Tabu search for a single machine scheduling problem with rejected jobs, setups and deadlines

Simon Thevenin, Nicolas Zufferey, Marino Widmer

9^{th} *International Conference of Modeling, Optimization and Simulation - MOSIM'12*
June 6-8, 2012 - Bordeaux - France
"Performance, interoperability and safety for sustainable development"

Tabu search for a single machine scheduling problem with rejected jobs, setups and deadlines

Simon Thevenin, Nicolas Zufferey
Faculty of Economics and Social Sciences
HEC - University of Geneva
1211 Geneva 4, Switzerland
simon.thevenin@unige.ch, nicolas.zufferey-hec@unige.ch

Marino Widmer
University of Fribourg - DIUF
Decision Support & Operations Research
1700 Fribourg, Switzerland
marino.widmer@unifr.ch

ABSTRACT: *This paper addresses a single machine scheduling problem with release dates, deadlines, setup costs and times, and the possibility to reject some jobs while encountering an abandon cost. The objective function to minimize is a sum of regular functions depending on the completion time of the jobs. The problem is inspired by a manufacturing scheduling problem. We design a greedy algorithm and a tabu search approach for the problem. We studied several restriction procedures. Realistic instances with up to 500 jobs are tackled.*

KEYWORDS: *Tabu search, single machine scheduling*

1 Introduction and literature review

In this paper, we consider the problem of scheduling n jobs on a single machine in order to minimize regular (i.e. non decreasing) objective functions. We take into account setup costs (c_{ij}) and setup times (s_{ij}). There is the possibility to reject some jobs, they are then said unperformed and a penalty cost (u_j) must be paid. For each job, a release date (r_j) and a deadline ($\bar{d}_j$) are given. Using the three field notation the problem can be denoted $(1|r_j, s_{i,j}, \bar{d}_j|\sum f_j(C_j), \sum u_j, \sum c_{i,j})$. For sake of simplicity, we call this problem (P) in the reminder of this paper. We design a greedy algorithm and a tabu search for (P).

In the literature, the two most popular regular objective functions are the sum of completion times ($\sum C_j$), the sum of tardiness ($\sum T_j$) and their weighted versions ($\sum w_j \cdot T_j$ and $\sum w_j \cdot C_j$). (Du and Leung, 1990) showed that the single machine scheduling problem where the objective function is to minimize the sum of weighted tardiness ($1||\sum w_j T_j$) is NP-hard. As it is a particular case of (P), (P) is NP-hard too. The one machine scheduling problem which aims to minimize the sum of completion time with release dates ($1|r_j|\sum C_j$) is NP-hard, unless the release dates are all equal, while the same problem without release date ($1||\sum C_j$) and its weighted version ($1||\sum w_j C_j$) are solvable in polynomial time.

(Baptiste and Le Pape, 2005) designed a branch and bound algorithm in a constraint programming framework to solve (P). They proposed a lower bound and a dominance rule. Their branch and bound was able to solve instances with up to 30 jobs, even if some instances with 24 jobs are still open. To our knowledge, it is the only paper addressing such a problem.

Some related problems have been studied and are described in the following literature review. In (Oğuz et al., 2010), the authors studied the one machine scheduling problem with release dates, deadlines, the possibility to reject some jobs, and sequence dependant setup times. The objective is to maximize the revenue, which is the sum of the gains associated with each performed job minus a weighted tardiness penalty. This problem differs from (P) by the fact that setup costs are not taken into account, and in (P) we consider general cost functions. The authors propose a MILP (mixed integer linear programming) formulation for the problem, which is able to solve instances with up to 15 jobs, as well as different heuristics.

For scheduling problems, we often encounter dispatching rules. It permits to define a priority order for the jobs. In some particular cases, dispatching rules give an optimal sequence of jobs, while in other cases they are greedy heuristics and permit to rapidly obtain relatively good solutions. For the one machine scheduling problem which aims to minimize the weighted tardiness where each job is subject to a release date ($1|r_j, s_{ij}|\sum(w_j T_j)$), the ATCS (apparent tardiness cost with setups) dispatching rule was defined in (Chang et al., 2004). (Shin et al., 2002) made an adaptation of the rule for the case where the objective function is the maximum lateness ($1|r_j, s_{ij}|L_{max}$) called MATCS (modified apparent tardiness cost with setups). In (Oğuz et al., 2010), MATCS for a prob-

lem related to (P) is proposed as well as a dynamic dispatching rule.

(Yang and Geunes, 2007) were interested in the single machine scheduling problem, with the possibility of not performing some jobs. A global deadline is given and jobs cannot be scheduled after it. Moreover, the processing time of each job can be reduced by a compression. The objective function to maximize is the profit of each performed job minus compression costs and tardiness costs. After having introduced a timing algorithm for the problem permitting to find, given a sequence of jobs, the optimal processing time of each job, they introduced two heuristics for the problem. The first heuristic is a GRASP algorithm where a schedule is built by a randomized dispatching rule, then the compression level of each job is obtained by applying the timing algorithm. The second heuristic is an approximation algorithm obtained by the adaptation of an algorithm for the intervals selection problem.

Local search methods are often effective for scheduling problems. (Laguna et al., 1991) studied several neighborhood structures to solve the single machine scheduling problem to minimize the sum of delay and setup costs. The neighborhood *Reinsert* consists in taking a job in the schedule and moving it to another position. The neighborhood *Swap* consists of swapping two jobs in the schedule. They conclude that *Reinsert* is better than *Swap*, but that using an *Hybrid* neighborhood yields better results. *Hybrid* is the union of *Swap* and *Reinsert*.
(Jouglet et al., 2008) studied the single machine problem with release date and several objective functions. They compared different greedy heuristics, and designed an effective tabu search by using dominance rules. They showed that using dominance rules can improve tabu search.
(Bożejko, 2010) used a distributed scatter search algorithm to solve the one machine scheduling problem with setup times to minimize the weighted tardiness ($1|s_{ij}|\sum w_jT_j$). At each step, path relinking is used to select the next solution in the neighborhood of the current solution. He obtained good results.
(Shin et al., 2002) were interested in the one machine scheduling problem with setup times and release date to minimize the maximum lateness ($1|r_j, s_{ij}|L_{max}$). They introduced a tabu search algorithm using *Reinsert*, *Swap*, *Hybrid*, and used a restriction: only the jobs scheduled before the job responsible of L_{max} can be moved. This is justified by the fact that moving others jobs will not decrease L_{max}. Moreover only the jobs contained in a subsequence of the schedule are allowed to move. The first job of this subsequence is called *search line*, and the size of the subsequence is fixed. At each iteration the current search line moves.
(Liao and Juan, 2007) designed an ant colony algorithm for the single machine problem with sequence dependent setups to minimize the weighted tardiness.
(Zufferey et al., 2008) proposed a tabu search algorithm and an adaptive memory algorithm for a satellite range scheduling problem with time windows, were the number of unperformed jobs has to be minimized. They took advantage of the graph coloring literature to solve their problem.

The readers interested in an overview of scheduling problem are referred to (Pinedo, 2008). The reminder of the paper is organized as follows: in Section 2, a formal description and an integer linear programming formulation of (P) are given. In Section 3, a greedy algorithm is introduced, while Section 4 describes a tabu search approach for (P). Results are given in Section 5. A conclusion ends up the paper.

2 Formal description of problem (P)

We are interested in a one machine scheduling problem in which n jobs have to be scheduled. For each job a release date r_j is given: it is the time from which it is possible to start processing job j. In manufacturing, it can be the time at which the raw material is expected to be delivered. Deadlines $\bar{d}_j$ are given: a job cannot end after its deadline. The deadline $\bar{d}_j$ is different from the due date d_j. Scheduling a job after its deadline is not possible. It may for instance be the time after which the penalty cost of scheduling late is higher than the cost of not performing the job, or the time after which the customer does not want to be served. Thus each job must be performed within a time window $[r_j; \bar{d}_j]$. Jobs are allowed to be unperformed: in this case a fixed penalty u_j is paid.
Jobs belong to different families. When the machine successively processes two jobs of different families, a setup must be performed. This implies a setup time s_{ij}, which is the time to tune the machine, and a setup cost c_{ij} (to pay employees which setup the machine and the needed material). At the beginning, the machine is in an initial state, which we represent by a dummy job 0 such that $p_0 = 0$. s_{0j} (resp. c_{0j}) is the necessary setup time (resp. cost) between the initial state and job j, which must be taken into account if j is scheduled in first position. The objective function $\sum_j f_j(C_j)$ is a sum, over all the jobs, of regular (i.e. non decreasing) functions depending on the completion times C_j. We consider general cost functions, allowing our algorithm to solve different problems, or even to use different cost functions for the different jobs.

We present now an integer linear formulation for (P), for which we assume that:

- $x_{jk} = 1$ if job k follows job j, 0 otherwise;
- $z_j = 1$ if j is unperformed, 0 otherwise;
- t_j is the starting time of job j.

For the need of the formulation, we artificially add a last job $n+1$. We have $p_{n+1} = 0$, $s_{(j)(n+1)} = 0, c_{(j)(n+1)} = 0, \forall\ j$.

$$\min \left[\sum_{j=0}^{n} \sum_{k=1}^{n+1} x_{jk} c_{jk} + \sum_{j=1}^{n} [f_j(C_j) + z_j(u_j - f(r_j))\,] \right] \quad (1)$$

s.t.

$$C_j = t_j + p_j \qquad \forall j \quad (2)$$

$$t_j \geq C_k + s_{kj}\, x_{kj} + (x_{kj} - 1)\, \bar{d}_k \quad \forall j, k \quad (3)$$

$$t_j \geq r_j \text{ and } C_j \leq \bar{d}_j + z_j\, (r_j - \bar{d}_j) \quad \forall j \quad (4)$$

$$z_j + \sum_{k=1}^{n+1} x_{jk} = 1 \qquad \forall j \neq n+1 \quad (5)$$

$$z_j + \sum_{k=0}^{n} x_{kj} = 1 \qquad \forall j \neq 0 \quad (6)$$

$$\sum_{j=0}^{n+1} x_{j0} = 0 \text{ and } \sum_{j=0}^{n+1} x_{(n+1)j} = 0 \text{ and } t_0 = 0 \quad (7)$$

$$x_{jk} \in \{0,1\} \qquad z_j \in \{0,1\} \qquad \forall j \quad (8)$$

Equation (1) gives the objective function: it is the sum of setup costs, and, for each job, its cost $f_j(C_j)$ if it is performed, and its unperformed penalty u_j otherwise. The completion time of each unperformed job is virtually set to its release date r_j by equation (4). For this reason in equation (1), for each unperformed job, we add its unperformed cost u_j and remove the cost $f(r_j)$ associated with its release date. (2) gives the completion time. (3) is used to compute the starting time of each job. If j follows k, j must start after the end of k plus the required setup time, otherwise $(x_{kj} - 1)$ is equal to -1, and the resulting constraint is less restrictive than $C_j \geq 0$, as $\bar{d}_k \geq C_k$. (4) enforces each performed job to be scheduled within its time window, and each unperformed job to be scheduled at r_j. (5) and (6) specify that each job must have a successor and a predecessor or being unperformed. (7) constraints job 0 to have no predecessor, job $n+1$ to have no successor, and the starting time of the schedule is 0.

In our model, a solution is represented by an ordered list Π of jobs and a set Ω of unperformed jobs. Let $\pi(p)$ be the index of job scheduled at position p. Thus, a sequence Π with $n' \leq n$ elements can be denoted by $\Pi = [\pi(1), \pi(2), \ldots, \pi(n')]$. Since the objective functions are regular, we can easily build the schedule from Π by setting the starting time of a job to the earliest feasible time as follows:

$$t_{\pi(p)} = \max\{C_{\pi(p-1)} + p_{\pi(p)} + s_{\pi(p-1)\pi(p)},\, r_{\pi(p)}\} \quad (9)$$

Assuming $\pi(0) = 0$, the cost function can then be computed by:

$$\sum_{j \in \Omega} u_j + \sum_{p=1}^{n'} f_{\pi(p)}(C_{\pi(p)}) + \sum_{p=0}^{n'-1} c_{\pi(p)\pi(p+1)} \quad (10)$$

3 Greedy algorithm for (P)

A combinatorial optimization problem is defined by a set of admissible solutions S and an objective function f. A solution is called admissible if it satisfies all the constraints. The goal is to minimize (or maximize) f over S.
For some problems, there exists no algorithm able to find an optimal solution in a polynomial time. Heuristics permit to find satisfying solutions to the problem in a reasonable amount of time. There are three main kinds of heuristics: constructive heuristics, local search methods, and evolutionary algorithms. Metaheuristics are higher level approaches, which try to combine problem specifics methods in order to obtain better solutions. (Blum and Roli, 2003) provide an overview on metaheuristics. For sake of simplicity, we will only use the word heuristic in the reminder of the paper.
A constructive heuristic builds a solution by starting from scratch. It adds elements step by step to the solution until it reaches a complete solution.

In the following, a greedy heuristic for (P) is introduced. It will then be used to generate initial solutions for tabu search, which is a local search method.

The algorithm begins with an empty schedule. Jobs are then taken one by one and placed in the schedule. Each step consists in finding the best place for the considered job j by taking into account only jobs which have already been scheduled. Job j can be inserted between all already scheduled jobs, and at the extremities of the sequence. For every possible place, the cost function is computed, and there is also the possibility of letting the job unperformed. The position which minimizes the cost is chosen. As the jobs are scheduled as early as possible with respect to the sequence, inserting a job before another can shift the entire schedule, thus it is necessary to be careful when computing the cost function at each step. By shifting a part of the schedule to the right, some jobs may end after their deadlines: they will then enter in the unperformed set. Note that the cost function is computed after having shifted and removed the jobs, thus jobs become unperformed only if it is better to do it.

We have to compute the cost function for each job and each possible position, thus the cost function is computed approximately n^2 times. Computing the cost function can be done in $O(n)$ in the worst case. Thus this algorithm runs in $O(n^3)$

The order in which jobs are inserted into the schedule has an influence on the results. After having tested several possibilities to sort the jobs, we decided to sort the jobs by:

- increasing slack times ($\bar{d}_j - r_j - p_j$);
- ties are broken by decreasing u_j;
- if there remain ties, they are broken randomly.

In order to reduce the time spent to reconstruct the schedule and to compute the cost function each time a job is added, we use an incremental cost function. If we add job j at position p, the incremental cost function aims at computing the difference in cost between the old schedule and the new one.

The cost due to the insertion of job j comprises the cost of the job itself (i.e. $f_j(C_j)$), the setup cost and the cost associated with the shifting process. It is straightforward to obtain the difference in setup cost if j is inserted at position p: $c_{\pi(p-1)j} + c_{j\pi(p+1)} - c_{\pi(p-1)(p+1)}$.

We have now to compute the augmentation of the cost Δ due to the shifting process, which we seek to minimize. If C_j denotes the completion time of job j before the insertion, and C'_j the completion time of j after the move, then $\Delta = \sum_{j=1}^{n}(f_j(C'_j) - f_j(C_j))$. By inserting a job j at position p, the completion times of the jobs before position p do not change (i.e. $C_{\pi(x)} = C'_{\pi(x)} \forall x < p$). The jobs after position p are shifted to the right. If there is idle time in the schedule, not all jobs are going to be shifted.

Figure 1 gives an example where adding a job shifts only one job. We assume that job 4 is scheduled at its release date, and there is some idle time between job 3 and 4. Inserting job 6 between jobs 2 and 3 will thus shift job 3 but not job 4. And all the reminder of the schedule keeps the same starting time.

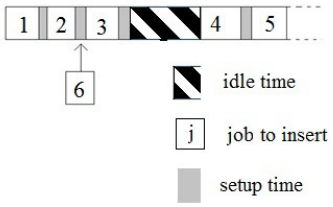

Figure 1: Insertion without an entire shifting

Note that the same thing can happen if a job is dropped during the shifting process. Hence, to compute the incremental cost, we start after the just inserted job, and compute all the completion times of the following jobs. We stop as soon as the starting time of a job remains unchanged. Assuming a job j is inserted at position p, an algorithm is given in Figure 2, which computes the cost associated with the shifting process.

set $x = p$.
compute the completion time of job $\pi(x)$
while the completion time of $\pi(x)$ changes, **do**
  if $\pi(x)$ ends after its dead line, **then**
    add the unperformed cost $u_{\pi(x)}$
    subtract $f_{\pi(x)}(C_{\pi(x)})$
    update the setup cost:
    $-c_{\pi(x-1)\pi(x)} - c_{\pi(x)\pi(x+1)} + c_{\pi(x-1)\pi(x+1)}$
  else
    add $f_{\pi(x)}(C'_{\pi(x)})$ and subtract $f_{\pi(x)}(C_{\pi(x)})$
  end if
  set $x = x + 1$
  compute the completion time of $\pi(x)$.
end while

Figure 2: Incremental cost function

The greedy algorithm still runs in $O(n^3)$ in the worst case, but on the considered manufacturing benchmark instances, the algorithm is in average 20 times faster with the incremental cost function than without.

4 Local search methods for (P)

4.1 Overall consideration

Local search methods need an initial solution, and then explore the solution space by going from the current solution s to a neighbor solution s', which is often obtained by making a slight modification on s. The modification is called a *move*, and the neighborhood $N(s)$ of a solution s is the set of solutions obtained by applying to s all possible moves.

A simple local search is the descent algorithm. Starting from a solution s, the descent algorithm explores all neighbors of s, and set as next current solution the solution in $N(s)$ which minimizes the objective function. The main issue with this method is that it is very likely to bring the search in a local optimum. To overcome this issue, tabu search makes use of recent memory, with a so called tabu list. It forbids to perform the reverse of the moves done during the last t iterations, where t is called tabu tenure. Readers interested by more information on tabu search are referred to (Glover and Laguna, 1999).

To have a good control on the search, the structure of a neighbor solution of s must be close to s, which means that each move should perform a slight modification of the solution. In other words, two neighbor solutions should have a relatively common structure. In addition, the neighborhood structure must allow to reach the optimal solution from any solution s. An

easy way to achieve this goal is to be able to reach any solution from any other solution by performing a sequence of moves.

For some problems and some move definitions, it can take a long time to visit all neighbors at each iteration. To tackle this issue, we can make use of the first improving move strategy: it consists in stopping exploring the neighborhood of the current solution as soon as a solution having a better cost than the current solution is found. Another way is to use restrictions: instead of browsing the whole neighborhood of a solution, a subset is used. The idea is to select, at each iteration, the best solution into the subset. Another aim of using such restrictions is to have a less aggressive and deterministic strategy and avoiding to be quickly trapped in a local optimum.

4.2 Neighborhood structures

We propose to use the following neighborhoods:

- **Neighborhood 1 (*Reinsert* and shift and drop)** moves a job from a position p to any other position p'. Once the move is done, jobs are shifted to be scheduled as early as possible. Jobs can be dropped if they end after their deadlines.
- **Neighborhood 2 (*Swap* and shift and drop)** swaps two jobs. Once the move is done, the schedule is shifted and jobs can be dropped if they end after their deadlines. If *Swap* is jointly used with *Reinsert*, swapping two consecutive jobs is forbidden, because adjacent pairwise interchange is included in *Reinsert*.
- **Neighborhood 3 (*Add* and shift and drop)** adds a job from the unscheduled set at a position p and the schedule is shifted. Jobs can be dropped if they end after their deadlines.
- **Neighborhood 4 (*Drop* and shift)** drops a job and shift the right part of the schedule to the left. Due to the shifting process, dropping a job can reduce the value of the cost function.

None of these neighborhoods can be used alone because a single one does not allow to reach all possible solutions (which means that the search space is not connected). *Add* must be present: it allows, once a job has been dropped, to again visit solutions containing the deleted job. Several papers (e.g. (Laguna et al., 1991) and (Shin et al., 2002)) showed that using hybrid moves (i.e. union of several moves) leads to better results: we will also do it. At each iteration, we apply to the current solution all possible moves (namely *Reinsert*, *Swap*, *Add*, and *Drop*) and we chose the neighbor solution which minimizes the cost.

On the one hand, we develop a descent algorithm using the four neighborhoods. It starts from an initial solution generated by the greedy algorithm. At each iteration, it generates the neighborhood of a solution and takes as next solution the best of them.

On the other hand, we propose a tabu search by adding tabu structures to the above descent algorithm. We design four tabu structures: **(1)** when a job has been added into the schedule, it cannot be dropped during t_1 iterations, and it cannot be moved (by any of the moves) during t_2 iterations; **(2)** when a job has been dropped, it cannot be added during t_3 iterations. Note that, for this tabu structure, the job is considered dropped only by Neighborhood 4 and not by the shift and drop procedure; **(3)** when a job has been reinserted, it cannot be moved during t_2 iterations and it cannot return between its two previous adjacent jobs during t_4 iterations; **(4)** when a job has been swapped, it cannot be moved during t_2 iterations and it cannot return between its two previous neighboring jobs during t_4 iterations. t_1, t_2, t_3, t_4 are parameters of the algorithm. We propose $t_2 < t_4$ because the tabu structure associated with t_2 is more restrictive than the one associated with t_4.

4.3 Other ingredients

In this subsection, we present other ingredients which are useful within the proposed local search methods: a way to quickly evaluate a neighbor solution (incremental cost), a way to reduce the size of the generated neighborhood at each iteration (restrictions), and a powerful tabu tenure which forbids solution values (and not solution attributes).

Incremental cost. The incremental cost function defined in Figure 2 can be adapted to the moves defined above. For instance, if we consider *Reinsert*, it is straightforward to obtain the difference in setup cost due to the move of a job j scheduled from position p to p'. We denote $\pi(p)$ the job at position p in the schedule before *Reinsert*. The setup cost is updated as follows, where the first line is the cost due to removing job j, and the second one is the cost imputable to the insertion of job j at position p'.

$$\begin{aligned} & c_{\pi(p-1)\pi(p+1)} - c_{j\pi(p+1)} - c_{\pi(p-1)j} \\ + \quad & c_{j\pi(p')} + c_{\pi(p'-1)j} - c_{\pi(p'-1)\pi(p')} \end{aligned}$$

The algorithm depicted in Figure 2, which computes the difference due to the shifting of the job, must be applied twice. One time by starting at the position where the job has been removed (position p), and another time by starting where it has been added (position p'). Assuming without loss of generality that

$p' > p$, we must be careful of not computing twice the incremental cost of a job after p'.

Restrictions. The following restriction is applied. We do not schedule a job j at position p if the job currently scheduled at this position ends before the release date of j (i.e. $r_j > C_{\pi(p)}$). If such a move is performed, j is going to be shifted to its release date, bringing with it all jobs scheduled between p and the job being processed at r_j. This leads to useless idle time, and thus higher cost.

We implement a tabu search which uses such restrictions. In addition two other restrictions have been tested: **Random subset** reduces the neighborhood size by using a random subset made of $q_1\%$ (parameter) of the entire neighborhood of the current solution; **First improving move** stops exploring the neighborhood of a solution as soon as an improving move is found (in order to avoid any bias, we randomly scan the neighbor solutions).

TabuCost. Preliminary experiments showed that there exist many solutions in the neighborhood with the same value.

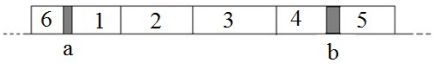

Figure 3: Example of equivalent solution

In the example depicted in Figure 3, we assume that jobs 1, 2, 3 and 4 belong to the same family, thus there is no setup cost between them. We also assume that all jobs have the same release date a, and that the cost function of all jobs (f_j) is constant over $[a, b]$. Jobs 1 to 4 might be positioned in any order before b, the cost remains the same as the beginning of job 5 remains the same, and the reminder of the schedule is unchanged. Note that this situation is specific to the case where the f_j's are constant over some intervals. This has the effect to bring the search into a local optimum. Once the solution cannot be improved, only moves leading to an equivalent solution are performed. To tackle this problem, (Jouglet et al., 2008) propose to associate a tabu status with the value of the recently visited solutions: the solutions which lead to such costs are tabu. This tabu structure has been implemented and is called *TabuCost* in the reminder of the document. We denote the tabu tenure of *TabuCost* by t_5.

5 Experiments

In this section, we first present the way to read the considered benchmark instances, and then describe and discuss the performed experiments.

5.1 Way to read the benchmark instances

The used test bed is called *Manufacturing Scheduling Library* (Le Pape, 2007, Nuijten et al., 2004) and denoted $MaScLib$ in this paper. It is a library containing instances of scheduling problems which are inspired from real manufacturing cases. Such benchmark instances have been made available to the research community by *ILOG*. Each instance contains between 8 and 500 jobs. Thirty of them do not consider setup costs or times (category NCOS), while fourteen assume setup costs and times (STC_NCOS). Note that instances STC_NCOS are not obtained by simply adding setup cost on NCOS instances: the data are very different. In order to facilitate the work of researchers who might want to tackle the same instances, we describe below the way we read the data. We also provide, for instances $NCOS\_02$ and $NCOS\_01$, the associated data and the best solution found by one of the proposed algorithms.

Each $MaScLib$ instance is represented by a set of tables (e.g., *ACTIVITY*, *MODE*, *DUE_DATE*, and *SETUP_MATRIX*). We give below, for each component of the problem, the column where the associated value can be found. If *COLUMN*$[j]$ denotes the value of the attribute *COLUMN* at the row having *ACTIVITY*_ID $= j$, we have from table *MODE*: $p_j =$ *PROCESSING_TIME*$[j]$, $r_j =$ *START_MIN*$[j]$, $\bar{d}_j =$ *END_MAX*$[j]$, and $u_j =$ *UNPERFORMED_COST*$[j]$. From table *DUE_DATE*, we have $d_j =$ *DUE_TIME*$[j]$ and $w_j =$ *TARDINESS_VARIABLE_COST*$[j]$. The family of job j is given in table *ACTIVITY*, in column *SETUP_STATE*$[j]$. The setup times (resp. costs) can be found in table *SETUP_MATRIX*, in column *SETUP_TIME* (resp. *SETUP_COST*). A *MODE_COST* is given in table *MODE*. Looking at the $MaScLib$ tables definition (which were provided with the instances), the mode cost is the "cost associated with the activity when processed in this mode". Thus we add this cost to each performed job, assuming it models a fixed cost which has to be paid in order to process the jobs. Therefore, we have $f_j(C_j) =$ *MODE_COST*$[j] + w_j \cdot \max(0, C_j - d_j)$.

By using the above way of reading the data, it may happen that, in the initial problem (denoted P_1), the cost associated with job j if scheduled at its deadline is lower than the unperformed cost (i.e., $f_j(\bar{d}_j) > u_j$). In other words, it is preferable to reject j rather than to schedule it late. In such a case, a preprocessing procedure is used to create a new problem (denoted P_2) by changing the value of $\bar{d}_j$ such that $f_j(\bar{d}_j) = u_j$. Obviously, any feasible solution of (P_2) is feasible for (P_1), and all optimal solutions of (P_2) are optimal solutions for (P_1). The data associated with instance $NCOS\_01$ (resp. $NCOS\_02$) are given in Table 3 (resp. Table 1). Table 4 gives a feasible solution for instance $NCOS\_01$, and its total cost is 800. Table 2

gives a feasible solution for $NCOS\_02$, and its total cost is 2570.

Table 1: Data read from $NCOS\_02$

j	p_j	r_j	$\bar{dj}$	u_j	$f_j(C_j)$
0	70	0	20000	950	$100 + \max\{0, 3 \cdot (C_j - 680)\}$
1	70	0	20000	950	$100 + \max\{0, 3 \cdot (C_j - 720)\}$
2	90	0	20000	1250	$100 + \max\{0, 7 \cdot (C_j - 640)\}$
3	90	0	20000	1250	$100 + \max\{0, 7 \cdot (C_j - 560)\}$
4	90	0	20000	1250	$100 + \max\{0, 7 \cdot (C_j - 800)\}$
5	70	0	20000	950	$100 + \max\{0, 3 \cdot (C_j - 160)\}$
6	90	0	20000	1250	$100 + \max\{0, 7 \cdot (C_j - 80)\}$
7	90	0	20000	1250	$100 + \max\{0, 7 \cdot (C_j - 120)\}$
8	90	0	20000	1250	$100 + \max\{0, 7 \cdot (C_j - 280)\}$
9	90	0	20000	1250	$100 + \max\{0, 7 \cdot (C_j - 280)\}$

Table 2: A feasible solution for $NCOS\_02$

Performed j	**7**	**5**	**9**	**8**	**3**	**1**	**2**	**0**	**4**
t_j	0	90	160	250	340	430	500	590	660
C_j	90	160	250	340	430	500	590	660	750
$f_j(C_j)$	100	100	100	520	100	100	100	100	100

Rejected j	6
u_j	1250

Table 3: Data read from $NCOS\_01$

j	p_j	r_j	$\bar{d_j}$	u_j	$f_j(C_j)$
0	10	0	1000	100	$100 + \max\{0, 2 \cdot (C_j - 395)\}$
1	30	30	1000	100	$100 + \max\{0, 2 \cdot (C_j - 105)\}$
2	135	315	1000	100	$100 + \max\{0, 2 \cdot (C_j - 840)\}$
3	170	155	1000	300	$100 + \max\{0, 2 \cdot (C_j - 680)\}$
4	45	40	1000	100	$100 + \max\{0, 2 \cdot (C_j - 105)\}$
5	45	40	1000	100	$100 + \max\{0, 2 \cdot (C_j - 900)\}$
6	30	30	1000	100	$100 + \max\{0, 2 \cdot (C_j - 635)\}$
7	170	165	1000	300	$100 + \max\{0, 2 \cdot (C_j - 495)\}$

Table 4: A feasible solution for $NCOS\_01$

Performed j	**6**	**0**	**5**	**3**	**7**
t_j	30	60	70	155	325
C_j	60	70	115	325	495
$f_j(C_j)$	100	100	100	100	100

Rejected j	4	1	2
u_j	100	100	100

5.2 Results

We now present the tested algorithms, each of them aims in underlying the contribution of an ingredient.

Greedy is the algorithm presented in Section 3. In the purpose of comparing *Greedy* with other algorithms, we restart *Greedy* during 10 minutes. The algorithm returns the best encountered value. For all local search algorithms, we used a time limit of 10 minutes (preliminary tests showed that running the algorithms more than 10 minutes does not improve the results).

Descent is implemented as described in Section 4.

Tabu1 is the tabu search obtained by adding tabu lists to the descent algorithm. The used tabu tenures are: $t_1 = 15$, $t_2 = 120$, $t_3 = 12$, $t_4 = 40$, for the large instances (more than 75 jobs), and $t_1 = 1$, $t_2 = 3$, $t_3 = 1$, $t_4 = 2$ for the small ones. Informal tests showed that using more refined tabu tenures did not improve the results.

Tabu2 adds restrictions to *Tabu1*, its purpose is to show the effect of the defined restrictions. We used the first improving move strategy in a random subset of neighbors. The size of the subset is $q_1 = 15\%$.

Tabu3 is an extension of *Tabu2* by adding *TabuCost* to it. We used $t_5 = 40$ for the large instances, and $t_5 = 3$ for the small ones.

The algorithms were implemented in C++ and ran on a computer with processor Intel i7 Quandcore (2.93 GHz RAM 8 Go DDR3). In order to compare our results with (Baptiste and Le Pape, 2005), the cost function is: $f_j = w_j \cdot T_j$, where T_j is the tardiness cost. If we call d_j the due date and C_j the completion time, we have $T_j = \max\{0; C_j - d_j\}$. For each algorithm, we compute the average of the results obtained over 10 runs.

Table 5 contains the results for the instance having setup times and costs. For instances without setups, results can be found in Table 6. The tables contain, in column *Best*, the best result found by a single run of one of our algorithms. Then, for each algorithm, the percentage gap between the average result over the 10 runs and *Best* can be found: $Gap = 100 \cdot \frac{Average - Best}{Best}$. The column *Baptiste* designates upper bound results found in (Baptiste and Le Pape, 2005).

Results show that the most refined version of tabu search is competitive: in average, it improves by approximately 20% the results found in (Baptiste and Le Pape, 2005). The improvements are larger when considering setups. Results also allow to highlight the impact of each proposed ingredient. *Descent* was about 9% better than *Greedy*. *Tabu1* slightly improves *Descent*: the gap is around 0.1% between those methods (such a small gap is probably due to the equivalent solutions issue described when presenting *TabuCost*). Using restrictions, *Tabu2* allows tabu search to perform more iterations, but in counterpart fewer neighbors are evaluated at each iteration. While adjusting the neighborhood restriction process, a balance between intensification (i.e., examination of all the neighbor solutions at each iteration) and diversification (i.e., a significant reduction of the explored neighborhood) must be found: the gap between *Tabu2* and *Tabu1* is around 1.3%. The *TabuCost* ingredient belongs to *Tabu3*: it leads to an improvement of 0.3% over *Tabu2*.

Table 5: Results for instances with setups

Instance	Size	Best	*Baptiste*	*Greedy*	*Descent*	*Tabu1*	*Tabu2*	*Tabu3*
STC_NCOS_01	8	700	31.43	41.43	0.00	0.00	0.00	0.00
STC_NCOS_01a	8	610	65.57	70.49	0.00	0.00	0.00	0.00
STC_NCOS_15	30	17,611	26.74	0.22	0.00	0.00	0.00	0.00
STC_NCOS_15a	30	5,584	15.49	0.05	0.00	0.00	0.00	0.00
STC_NCOS_31	75	6,615	0.00	15.42	3.79	3.36	9.35	4.20
STC_NCOS_31a	75	7,590	0.00	26.10	3.67	2.93	6.85	4.62
STC_NCOS_32	75	24,048	7.18	4.75	2.82	2.65	2.01	0.90
STC_NCOS_32a	75	16,798	0.65	3.92	0.00	0.00	0.00	0.29
STC_NCOS_41	90	44,641	91.25	9.24	0.04	0.04	0.03	0.05
STC_NCOS_41a	90	19,092	40.52	4.30	0.02	0.03	0.03	0.03
STC_NCOS_51	200	139,675	121.06	78.25	54.64	54.64	0.00	0.00
STC_NCOS_51a	200	210,970	51.08	19.57	0.00	0.00	0.00	0.00
STC_NCOS_61	500	1,495,045	0.00	0.00	0.00	0.00	0.00	0.00
STC_NCOS_61a	500	1,814,605	0.36	0.09	0.00	0.00	0.00	0.00
AVERAGE			32.24	19.56	4.64	4.55	1.30	0.72

Table 6: Results for instances without setup

Instance	Size	Best	*Baptiste*	*Greedy*	*Descent*	*Tabu1*	*Tabu2*	*Tabu3*
NCOS_01	8	800	0.00	0.00	0.00	0.00	0.00	0.00
NCOS_01a	8	800	0.00	0.00	0.00	0.00	0.00	0.00
NCOS_02	10	2,570	6.61	12.84	12.84	10.27	0.00	0.00
NCOS_02a	10	1,210	9.09	0.83	0.83	0.74	0.83	0.00
NCOS_03	10	6,460	0.77	11.15	0.00	0.00	0.00	0.00
NCOS_03a	10	1,690	5.33	14.20	0.00	0.00	0.00	0.00
NCOS_04	10	1,011	0.00	0.00	0.00	0.00	0.00	0.00
NCOS_04a	10	1,008	0.00	0.00	0.00	0.00	0.00	0.00
NCOS_05	15	1,500	0.00	0.00	0.00	0.00	0.00	0.00
NCOS_05a	15	1,500	0.00	0.00	0.00	0.00	0.00	0.00
NCOS_11	20	2,022	0.00	6.08	0.00	0.00	0.00	0.00
NCOS_11a	20	2,006	0.00	0.00	0.00	0.00	0.00	0.00
NCOS_12	24	6,844	23.09	9.96	0.00	0.00	0.00	0.00
NCOS_12a	24	4,270	26.74	11.08	0.00	0.00	0.00	0.00
NCOS_13	24	3,912	30.93	19.33	3.94	3.15	0.92	0.00
NCOS_13a	24	3,441	18.83	13.25	0.00	0.00	0.29	0.00
NCOS_14	25	6,990	6.58	0.00	0.00	0.00	0.00	0.00
NCOS_14a	25	3,195	1.10	10.64	0.00	0.00	0.00	0.00
NCOS_15	30	3,052	0.00	0.00	0.00	0.00	0.00	0.00
NCOS_15a	30	3,035	0.00	0.49	0.00	0.00	0.00	0.00
NCOS_31	75	9,550	92.04	7.43	0.52	0.94	2.31	1.43
NCOS_31a	75	8,740	106.35	7.84	0.13	0.00	0.00	2.32
NCOS_32	75	17,310	8.84	4.22	2.20	2.20	1.72	0.00
NCOS_32a	75	14,720	2.11	1.36	0.00	0.00	0.00	0.65
NCOS_41	90	13,483	131.09	34.93	0.35	0.35	0.28	0.11
NCOS_41a	90	10,546	20.03	14.14	0.64	0.58	0.26	0.03
NCOS_51	200	36,170	6.25	5.67	0.08	0.07	0.19	0.00
NCOS_51a	200	36,170	6.25	5.67	0.08	0.07	0.08	0.00
NCOS_61	500	1,269,365	0.27	0.19	0.00	0.00	0.00	0.00
NCOS_61a	500	1,485,232	0.53	0.10	0.00	0.00	0.00	0.00
AVERAGE			16.76	6.38	0.72	0.61	0.23	0.15

6 Conclusion

In this paper, we consider a one machine scheduling problem with release dates, setups, deadlines, and the possibility of leaving jobs unperformed. This is a new problem, which is relevant in practice. It was introduced by Baptiste and Le Pape (2005) and there is no existing heuristic for it. We introduce a integer linear formulation for the problem, a greedy heuristic, and we design a tabu search which is efficient compared to the solution proposed by Baptiste and Le Pape (2005). We proposed and tested different ingredients for tabu search, and demonstrate their efficiency in solving manufacturing case instances.

Future work consists in the development of hybrid metaheuristics for the same problem, and to study some extensions of the problem (e.g. several machines, non regular cost functions).

References

Baptiste, P. and Le Pape, C. (2005). Scheduling a single machine to minimize a regular objective function under setup constraints, *Discrete Optimization* **2**(1): 83–99.

Blum, C. and Roli, A. (2003). Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison, *ACM Computing Surveys* **35 (3)**: 268–308.

Bożejko, W. (2010). Parallel path relinking method for the single machine total weighted tardiness problem with sequence-dependent setups, *Journal of Intelligent Manufacturing* **21**: 777–785.

Chang, T.-Y., Chou, F.-D. and Lee, C.-E. (2004). A heuristic algorithm to minimize total weighted tardiness on a single machine with release dates and sequence-dependent setup times, *Journal of the Chinese Institute of Industrial Engineers* **21**(3): 289–300.

Du, J. . and Leung, J. Y. (1990). Minimizing total tardiness on one machine is NP-hard, *Mathematics of Operations Research* **15**: 483–495.

Glover, F. and Laguna, M. (1999). *Tabu search*, Kluwer.

Jouglet, A., Savourey, D., Carlier, J. and Baptiste, P. (2008). Dominance-based heuristics for one-machine total cost scheduling problems, *European Journal of Operational Research* **184**(3): 879–899.

Laguna, M., Barnes, J. W. and Glover, F. (1991). Tabu search methods for a single machine scheduling problem, *Journal of Intelligent Manufacturing* **2**: 63–73. 10.1007/BF01471219.

Le Pape, C. (2007). A Test Bed for Manufacturing Planning and Scheduling Discussion of Design Principles, *International Workshop on Scheduling a Scheduling Competition*, Providence Rhode Island USA.

Liao, C.-J. and Juan, H.-C. (2007). An ant colony optimization for single-machine tardiness scheduling with sequence-dependent setups, *Computers & Operations Research* **34**(7): 1899–1909.

Nuijten, W., Bousonville, T., Focacci, F., Godard, D. and Pape, C. L. (2004). Towards an industrial manufacturing scheduling problem and test bed, *In Proceedings Project Management and Scheduling*, Nancy, pp. 162–165.

Oğuz, C., Sibel Salman, F. and Bilgintürk Yalçın, Z. (2010). Order acceptance and scheduling decisions in make-to-order systems, *International Journal of Production Economics* **125**(1): 200–211.

Pinedo, M. (2008). *Scheduling: Theory, Algorithms, and Systems*, Springer.

Shin, H. J., Kim, C.-O. and Kim, S.-S. (2002). A tabu search algorithm for single machine scheduling with release times, due dates, and sequence-dependent set-up times, *The International Journal of Advanced Manufacturing Technology* **19**: 859–866.

Yang, B. and Geunes, J. (2007). A single resource scheduling problem with job-selection flexibility, tardiness costs and controllable processing times, *Computers & Industrial Engineering* **53**(3): 420 – 432.

Zufferey, N., Amstutz, P. and Giaccari, P. (2008). Graph Colouring Approaches for a Satellite Range Scheduling Problem, *Journal of Scheduling* **11 (4)**: 263 – 277.