints are deduced. In particular, constraint propagation cannot detect all inconsistencies. Consequently, one needs to perform some kind of search to determine if the CSP instance at hand has a solution or not. Most commonly, search is performed by means of a *tree search* algorithm.

We associate a start variable $\mathcal{S}_i$ to each job J_i. Its domain is initially set to $[r_i, d_i - p_i]$. Throughout the search, the domains of start variables change but to keep things simple, we denote by r_i the minimum value in the domain of $\mathcal{S}_i$ and by d_i the maximum value in $\mathcal{S}_i$ plus p_i. To model the objective function, we add a variable $\mathcal{F}$ to the model. It is constrained to be equal to the value of the objective function.

$$\mathcal{F} = \sum_i f_i(\mathcal{S}_i + p_i) \tag{1}$$

To propagate the above constraint, we rely on Arc-B-consistency [28], (*i.e.*, Arc-consistency restricted to the bounds of the domains of the variables). Given a constraint c over n variables $x_1, \ldots, x_n$ and a domain $d(x_i) = [lb(x_i), ub(x_i)]$ for each variable x_i, c is said to be "arc-B-consistent" if and only if for any variable x_i and each of the bound values $v_i = lb(x_i)$ and $v_i = ub(x_i)$, there exist values $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$ in $d(x_1), \ldots, d(x_{i-1}), d(x_{i+1}), \ldots, d(x_n)$ such that $c(v_1, \ldots, v_n)$ holds. Arc-B-consistency can be easily achieved on (1) since the f_i functions are non-decreasing.

To propagate the resource constraints, we use the disjunctive constraint and the edge-finding mechanism, as implemented in ILOG SCHEDULER [6]. It consists in determining whether an activity must, can, or cannot be the first or the last to execute among a set of activities that require the same machine [11]. This mechanism provides tightened time bounds for activities requiring the same machine. It is known to be extremely powerful and can be implemented in $O(n \log n)$.

Once all constraints of the problem are added, a common technique to look for an optimal solution is to solve successive decision variants of the problem. Several strategies can be considered to minimize the value of $\mathcal{F}$. One way is to iterate on the possible values, either from the lower bound of its domain up to the upper bound until one solution is found, or from the upper bound down to the lower bound determining each time whether there still is a solution. Another way is to use a dichotomizing algorithm, where one starts by computing an initial upper bound ub and an initial lower bound lb for $\mathcal{F}$. Then

(1) Set $D = \left\lfloor \dfrac{lb + ub}{2} \right\rfloor$
(2) Constrain $\mathcal{F}$ to be at most D. Then solve the resulting CSP, *i.e.*, determine a solution with $\mathcal{F} \leq D$ or prove that no such solution exists. If a solution is found, set ub to the value of $\mathcal{F}$ in the solution; otherwise, set lb to $D + 1$.
(3) Iterate steps 1 and 2 until $ub = lb$.

We rely on the *edge-finding* branching scheme (see for instance [11]). Rather than searching for the starting times of jobs, we look for a *sequence* of jobs. This sequence is built both from the beginning and from the end of the schedule. Throughout the search tree, the status of a job changes. It can be marked as "ranked" (*i.e.*, already scheduled at the begining or at the end of the sequence), or "unranked". Among unranked jobs, a job is a "possible first" (last) if it can be the first (last) one to execute among all unranked jobs. Conversely, some unranked jobs are marked as "non-possible first" (last). The status of the jobs is dynamically maintained throughout the search. The edge-finding rules detect immediately that some jobs can, cannot or must be the first to execute in a sequence. Moreover, they maintain some consistency between the

scheduling data (release date, deadline) and the status of the jobs. Thanks to the edge-finding rules the number of "possible first" jobs is usually low at each node of the search tree.

Our branching strategy is very simple : We select one job among the unranked jobs that can be first and we make it precede all other unranked jobs. Upon backtracking, this job is marked as "non-possible first" and another unranked job is chosen.

The heuristic used to select the job to schedule first is fairly difficult to setup since we have arbitrary cost functions. Following preliminary experiments, we have decided to use the $PRTT$ function of Chu and Portmann [18]. It has been introduced for the single machine total tardiness problem and it has been shown to be very efficient. If δ_i denotes the due date of J_i, $PRTT(i)$ is then defined as $\max(r_i+p_i, \delta_i)$. In our case, we do not have a due date but we define artificially one as the first time point t after r_i such that $f_i(r_i) < f_i(t+1)$. If f_i is constant its due date is then set to the deadline d_i. Since f_i functions are arbitrary, it is very difficult to build reasonably good heuristics for the problem. We believe that in concrete cases, our naive adaptation of $PRTT$ works rather well. The experimental evaluation of this heuristic is however not in the scope of the paper.

Finally, we use the "No Good Recording" technique, a simple and powerful technique to detect infeasible situations that have "almost" been encountered (see for instance [4]). Whenever it is known that the current partial sequence cannot be extended to a feasible schedule improving on the best-known solution, the characteristics of the partial sequence that make this extension impossible are recorded as a "No Good". If later in the search these characteristics are encountered again, the corresponding node is immediately discarded.

In our algorithm, we save:

- the *set* of jobs belonging to the partial sequence (recall that the schedule is built from left to right);
- the completion time of the last job in the partial sequence;
- and the cost associated to the partial sequence.

If later in the search tree there exists a "No Good" including the current partial sequence, with a smaller (or equal) completion time and a smaller (or equal) total cost, then we immediately backtrack. Indeed, the partial sequence at the current node cannot be extended to an improving complete schedule, since otherwise the "No Good" could have been extended to an improving complete schedule.

2 Lower Bound

In this section and in Section 3, we assume that jobs cannot be left unperformed and are not subjected to setup times and costs. We also assume that the cost functions are defined at any time point (*i.e.*, before $r_i + p_i$ and after the deadline). If this is not the case we can extend a function f_i as follows $\forall t \leq r_i + p_i, f_i(t) = f_i(r_i + p_i)$ and $\forall t \geq d_i, f_i(t) = f_i(d_i)$. The lower bound is computed in two steps:

step 1 First we compute a vector $(C_{[1]}, C_{[2]}, ..., C_{[n]})$ such that $\forall i, C_{[i]}$ is a lower bound of the ith smallest completion time in any schedule.

step 2 Second we define an assignment problem between jobs and the above completion times $C_{[1]}, C_{[2]}, ..., C_{[n]}$. The cost of assigning the job J_i to the date $C_{[u]}$ is $f_i(C_{[u]})$ and we seek to minimize the total assignment cost.

Note that if the functions f_i were not regular (non-decreasing), the optimal assignment cost would not be a lower bound because optimal schedules would not be left shifted.

To achieve the *first step*, we allow preemption and jobs are scheduled according to the SRPT (Shortest Remaining Processing Time) rule: Each time a job becomes available or is completed, a job with the shortest remaining processing time among the available and uncompleted jobs is scheduled. It is well known that the completion times $C_{[1]}, C_{[2]}, ..., C_{[n]}$ obtained on this preemptive schedule are minimal, *i.e.*, $\forall i, C_{[i]}$ is a lower bound of the completion time of the i-th job in any preemptive schedule. Using heap structures, the preemptive SRPT schedule can be build in $O(n \log n)$. This first step is a straight adaptation of Chu's lower bound for $1|r_i|\sum T_i$.

The *second step* is a simple assignment problem and the Hungarian algorithm could be used to compute an optimal solution in cubic time. As we wish to use this lower bound in a Branch and Bound procedure, we propose to compute a fast lower bound of the assignment problem. It is possibly not as good as the optimal assignment value but is computed much faster. In the following, we rely on the fact that the f_i functions are non-decreasing.

Our basic observation is that, since the f_i functions are non-decreasing and since $C_{[1]} \leq C_{[2]} \leq ... \leq C_{[n]}$, the total assignment cost is at least

$$\sum_{i=1}^{n} f_i(C_{[1]}).$$

More generally, at least $n-k+1$ jobs remain to be scheduled after or at $C_{[k]}$. Hence, $\forall k$, the total assignment cost is at least

$$\begin{aligned} \min \quad & \sum_{i \in V}^{n} f_i(C_{[k]}) \\ V \subseteq \{1, ..., n\} \quad & \\ |V| = n-k+1 & \end{aligned}$$

Our algorithm works as follows: Initially, the lower bound lb is set to 0. We then iterate from $k = 1$ up to n. At each iteration, we increase the lower bound lb to take into account the fact that $n-k+1$ jobs remain to be scheduled after or at time $C_{[k]}$ and we modify the f_i functions:

- The lower bound lb is increased of the $n-k+1$ smallest values in $\{f_1(C_{[k]}), ..., f_n(C_{[k]})\}$.
- The functions f_i are changed into $f_i(t) - f_i(C_{[k]})$.

Proof of the correctness of the algorithm. Let $A(f_1, ..., f_n, C_{[k]}, C_{[k+1]}, ..., C_{[n]})$ denote the cost of an optimal assignment of $n-k+1$ jobs among n to the completion times $C_{[k]}, ..., C_{[n]}$ using the cost functions $f_1, ..., f_n$. We prove by induction that after each iteration of the algorithm, $lb + A(f_1, ..., f_n, C_{[k]}, ..., C_{[n]})$ is a lower bound of the initial problem.

Consider an optimal assignment for $A(f_1, ..., f_n, C_{[k]}, ..., C_{[n]})$ and let $\sigma(1), ..., \sigma(n-k+1)$ denote the jobs assigned respectively to $C_{[k]}, ..., C_{[n]}$. $A(f_1, ..., f_n, C_{[k]}, ..., C_{[n]})$ is then equal to

$$\sum_{i=1}^{n-k+1} f_{\sigma(i)}(C_{[k+i-1]})$$

and it can be decomposed as

$$\sum_{i=1}^{n-k+1} f_{\sigma(i)}(C_{[k]}) + \sum_{i=1}^{n-k+1} \left(f_{\sigma(i)}(C_{[k+i-1]}) - f_{\sigma(i)}(C_{[k]}) \right).$$

The first sum is not smaller than the sum of the $n-k+1$ smallest values in $\{f_1(C_{[k]}), ..., f_n(C_{[k]})\}$. Moreover the second sum equals

$$\sum_{i=2}^{n-k+1} \left(f_{\sigma(i)}(C_{[k+i-1]}) - f_{\sigma(i)}(C_{[k]}) \right).$$

and thus it is an upper bound of an optimal assignment of $n-k$ jobs among n to the completion times $C_{[k+1]}, ..., C_{[n]}$ using the cost functions $f_1 - f_1(C_{[k]}), ..., f_n - f_n(C_{[k]})$. Hence,

$$lb + \sum_{i=1}^{n-k+1} f_{\sigma(i)}(C_{[k]}) + A(f_1 - f_1(C_{[k]}), ..., f_n - f_n(C_{[k]}), C_{[k+1]}, ..., C_{[n]})$$

is also a lower bound of the initial problem.

A major weakness of the above lower bound is that release dates are taken into account in the first step (computation of the earliest possible completion times) but not in the second one (assignment). Consider a 2-job instance with J_1 ($r_1 = 0, p_1 = 2, d_1 = 4, f_1(2) = f_1(3) = f_1(4) = 0$) and J_2 ($r_2 = 1, p_2 = 2, d_2 = 4, f_2(3) = 0, f_2(4) = 1$). The completion time vector is $(2, 4)$ and since we do not take into account release dates the optimal assignment is 0 while there is no feasible schedule with a cost lower than 1.

3 Dominance Properties

A dominance rule is a constraint that can be added to the initial problem without changing the value of the optimum, *i.e.*, there is at least one optimal solution of the problem for which the dominance holds. Dominance rules can be of prime interest since they can be used to reduce the search space. However they have to be used with care since the optimum can be missed if conflicting dominance rules are combined.

In [4] we have introduced a set of dominance rules for $1|r_i|\sum T_i$ that generalize and extend Emmons rules [23]. Unfortunately, in the context of arbitrary non-decreasing objective functions, we have very little information available and it is rather difficult to generalize such rules. We propose a very simple set of rules which allow us to add precedences between jobs. As we will see in Section 5, adding such precedences tightens the problem, increases the lower bound and hence improves our search procedure.

Definition 1 *J_i dominates J_j if and only if (1) $p_i \leq p_j$, (2) $r_i \leq r_j$, (3) $d_i \leq d_j$ and (4) $\forall t \in [r_j + p_j, d_i], (f_i - f_j)(t)$ is non-decreasing.*

Proposition 1 *If J_i dominates J_j and if $[r_j, d_i]$ does not contain the release date of another job then, there is an optimal schedule in which J_i precedes J_j.*

PROOF. Let us consider an optimal schedule in which J_j precedes J_i. We swap the two jobs and move backward all jobs inbetween them. More precisely J_i is completed at time $C_j - p_j + p_i$ and J_j is completed at C_i. All jobs in between are scheduled $p_j - p_i$ units of time earlier. Conditions (1), (2) and (3) ensure that the release dates and the deadlines of J_i and J_j are met. Moreover, jobs in between have a release date smaller than r_j hence they can be moved backward (*i.e.*, no release date is blocking). Finally, thanks to condition (4)

and because the f_u functions are regular, the cost of the resuting schedule is at least as good as the inital one.

Proposition 1 is used as a dominance rule at each node of the search tree. To apply the corresponding rule, the most time-consuming part is to check if condition (4) holds or not. Most often the functions are piecewise linear functions and thus the tests can be easily implemented in time proportional to the number of linear pieces. When two such jobs J_i, J_j are detected, we adjust release dates and deadlines as follows : $r_j \leftarrow \max(r_j, r_i + p_i)$ and $d_i \leftarrow \min(d_i, d_j - p_j)$.

4 Extensions

Following the requirements of ILOG's customers, two kind of extensions have been considered. First, it often happens that machines of the shop floor become *overloaded* and thus some jobs J_i cannot be performed within their time window $[r_i, d_i]$. Some jobs must then be left "unperformed". Second, *setup times and setup costs* have to be taken into account.

4.1 Unperfomed Jobs

Unperformed jobs can be easily modeled within our initial framework. Indeed, we can set the deadline d_i to ∞ and change the function f_i into f'_i so that for $t \leq d_i$, $f'_i(t) = f_i(t)$, and for $t \geq d_i$, $f'_i(t) = u_i$. Provided this change is made, the unperformed jobs are now performed very late and it is easy to see that the models are equivalent.

However, our initial Branch and Bound procedure does not perform well on such problems since we try to order all jobs, including those that are unperformed. This leads to a huge number of unnecessary nodes since once it is known that some job is unperformed, it can be scheduled arbitrarily late. To take this remark into account, we have slightly modified our Branch and Bound with the following rule: If a job J_i has no deadline (*i.e.*, $d_i = \infty$) and if f'_i is constant after its earliest possible end time (as updated by constraint propagation), then the job is arbitrarily scheduled at some very large time point and hence it is not considered any longer in the search tree.

4.2.1 Constraint Propagation

We rely on the ILOG SCHEDULER mechanism to take into account setup times and costs. For each job J_i, a lower bound on the setup time and cost between its (unknown) predecessor J_j and J_i is maintained, depending on the possible predecessors of J_i. Note that when J_i is ranked, its predecessor J_j is exactly known, and the setup time and cost consequently set to their exact values. When setup times satisfy the triangle inequality, Arc-B-consistency is also achieved on the disjunctive constraint

$$\mathcal{S}_i + p_i + \delta(\phi(J_i), \phi(J_j)) \leq \mathcal{S}_j \text{ or } \mathcal{S}_j + p_j + \delta(\phi(J_j), \phi(J_i)) \leq \mathcal{S}_i.$$

4.2.2 Lower Bound

We have also been able to extend the lower bound (Section 2) to take into account the setup costs. Recall that the first step of the lower bound computation is to compute a vector $(C_{[1]}, C_{[2]}, ..., C_{[n]})$ such that $\forall i, C_{[i]}$ is a lower bound of the ith smallest completion time in any schedule. To achieve this, we allow preemption and jobs are scheduled according to the SRPT (Shortest Remaining Processing Time) rule. Our basic idea is to insert some "idle" time periods in this SRPT schedule to take into account setup times. This mechanism extends [32] and is closely related to [38] although it has been developped independently.

To do so, we first compute a lower bound on the number of setups $s(k)$ and on the total setup time $\tau(k)$ that have to take place before starting the kth job in a schedule. To compute all $s(k)$ values, we compute for each family $1, ..., q$ the number of jobs in the family $n_1, ..., n_q$. Without any loss of generality, we assume that $n_1 \geq n_2 \geq ... \geq n_q$. Given this notation, it is clear that in any schedule, at least 1 setup must occur before starting the $(n_1 + 1)$th job, 2 setups must occur before starting the $(n_1 + n_2 + 1)$th job, etc. Hence,

$$\forall 1 \leq f \leq q, \forall 1 \leq i \leq n_f, s(\sum_{u=1}^{u<f} n_u + i) = f - 1.$$

Given this definition of the $s(k)$ values, we can define $\tau(k)$ as follows: $\tau(k+1) = \tau(k)$ if $s(k+1) = s(k)$ and otherwise $\tau(k+1)$ equals $\tau(k)$ plus the $s(k+1)$th smallest value in the setup matrix δ.

Now we come back to our initial question and we define $C_{[k]}$ as the minimum time at which the kth job is completed in a schedule that contains $\tau(k)$ idle time points between the minimum release date and the completion of the kth

job. Given this definition of $C_{[k]}$, it is easy to see that in an optimal schedule (for $C_{[k]}$), all jobs can be shifted to the right so that we have no idle time once the first job has started. Hence, we can compute $C_{[k]}$ as follows: (1) Wait $\tau(k)$ units of time from the minimum release date $\min_i r_i$ up to $\min_i r_i + \tau(k)$ and (2) apply the SRPT dispatching rule. The completion time of the kth job in this schedule is optimal.

The above algorithm requires to run the SRPT dispatching rule each time an idle interval is inserted. Since there are at most $q \leq n$ distinct values of $\tau(k)$, no more than q SRPT schedules are relevant. Each SRPT schedule can be computed in $O(n \log n)$, hence the overall complexity is $O(qn \log n)$.

4.2.3 Dominance Properties

The dominance rule proposed in Section 3 is easy to extend to jobs in the same family.

Proposition 2 *If J_i dominates J_j, if $[r_j + p_j, d_i]$ does not contain a release date and if J_i and J_j belong to the same family then, there is an optimal schedule in which J_i precedes J_j.*

4.2.4 No Good Recording

The situation is slightly more complex since the family of the last job in a partial sequence has an impact on the remaining jobs. So, on top of the set of jobs, the completion time of the last job and the cost associated to the partial sequence, we also store the family of the last job in the sequence. The "No Good" test then works as follows: If later in the search tree we have a partial sequence including the same set of jobs as in the No Good sequence that does not improve the completion time of its last job nor its total cost and if the family of the last job in the No Good sequence is the same as the family of the last job in the current sequence then we immediately backtrack.

5 Experimental Results

We have tested four variants of our Branch and Bound algortihm:

- Either we use the lower bound (LB) or not (NO-LB) and
- either we use the No Good Recording (NG) technique or not (NO-NG).

We have run our tests on several instances from the Manufacturing Scheduling Library (MaScLib) [29] available at `www2.ilog.com/masclib`. These instances

are inspired by industrial real life situations and they have been made available to the research community to facilitate manufacturing scheduling research by providing an industrial basis to test scheduling algorithms and through that increase overall interest in manufacturing scheduling research. `NCOS` instances assume no setup while `STC_NCOS` instances assume setup times and costs. Two variants have been considered: In the first one all activities have to be performed while in the other one, activities can be unperformed. The instances we have considered are the ones in which the overall objective to minimize is the sum of processing costs, setup costs, tardiness costs (with respect to ideal due-dates), and non-performance costs, as defined in [29]. Our Branch and Bound algortihm does not apply to MaScLib instances with earliness costs, as earliness costs lead to non-regular cost functions.

Instances have been grouped according to their size and for each group, we have reported the number of instances solved among all instances of the group (SLVD) and the average relative GAP% between the upper bound and the lower bound when the search is stopped, *i.e.*, either when the optimal solution is found or after 30 minutes of CPU time on a 1.4 GHz PC running Windows XP.

The combination of the lower bound and of the No Good Recording proves to be very useful both in terms of number of instances solved and of gap reduction. To further evaluate the effect of LB and NG we report in Table 2 the average CPU time and the average number of backtracks for instances of the problem that are solved by all versions of our Branch and Bound procedure.

Few instances of the MaScLib have a special structure (no deadline, no setup, no unperformed jobs, simple objective functions) and hence they can be considered as instances of $1|r_i|\sum w_iT_i$. For this later problem Jouglet has proposed a very efficient Branch and Bound procedure [25] that incorporates lower bounds, strong dominance properties and specific branching strategies that we cannot use in our general Branch and Bound. Still, we have been able to compare the efficiency of the two procedures. Among 30 instances, our procedure is able to solve 22 of them that are also solved by Jouglet's specific procedure. However we need, on the average, 30 seconds of CPU time and 1707 backtracks while Jouglet's procedure requires 3 seconds and 215 backtracks only. Also note that, within 30 minutes, two more instances are solved by Jouglet. As mentioned earlier, our Branch and Bound tackles a much more complex problem than $1|r_i|\sum w_iT_i$ so the fact that it does not perform as well as specific techniques is not very surprising.

Tables 3, 4, 5 and 6 provide detailed results. For each instance, the tables provide the best-known lower bound (Best LB), the best known upper bound (Best UB), and the lower and upper bounds (LB and UB) provided by our Branch and Bound with lower bounds and nogoods. The tables also provide

NCOS INSTANCES

Size	NO-LB, NO-NG		NO-LB, NG		LB, NO-NG		LB, NG	
	SLVD	GAP%	SLVD	GAP%	SLVD	GAP%	SLVD	GAP%
$10 \leq n \leq 30$	16/20	1.1	20/20	0.0	20/20	0.0	20/20	0.0
$75 \leq n \leq 90$	0/6	134.7	2/6	134.0	0/6	115.5	2/6	77.9
$200 \leq n \leq 500$	0/4	21.4	0/4	21.4	0/4	18.9	0/4	18.9

STC_NCOS INSTANCES

Size	NO-LB, NO-NG		NO-LB, NG		LB, NO-NG		LB, NG	
	SLVD	GAP%	SLVD	GAP%	SLVD	GAP%	SLVD	GAP%
$10 \leq n \leq 30$	2/4	18.9	3/4	16.0	2/4	14.5	3/4	1.4
$75 \leq n \leq 90$	2/6	292.3	2/6	292.3	2/6	197.4	2/6	193.5
$200 \leq n \leq 500$	0/4	587.4	0/4	587.4	0/4	321.3	0/4	321.3

NCOS INSTANCES (WITH UNPERFORMED)

Size	NO-LB, NO-NG		NO-LB, NG		LB, NO-NG		LB, NG	
	SLVD	GAP%	SLVD	GAP%	SLVD	GAP%	SLVD	GAP%
$10 \leq n \leq 30$	14/20	32.4	15/20	29.0	14/20	16.2	15/20	12.7
$75 \leq n \leq 90$	0/6	143.8	0/6	143.8	0/6	142.3	0/6	142.3
$200 \leq n \leq 500$	0/4	1371.1	0/4	1371.1	0/4	33.8	0/4	33.8

STC_NCOS INSTANCES (WITH UNPERFORMED)

Size	NO-LB, NO-NG		NO-LB, NG		LB, NO-NG		LB, NG	
	SLVD	GAP%	SLVD	GAP%	SLVD	GAP%	SLVD	GAP%
$10 \leq n \leq 30$	2/4	40.7	2/4	24.4	2/4	22.5	2/4	22.5
$75 \leq n \leq 90$	2/6	235.4	2/6	235.4	2/6	222.2	2/6	222.2
$200 \leq n \leq 500$	0/4	2152.1	0/4	2152.1	0/4	399.2	0/4	399.2

Table 1
Experimental Results on 88 instances from MaScLib

	NO-LB, NO-NG		NO-LB, NG		LB, NO-NG		LB, NG	
	CPU	BCK	CPU	BCK	CPU	BCK	CPU	BCK
`NCOS` (No Unp.)	6.3	5632	2.9	1332	20.1	5632	9.9	1332
`STC_NCOS` (No Unp.)	0.4	974	0.2	144	0.6	774	0.3	127
`NCOS` (Unp.)	0.7	1763	0.1	161	0.6	1276	0.1	131
`STC_NCOS` (Unp.)	172.3	72156	66.1	8391	186.9	71797	72.8	8388

Table 2
Average Number of Backtracks and Average CPU Time for instances solved by all versions of the Branch and Bound

the origin of the best known lower and upper bounds, *i.e.*:

- "LB,NG" for our Branch and Bound algorithm with lower bounds and no-goods;
- "Jouglet" for the Branch and Bound algorithm presented in [25];
- "Sourd" for the local search algorithm presented in [39];
- "ILOG Unp." for an unpublished constraint programming and local search algorithm developed by T. Bousonville, F. Focacci and D. Godard at ILOG.

Our Branch and Bound algorithm provides the best-known lower bound for 82 instances out of 88 and the best know upper bound for 50 instances. In many cases, however, the solutions found by other algorithms are much better than the solutions found by our Branch and Bound. This reflects the fact that we still have to find good heuristics to explore the search space.

6 Conclusion

In this paper, we have presented a Branch and Bound procedure for the one machine scheduling problem with:

- Release dates and deadlines.
- Costs dependent on the completion times of activities.
- Possibilities of leaving some jobs "unperformed".
- Setup times and costs.

To our knowledge, this is the first exact procedure for such a general problem. Further work includes the development of heuristics to more efficiently explore the search space and the generalization of this procedure to more complex scheduling problems, *e.g.*, to a multi-machine environment.

We tested our procedure on 88 instances of the Manufacturing Scheduling Library (MaScLib) [29], available at `www2.ilog.com/masclib`. We closed 46

Instance	Size	Best LB	Origin	Best UB	Origin	LB	UB
NCOS_01	8	800	LB, NG	800	LB, NG	800	800
NCOS_01a	8	800	LB, NG	800	LB, NG	800	800
NCOS_02	10	3220	LB, NG	3220	LB, NG	3220	3220
NCOS_02a	10	1420	LB, NG	1420	LB, NG	1420	1420
NCOS_03	10	6780	LB, NG	6780	LB, NG	6780	6780
NCOS_03a	10	1780	LB, NG	1780	LB, NG	1780	1780
NCOS_04	10	1011	LB, NG	1011	LB, NG	1011	1011
NCOS_04a	10	1008	LB, NG	1008	LB, NG	1008	1008
NCOS_05	15	1500	LB, NG	1500	LB, NG	1500	1500
NCOS_05a	15	1500	LB, NG	1500	LB, NG	1500	1500
NCOS_11	20	2022	LB, NG	2022	LB, NG	2022	2022
NCOS_11a	20	2006	LB, NG	2006	LB, NG	2006	2006
NCOS_12	24	7966	LB, NG	7966	LB, NG	7966	7966
NCOS_12a	24	5183	LB, NG	5183	LB, NG	5183	5183
NCOS_13	24	5648	LB, NG	5648	LB, NG	5648	5648
NCOS_13a	24	4024	LB, NG	4024	LB, NG	4024	4024
NCOS_14	25	7510	LB, NG	7510	LB, NG	7510	7510
NCOS_14a	25	3230	LB, NG	3230	LB, NG	3230	3230
NCOS_15	30	3052	LB, NG	3052	LB, NG	3052	3052
NCOS_15a	30	3035	LB, NG	3035	LB, NG	3035	3035
NCOS_31	75	9540	LB, NG	9950	Sourd	9540	22990
NCOS_31a	75	9045	LB, NG	9230	Sourd	9045	15770
NCOS_32	75	22560	LB, NG	22560	LB, NG	22560	22560
NCOS_32a	75	15030	LB, NG	15030	LB, NG	15030	15030
NCOS_41	90	9536	LB, NG	13501	Sourd	9536	30553
NCOS_41a	90	9602	LB, NG	10536	Sourd	9602	12658
NCOS_51	200	38220	Jouglet	38220	Jouglet	27940	38430
NCOS_51a	200	38220	Jouglet	38220	Jouglet	27940	38430
NCOS_61	500	1271345	Jouglet	1271345	Jouglet	1268756	1273760
NCOS_61a	500	1570382	Jouglet	1570382	Jouglet	1568000	1571902

Table 3
Detailed Results on NCOS Instances

Instance	Size	Best LB	Origin	Best UB	Origin	LB	UB
NCOS_01	8	800	LB, NG	800	LB, NG	800	800
NCOS_01a	8	800	LB, NG	800	LB, NG	800	800
NCOS_02	10	2740	LB, NG	2740	LB, NG	2740	2740
NCOS_02a	10	1320	LB, NG	1320	LB, NG	1320	1320
NCOS_03	10	6510	LB, NG	6510	LB, NG	6510	6510
NCOS_03a	10	1780	LB, NG	1780	LB, NG	1780	1780
NCOS_04	10	1011	LB, NG	1011	LB, NG	1011	1011
NCOS_04a	10	1008	LB, NG	1008	LB, NG	1008	1008
NCOS_05	15	1500	LB, NG	1500	LB, NG	1500	1500
NCOS_05a	15	1500	LB, NG	1500	LB, NG	1500	1500
NCOS_11	20	2022	LB, NG	2022	LB, NG	2022	2022
NCOS_11a	20	2006	LB, NG	2006	LB, NG	2006	2006
NCOS_12	24	6346	LB, NG	7404	Sourd	6346	8424
NCOS_12a	24	4373	LB, NG	4902	Sourd	4373	5412
NCOS_13	24	2400	LB, NG	4462	Sourd	2400	5122
NCOS_13a	24	2478	LB, NG	3843	Sourd	2478	4089
NCOS_14	25	6214	LB, NG	7140	Sourd	6214	7450
NCOS_14a	25	3230	LB, NG	3230	LB, NG	3230	3230
NCOS_15	30	3052	LB, NG	3052	LB, NG	3052	3052
NCOS_15a	30	3035	LB, NG	3035	LB, NG	3035	3035
NCOS_31	75	6450	LB, NG	9680	Sourd	6450	18340
NCOS_31a	75	7500	LB, NG	9230	Sourd	7500	18035
NCOS_32	75	7500	LB, NG	18180	Sourd	7500	18840
NCOS_32a	75	7740	LB, NG	14890	Sourd	7740	15030
NCOS_41	90	9000	LB, NG	13501	Sourd	9000	31158
NCOS_41a	90	9179	LB, NG	10536	Sourd	9179	12658
NCOS_51	200	23910	LB, NG	36600	Sourd	23910	38430
NCOS_51a	200	23910	LB, NG	36600	Sourd	23910	38430
NCOS_61	500	1195804	LB, NG	1270337	Sourd	1195804	1272736
NCOS_61a	500	1387250	LB, NG	1493152	LB, NG	1387250	1493152

Table 4
Detailed Results on NCOS Instances (with Unperformed)

Instance	Size	Best LB	Origin	Best UB	Origin	LB	UB
STC_NCOS_01	8	1100	LB, NG	1100	LB, NG	1100	1100
STC_NCOS_01a	8	1100	LB, NG	1100	LB, NG	1100	1100
STC_NCOS_15	30	19385	LB, NG	19992	Sourd	19385	20456
STC_NCOS_15a	30	5695	LB, NG	5695	LB, NG	5695	5695
STC_NCOS_31	75	6615	LB, NG	6615	LB, NG	6615	6615
STC_NCOS_31a	75	7590	LB, NG	7590	LB, NG	7590	7590
STC_NCOS_32	75	18464	LB, NG	38386	ILOG Unp.	18464	40456
STC_NCOS_32a	75	14494	LB, NG	16798	ILOG Unp.	14494	16908
STC_NCOS_41	90	11271	LB, NG	46802	Sourd	11271	108371
STC_NCOS_41a	90	10717	LB, NG	18978	Sourd	10717	28254
STC_NCOS_51	200	45890	LB, NG	362535	ILOG Unp.	45890	389250
STC_NCOS_51a	200	70540	LB, NG	390540	Sourd	70540	443890
STC_NCOS_61	500	1426196	LB, NG	1495045	LB, NG	1426196	1495045
STC_NCOS_61a	500	1769960	LB, NG	1821085	LB, NG	1769960	1821085

Table 5
Detailed Results on STC_NCOS Instances

of these instances. Note that when jobs can be unperformed or are subjected to setup constraints, small instances with 24 or 30 jobs are still open. We hope this will encourage other researchers to tackle the problem described in this paper.

Acknowledgment

The authors are grateful to Filippo Focacci and Antoine Jouglet for many enlightening discussions on manufacturing scheduling.

References

[1] T.S. Abdul-Razacq, C.N. Potts, and L.N. Van Wassenhove. A survey of algorithms for the single machine total weighted tardiness scheduling problem. *Discrete Applied Mathematics*, 26:235–253, 1990.

[2] M.S. Akturk and D. Ozdemir. An exact approach to minimizing total weighted tardiness with release dates. *IIE Transactions*, 32:1091–1101, 2000.

Instance	Size	Best LB	Origin	Best UB	Origin	LB	UB
STC_NCOS_01	8	920	LB, NG	920	LB, NG	920	920
STC_NCOS_01a	8	1010	LB, NG	1010	LB, NG	1010	1010
STC_NCOS_15	30	13774	LB, NG	18035	Sourd	13774	22321
STC_NCOS_15a	30	5035	LB, NG	5695	Sourd	5035	6449
STC_NCOS_31	75	6615	LB, NG	6615	LB, NG	6615	6615
STC_NCOS_31a	75	7590	LB, NG	7590	LB, NG	7590	7590
STC_NCOS_32	75	7500	LB, NG	25048	Sourd	7500	25774
STC_NCOS_32a	75	10460	LB, NG	16798	ILOG Unp.	10460	16908
STC_NCOS_41	90	9029	LB, NG	43827	Sourd	9029	85378
STC_NCOS_41a	90	9503	LB, NG	18913	Sourd	9503	26828
STC_NCOS_51	200	33975	LB, NG	230765	Sourd	33975	308770
STC_NCOS_51a	200	36770	LB, NG	251180	Sourd	36770	318740
STC_NCOS_61	500	1339268	LB, NG	1495045	LB, NG	1339268	1495045
STC_NCOS_61a	500	1661000	LB, NG	1818085	Sourd	1661000	1821085

Table 6
Detailed Results on STC_NCOS Instances (with Unperformed)

[3] M.S. Akturk and D. Ozdemir. A new dominance rule to minimize total weighted tardiness with unequal release dates. *European Journal of Operational Research*, 135:394–412, 2001.

[4] Ph. Baptiste, J. Carlier, and A. Jouglet. A branch and bound procedure to minimize total tardiness on one machine with arbitrary release dates. *to appear in European Journal of Operational Research*, 2002.

[5] Ph. Baptiste, C. Le Pape, and L. Peridy. Global constraints for partial csps: A case study of resource and due-date constraints. In *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming, Pisa*, 1998.

[6] Ph. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-based Scheduling*, volume 39 of *ISOR*. Kluwer Academic Publishers, 2001.

[7] Ph. Baptiste, L. Peridy, and E. Pinson. A branch and bound to mininimze the number of late jobs on a single machine with release time constraints. *European Journal of Operational Research*, 144:1–11, 2003.

[8] H. Belouadah, M.E. Posner, and C.N. Potts. Scheduling with release dates on a single machine to minimize total weighted completion time. *Discrete Applied Mathematics*, 36:213–231, 1992.

[9] L. Bianco and S. Ricciardelli. Scheduling of a single machine to minimize total weighted completion time subject to release dates. *Naval Research Logistics*, 29:151–167, 1982.

[10] Peter Brucker. *Scheduling Algorithms*. Springer, 2001.

[11] J. Carlier and E. Pinson. A practical use of jackson's preemptive schedule for solving the job-shop problem. *Annals of Operations Research*, 26:269–287, 1990.

[12] S. Chand, R. Traub, and R. Uzsoy. Single-machine scheduling with dynamic arrivals: Decomposition results and an improved algorithm. *Naval Research Logistics*, 43:709–716, 1996.

[13] R. Chandra. On $n/1/\overline{F}$ dynamic determistic systems. *Naval Research Logistics*, 26:537–544, 1979.

[14] S. Chang, Q. Lu, Tang G., and W. Yu. On decomposition of the total tardiness problem. *Operations Research Letters*, 17:221–229, 1995.

[15] C. Chu. A branch and bound algorithm to minimize total flow time with unequal release dates. *Naval Research Logistics*, 39:859–875, 1991.

[16] C. Chu. A branch and bound algorithm to minimize total tardiness with different release dates. *Naval Research Logistics*, 39:265–283, 1992.

[17] C. Chu. Efficient heuristics to minimize total flow time with release dates. *Operations Research Letters*, 12:321–330, 1992.

[18] C. Chu and M.C. Portmann. Some new efficient methods to solve the $n|1|r_i|\sum T_i$ scheduling problem. *European Journal of Operational Research*, 58:404–413, 1991.

[19] S. Dauzère-Pérès and M. Sevaux. A branch and bound method to minimize the number of late jobs on a single machine. Technical report, Research report 98/5/AUTO, Ecole des Mines de Nantes, 1998.

[20] D.S. Deogun. On scheduling with ready times to minimize mean flow time. *Comput. J.*, 26:320–328, 1983.

[21] M.I. Dessouky and D.S. Deogun. Sequencing jobs with unequal ready times to minimize mean flow time. *SIAM J. Comput.*, 10:192–202, 1981.

[22] J. Du and J.Y.T. Leung. Minimizing total tardiness on one processor is NP-Hard. *Mathematics of Operations Research*, 15:483–495, 1990.

[23] H. Emmons. One-machine sequencing to minimize certain functions of job tardiness. *Operations Research*, 17:701–715, 1969.

[24] A.M.A Hariri and C.N. Potts. An algorithm for single machine sequencing with release dates to minimize total weighted completion time. *Discrete Applied Mathematics*, 5:99–109, 1983.

[25] A. Jouglet, Ph. Baptiste, and J. Carlier. *Handbook of Scheduling: Algorithms, Models and Performance Analysis*, chapter Branch-and-Bound Algorithms for Total Weighted Tardiness. CRC Press, 2004.

[26] J. Labetoulle, E.L. Lawler, J.K. Lenstra, and A.H.G Rinnooy Kan. *Progress in Combinatorial Optimization*, chapter Preemptive scheduling of uniform machines subject to release dates. Academic Press, New York, 1984.

[27] E.L. Lawler. A pseudo-polynomial algorithm for sequencing jobs to minimize total tardiness. *Annals of Discrete Mathematics*, 1:331–342, 1977.

[28] O. Lhomme. Consistency techniques for numeric CSPs. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*,

1993.

[29] W. Nuijten, T. Bousonville, F. Focacci, D. Godard, and C. Le Pape. Towards an industrial manufacturing scheduling problem and test bed. In *Proceedings of PMS'04, Nancy*, 2004.

[30] C.N. Potts and L.N. Van Wassenhove. A decomposition algorithm for the single machine total tardiness problem. *Operations Research Letters*, 26:177–182, 1982.

[31] R.M.V. Rachamadugu. A note on weighted tardiness problem. *Operation Research*, 35:450–452, 1987.

[32] G.L. Ragatz. A branch-and-bound method for minimum tardiness sequencing on a single processor with sequence dependent setup times. In *Twenty-fourth Annual Meeting of the Decison Sciences Institute*, 1993.

[33] G. Rinaldi and A. Sassano. On a job scheduling problem with different ready time: Some properties and a new algorithm to determine the optimal solution. *Operation Research*, 1977. Rapporto dell'Ist. di Automatica dell'Universita di Roma e del C.S.S.C.C.A.-C.N.R.R, Report R.77-24.

[34] A.H.G. Rinnooy Kan. *Machine sequencing problem: classification, complexity and computation*. Nijhoff. The Hague, 1976.

[35] A.H.G. Rinnooy Kan, B.J Lageweg, and J.K. Lenstra. Minimizing total costs in one-machine scheduling. *Operation Research*, 23:908–927, 1975.

[36] R. Sadykov. A hybrid branch-and-cut algorithm for the one-machine scheduling problem. In *Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems*, 2004.

[37] W.E. Smith. Various optimizers for single stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.

[38] A. Souissi, I. Kacem, and C. Chu. New lower bound for minimizing total tardiness on a single machine with sequence-dependent setup times. In *Proceedings of the Ninth International Conference on Project Management and Scheduling, PMS04*, 2004.

[39] F. Sourd. Earliness-tardiness scheduling with setup considerations. *Computers & Operations Research*, To appear.

[40] W. Szwarc, F. Della Croce, and A. Grosso. Solution of the single machine total tardiness problem. *Journal of Scheduling*, 2:55–71, 1999.

[41] W. Szwarc, A. Grosso, and F. Della Croce. Algorithmic paradoxes of the single machine total tardiness problem. *Journal of Scheduling*, 4:93–104, 2001.