AN O($N \log N$) STABLE ALGORITHM FOR IMMEDIATE SELECTIONS ADJUSTMENTS

Laurent Péridy and David Rivreau
Institut de Mathématiques Appliquées, Université Catholique de l'Ouest, France
{ laurent.peridy,david.rivreau } @ima.uco.fr

Abstract Using local operations within branch-and-bound methods for job-shop scheduling problems has been proved to be very effective. In this paper, we present an efficient algorithm that applies ascendant set-like adjustments for the immediate selections. This procedure is given within an original framework that guarantees a good convergence process and an easy integration of other classical disjunctive elimination rules.

Keywords: disjunctive scheduling, edge finding, local adjustments, elimination rule, job-shop.

INTRODUCTION

Many scheduling problems found in a typical factory environment involve the processing of jobs on a fixed set of machines that can handle at most one job at a time. If we focus on one machine, we are given a set of operations to be processed without interruption in their time windows. The purpose of local adjustments is to narrow these time windows in order to speed up the enumerative approaches used for the whole problem. This kind of elimination rule has been in particular successfully applied to solve to optimality notoriously difficult scheduling problems such as job-shops (Carlier and Pinson, 1989; Brinkkötter and Brucker, 2001). In this paper, we consider the immediate selections due to Carlier (1975) and give an O($n \log n$) procedure that finds all adjustments associated with these selections.

The paper is organised as follows. In the first section, we recall the main classical adjustment procedures and give some properties that entitle to design stable algorithms. Section 2 is devoted to the presentation of the new elimination rule. Then, in Section 3, this procedure is implemented by a stable procedure that it is proved to run in O($n \log n$) time. Finally, we report some experimental results on job-shop in Section 4 and draw some conclusions in Section 5.

1. LOCAL ADJUSTMENTS FOR DISJUNCTIVE PROBLEM

1.1 Disjunctive Scheduling Problem

As mentioned before, we concentrate on the process of a set $\mathcal{O}$ of n operations on a single machine that can process only one operation at a time. Each operation i from $\mathcal{O}$ is given an integer processing time p_i and must be processed in a certain time window $[r_i, d_i]$. No pre-emption is allowed. Therefore, any feasible schedule of $\mathcal{O}$ is characterised by a set $\{t_i\}$ of starting times for operations such that the following two relations hold:

$$\begin{array}{ll} r_i \le t_i \le d_i - p_i & \forall i \in \mathcal{O} \\ (t_i + p_i \le t_j) \vee (t_j + p_j \le t_i) & \forall (i, j) \in \mathcal{O} \times \mathcal{O} \end{array}$$

The main goal of local operations is precisely to reduce the time windows bounds of operations in order to reduce the problem size. Since adjustments of release dates and of deadlines are clearly symmetrical, we will henceforth only consider release date adjustments.

1.2 Local Adjustments

One of the first local adjustments has been proposed by Carlier (1975). This elimination rule attempts to deduce an adjustment from the relative positioning of two given operations i and j. It can be stated as follows:

Immediate selections adjustments (Carlier, 1975). If $r_j + p_j + p_i > d_i$ then i precedes j in any feasible solution. In that case, we can let

$$r_j \leftarrow \max(r_j, r_i + p_i)$$

These immediate selections have been extended by Carlier and Pinson (1989). To this end, they evaluate the relative positioning of an operation i in a given subset J such that $i \notin J$. Three cases are distinguished:

(C1) Operation i cannot be scheduled *before* subset J if

$$r_i + p_i + \sum_{j \in J} p_j > \max_{j \in J} d_j$$

(C2) Operation i cannot be scheduled *inside* subset J if

$$\min_{j \in J} r_j + p_i + \sum_{j \in J} p_j > \max_{j \in J} d_j$$

(C3) Operation i cannot be scheduled *after* subset J if

$$\min_{j \in J} r_j + \sum_{j \in J} p_j + p_i > d_i$$

Carlier and Pinson deduce the so-called ascendant sets adjustments from those conditions:

> *Ascendant sets adjustments (Carlier and Pinson, 1989).* If (C1) and (C2) are satisfied then i is processed after all operations from J in any solution. In that case, we can let
>
> $$r_i \leftarrow \max\left(r_i \,;\, \max_{J' \subseteq J}\left\{\min_{j \in J'} r_j + \sum_{j \in J'} p_j\right\}\right)$$

It can be noticed that the potential ascendant set adjustment of r_i corresponds to the optimal makespan of the pre-emptive schedule of J.

It has been proved by Carlier and Pinson (1990, Theorem 1) that the ascendant set adjustment of i leads to the immediate selection $r_j + p_j + p_i > d_i$ for all $j \in J$. However, some of the precedence relations identified by immediate selections cannot be found by the ascendant sets procedure. It follows that a better adjustment is missed, even when ascendant sets adjustments are used with classical immediate selections.

In the remainder of the paper, we present a procedure that allows us to apply the ascendant set adjustments to all the precedence relations found by immediate selections and, by extension, induced by Carlier and Pinson (1990, Theorem 1), to all precedence relations found by the ascendant sets procedure. To distinguish our immediate selection adjustments from the original version of Carlier, we speak from now on of *improved* immediate selection adjustments.

1.3 Properties

We recall the main concepts given in Péridy and Rivreau (2005) to qualify the properties of local operations and related algorithms. In particular, we focus on the characteristics that allow us to define a class of methods for which several adjustments of release dates can be combined in a single stable pass.

So, let E be the set of n-dimensional vectors of possible release dates for a given one-machine problem. Clearly, any local adjustment can be seen as a function f from E to E. A few questions arise naturally. First of all, is it necessary to apply a local adjustment procedure in several runs to reach the fixpoint of f (in other words, does the local adjustment procedure is stable or not)? How to combine several local adjustment procedures? In what order? These questions have been investigated in Péridy and Rivreau (2005) for the classical adjustment procedures, but in this paper, we are only interested in the first question, since we specifically focus on a single adjustment rule. Nevertheless, it remains the case that improved immediate selections adjustments are easy to integrate in the more general framework defined in Péridy and Rivreau (2005).

The stability of our general framework is based on two properties of the underlying adjustments:

- the adjustments must be *increasing*;
- the adjustments must be *non-anticipative*.

The increasing characteristic is a property defined on the following partial order $\preceq$ on E (which defines $(E, \preceq)$ as a lattice):

$$u \preceq v \quad \Leftrightarrow \quad \forall i \in \mathcal{O}, u_i \leq v_i$$

Increasing property (monotonicity). A function f from E to E—or a local adjustment—will be said to be increasing if the following relation holds:

$$\forall (u,\, v) \in E \times E, \quad u \preceq v \Rightarrow f(u) \preceq f(v)$$

This monotonicity characteristic is crucial to reach a unique fix-point when several adjustment procedures are involved. For more details, see Tarski (1955) and Péridy and Rivreau (2005). There is also a second, more interesting, outcome to monotonicity due to the fact that adjustments of release dates can only occurs at specific point of the planning horizon: clearly, with this property you can "jump" from two consecutive critical time breakpoints without checking the in-between values. These points—the so-called *critical time breakpoints*—are defined more precisely in the next section and roughly correspond to the initial release dates and to the completion times of some specific sets. Finally, this increasing characteristic seems a priori to be a natural property: finding less information from a more constrained problem is a little bit counter-intuitive. However, if the great majority of local adjustments are indeed monotonic, it should be noted that some of them—for instance Fix Triple Arcs (Brucker *et al.*, 1994)—are non-increasing.

Non-anticipative property. A local adjustment f is said to be non-anticipative if the final adjustment value α_i of any release date is independent of the final adjustment values of release dates of operations such that $\alpha_j \geq \alpha_i$.

This second core property means that the final adjustment α_i of initial release date r_i is only a function of all processing times, all deadlines and of final adjustments values α_j of operations such that $\alpha_j < \alpha_i$. This characteristic allows in particular a chronological study of critical time breakpoints: at each time breakpoint we can check if a given release date reaches its final adjustment value or not. Moreover, since this value does not rely on future adjusted release dates, the overall procedure can be proved to be stable ($f \circ f = f$).

Not-first, immediate selections and ascendant set adjustments have been in particular shown to satisfy these properties (Péridy and Rivreau, 2005). In this paper, this framework is completed with the improved immediate selections.

2. IMPROVED IMMEDIATE SELECTIONS ADJUSTMENTS

2.1 Object

For the sake of clarity, the ascendant set-like adjustments for immediate selections will be precisely stated as follows:

Improved Immediate selections adjustments. Let $i \in \mathcal{O}$ and also $J = \{j \in \mathcal{O} \setminus \{i\} \mid r_i + p_i + p_j > d_j\}$. Operation i must be processed after all operations from J. Hence, we can let

$$r_i \leftarrow \max\left(r_i \; ; \; \max_{J' \subseteq J} \left\{ \min_{j \in J'} r_j + \sum_{j \in J'} p_j \right\}\right)$$

One can observe that these improved immediate selections adjustments are increasing and non-anticipative. Indeed, increasing the value of a release date can only add new selections and also result in an increase of the values of adjustments made. Hence, improved immediate selections are increasing. Moreover, once the adjustments are stabilised, we necessarily have for any operation i:

$$\alpha_i \leftarrow \max\left(r_i \; ; \; \max_{J' \subseteq J} \left\{ \min_{j \in J'} \alpha_j + \sum_{j \in J'} p_j \right\}\right)$$

Since all durations are positive, it follows that any final adjustment value α_i of a release date only relies on the final adjustment values α_j of operations such that $\alpha_j < \alpha_i$, and thus improved immediate selections are also non-anticipative.

As already mentioned, these properties correspond to the framework of Péridy and Rivreau (2005): therefore, we can use here the same technique which consists in a chronological study of potential adjustment dates (the critical time breakpoints).

The present contribution will mostly concern

- the *quality* (value) of adjustments performed;
- the *stability* of the algorithm;

- the $O(n \log n)$ *complexity* of this procedure.

However, we should add that there still remains an important open question: is it possible to design an effective and stable algorithm that is able to simultaneously perform adjustments of release and due dates? Indeed, like most adjustment procedures in the literature, when the adjustment of release dates is performed it is assumed that the due dates are fixed (and vice versa). Therefore, if we consider the whole process, which implies adjusting both release and due dates, any adjustment of a due date requires us to start again the adjustment procedure on release dates (and reciprocally). Finally, it appears that the overall stability is probably a difficult problem to handle if we consider the literature, which remains very discrete on that particular subject.

2.2 Notation and Basic Properties

In order to explain and justify our procedure, we need to introduce some auxiliary notation and exhibit some properties. In the following sections we will assume that operations are numbered in increasing $d_i - p_i$ order, i.e.

$$d_1 - p_1 \leq d_2 - p_2 \leq \cdots \leq d_n - p_n$$

Let us recall that our algorithm proceeds by a chronological examination of critical time breakpoints at which adjustments can occur. For each critical breakpoint t, some operations can either reach their final adjustment value, or be delayed (i.e. adjusted on a later date). We will denote by D the set of operations that are at least delayed up to t (those who satisfy $r_i < t \leq \alpha_i$) and by L the set of operations that are not available before t (with $t \leq r_i$). Please note that for operations from $L \cup D$ we will necessarily have $\alpha_i \geq t$ at the end of the algorithm, and that operations that do not belong to $L \cup D$ have been necessarily adjusted before t. For reasons of convenience, at a given critical time breakpoint t, the α_i-values of unfixed operations—those in $L \cup D$—are arbitrarily set to $+\infty$.

Now, let us consider a given subset of operations at time t. We denote

$$C(J) = \max_{J' \subset J} \left\{ \min_{j \in J'} \alpha_j + \sum_{j \in J'} p_j \right\}$$

By definition, if J contains one operation from $L \cup D$, then $C(J)$ is arbitrarily set to $+\infty$. We will also denote by K_l the following set:

$$K_l = \{k \in O \mid k \leq l\} = \{k \in O \mid d_k - p_k \leq d_l - p_l\}$$

We can now express the improved immediate selections with this notation. Let us assume that we are at a given critical time breakpoint t. We need to evaluate for operations in D and those in L with $r_j = t$, if they must be delayed or, on the contrary, if their final adjustment value α_j is equal to t. Let us denote by j a given operation from $D \cup \{k \in L \mid r_k = t\}$ and let operation i be defined as follows:

$$i = \min\{l \in \mathcal{O} \mid C(K_l) > t\}$$

Clearly, for all $k < i$, we have $k \notin L \cup D$ (otherwise, $C(K_k) = +\infty$, which is in contradiction with the definition of i). In other words, i is the only one operation from K_i that can be in $L \cup D$.

If $j \neq i$ then clearly j must be delayed if $t + p_j > d_i - p_i$. Indeed, in that case we have $t + p_j > d_k - p_k$ for all k in K_i. Hence, K_i is a valid set of predecessors for j. Since the completion time $C(K_i)$ of K_i is greater than t, then j should be delayed. On the other hand, if $t + p_j \leq d_i - p_i$, then any potential set K of predecessors is strictly included in K_i. By construction of K_i, we have necessarily $C(K) \leq t$: it follows that j cannot be delayed at time t, with respect to the improved immediate selections.

So, let us assume now that $j = i$. Clearly, operation i cannot be in the set of its potential predecessors. So we must remove i to this set and define i' as

$$i' = \min\{l \in \mathcal{O} \mid C(K_l \setminus \{i\}) > t\}$$

The same reasoning as used for j applies, and thus, we conclude that operation i must be delayed if and only if $t + p_i > d_{i'} - p_{i'}$. For $i \notin L \cup D$, we will arbitrarily set $i' = i + 1$, so in any case we have $i' > i$. Please note that if $L \cup D = \{i\}$—in other words, if $i = n$—then operation i cannot be delayed by any operation at time t (indeed, we have $C(\mathcal{O} \setminus \{i\}) \leq t$). In that case operation i' is not considered.

There is a strong relation between sets K_i and $K_{i'}$ that guarantees we avoid any removal of operation from these sets during the execution of the algorithm. Therefore, the sequence of sets K_i and $K_{i'}$ will always be increasing for the inclusion operator. This property is stated in the next proposition.

Proposition 1 *Let i and i' be defined as above for a given critical time period t. Then, we have*

$$C(K_{i'-1}) = C(K_i)$$

Proof. If $i \notin L \cup D$, the result is straightforward since $i' = i + 1$. Now, if $i \in L \cup D$, we have $C(K_{i'-1} \setminus \{i\}) \leq t$ by definition of i'. Since operation i belongs to $L \cup D$, we have $\alpha_i \geq t$. It follows that the value of $C(K_{i'-1})$ is given by the completion time of i, and $C(K_{i'-1}) = C(K_i)$. □

2.3 Example

Before describing the details of the algorithm, we will illustrate its operating mode and main characteristics through the following example.

	1	2	3	4
r_i	6	0	4	14
p_i	7	2	7	4
d_i	18	15	22	26
$d_i - p_i$	11	13	15	22

As already mentioned, we proceed by a chronological examination of critical time breakpoints that correspond to initial release dates and potential final adjustment values. For a given critical breakpoint t, operations i and i', and sets L and D are defined as in the previous section.

At every critical time, the next potential adjustment date for operations of $(L \cup D) \setminus \{i\}$ is given by $C(K_i)$. If the final adjustment value α_i has not yet been determined (if $i \in L \cup D$), it is also necessary to take into account its possible adjustment date $C(K_{i'} \setminus \{i\})$.

For brevity purposes, we start our presentation at time $t = 6$: final adjustment values α_2 and α_3 for operations 2 and 3 have been already determined to be equal to the initial release dates (since $r_2 + p_2 \leq d_1 - p_1$ and $r_3 + p_3 \leq d_1 - p_1$).

Operation i is equal to 1 and $C(K_i) = +\infty$ since operation i still belongs to L. The related operation i' is 3 because $C(K_2 \setminus \{1\}) = 2 \leq 6$. Since operation 3 has been adjusted, the exact $C(K_3 \setminus \{1\})$-value is known and is equal to 11.

Critical time breakpoint $t = 6$.

- Operation 1 becomes available: we have $t + p_1 \leq d_{i'} - p_{i'}$, then 1 is not delayed, and $\alpha_1 = 6$
- α_i is fixed: we can determine $C(K_i) = 13$. Operation i is adjusted, so $C(K_{i'} \setminus \{i\})$-value becomes useless: we set $C(K_{i'} \setminus \{i\}) = +\infty$
- The next critical time breakpoint is given by the minimum value over the release dates of operations from L and the $C(K_i)$-value: so $t = 13$.

Critical time breakpoint $t = 13$.

- $C(K_i) = t$: i and i' must be updated. From Proposition 1, we know that the next i-value is necessarily greater or equal than the current value of i'. So, we have $i \geq 3$. Moreover $C(K_3) = 18 > t$, it follows that $i = 3$. Once i updated, we need to reevaluate i'. Necessarily, i' is greater than the new i-value. Since operation 4 belongs to $L \cup D$, we deduce $i' = 4$ and $C(K_{i'} \setminus \{i\}) = +\infty$.
- The next critical time breakpoint is given by the minimum value over the release dates of operations from L and the $C(K_i)$- and $C(K_{i'} \setminus \{i\})$-values: so $t = 14$.

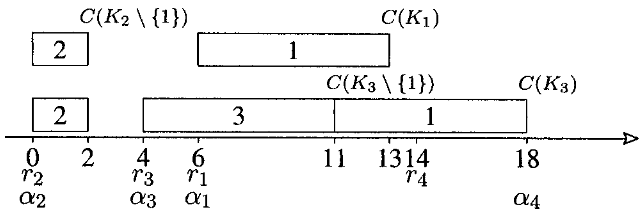

Figure 1. Example 1.

Critical time breakpoint $t = 14$.

- Operation 4 becomes available: $t + p_4 > d_i - p_i$, so 4 is delayed: $D \leftarrow D \cup \{4\}$
- The next critical time breakpoint is given by the minimum value over the release dates of operations from L and the $C(K_i)$-values: so $t = 18$.

Critical time breakpoint $t = 18$.

- $C(K_i) = t$: save operation $i' = 4$, no operation allows to increase $C(K_i)$-value, so the release date of operation 4 is definitively adjusted to $t = 18$.
- All operations are considered, the algorithm is completed.

3. IMPLEMENTATION

3.1 Notation and Basic Properties

In our algorithm, we reuse the main notation given in Section 2.2. For implementation reasons, two sets K and K' related to K_i and $K_{i'} \setminus \{i\}$ are introduced. The $C(K_i)$ and $C(K_{i'} \setminus \{i\})$ values are abbreviated in C and C'. Sets K and K' are implemented by means of red–black trees in order to be able to get in constant time the $C(K)$-value and to insert a new operation in O$(\log n)$. Please note that i and i' are inserted in K and K' only when final values α_i and $\alpha_{i'}$ are known.

As said before, the property described in Section 2.2 is the basis of the efficiency of our algorithm, since it guarantees that sets K and K' can be updated in an incremental fashion, without any removal of operation. Indeed, when K must be updated (that is when i needs to be incremented), we know that all operations between the current values of i and i' must be added to K, since we have $C(K_{i'-1}) = C(K_i)$. In more precise terms, if we note i_k, i_{k+1}, i'_k, i'_{k+1} the consecutive values of i and i' during the execution of the procedure, we have

$$i_k < i'_k \leq i_{k+1} < i'_{k+1}$$

Thus, this property enables us to gradually add operations in K and K' when necessary, that it is to say each time i or i' reaches its final adjustment value.

Finally, in the algorithm, it is implicitly assumed that there is a dummy operation $n+1 \notin \mathcal{O}$, with the following characteristics: $d_{n+1} = d_n, p_n = 0$.

3.2 Algorithm

The main algorithm is detailed below. At the beginning, all the operations are still to be considered and the first critical time breakpoint is the minimum release date (lines 1–2). In the main loop, we are given a current time breakpoint t.

For this critical time breakpoint, it is necessary to determine the relevant operations i and i', the related K and K' sets and C and C' values (procedure **update_i_i′**, line 4). With this information we can evaluate if the new available operation or previously delayed ones must be delayed or not (procedure **update_L_D**, line 5). After this step, all delayed operations at time t are in D.

If operation i reaches its final adjustment value at time t, it is inserted in red–black tree K and we deduce the exact value of $C(K_i)$ (line 7). Since operation i cannot be adjusted any more, this operation is also inserted in red–black tree K' (line 8). In the same way, if i' reaches its final adjustment value, it is added in K' and $C(K_{i'})$ is updated (line 10).

At last, the next critical time breakpoint to consider is updated, according to the fact that operation i can still be adjusted to C' (line 12) or not (line 13). The main loop is finished when all operations have been considered ($L \cup D = \varnothing$).

```
    procedure adjustments($r, \alpha$)
    {
1.      $L \leftarrow \mathcal{O}, \quad D \leftarrow \varnothing, \quad t \leftarrow \min_{j \in L} r_j$
2.      $i \leftarrow 1, K \leftarrow \varnothing, C \leftarrow +\infty, i' \leftarrow 2, K' \leftarrow \varnothing, C' \leftarrow +\infty$
3.      while ($L \cup D \neq \varnothing$) do
        {
4.          update_i_i′($t, L, D, i, K, C, i', K', C'$)
5.          update_L_D($t, L, D, i, i'$)
6.          if ($\alpha_i = t$)
                then
                {
7.                  $C \leftarrow$ insert($K, i$)
8.                  insert($K', i$)
                }
9.          if ($\alpha_{i'} = t$)
10.             then $C' \leftarrow$ insert($K', i'$)
11.         if ($i \in L \cup D$)
12.           then $t' \leftarrow \min\{\min_{j \in L} r_j\,;\ C\,;\ C'\}$
13.           else $t' \leftarrow \min\{\min_{j \in L} r_j\,;\ C\}$
        }
    }
```

Procedure **update_i_i'** is reproduced below.

First, we consider operation i and related set K: if $C = t$ then the set K cannot delay any operation after t. Therefore, it is necessary to increase i to add operations in this set (procedure **updateC**) until we get either a new operation i which has not yet reach its final adjustment value, or a definitively adjusted operation such that $C(K_i) > t$. In both cases, operations between $i' + 1$ and $i - 1$ are added in the set K'. If the new operation i is not yet adjusted (lines 3–6), we need to evaluate the new related i'. For that purpose, a call to **updateC** beginning at index $i + 1$ is made (lines 5–6). On the other hand, if i is already adjusted, then i' is not necessary for the current operation: in that case, we add i in K' for further computations, we set $i' = i + 1$ and insert i' in K' if $\alpha_{i'}$ is known (lines 7–10).

The same modus operandi is used to update operation i' (lines 11–12).

```
    procedure update_i_i'($t$, $L$, $D$, $i$, $K$, $C$, $i'$, $K'$, $C'$)
    {
1.     if ($C = t$)
          then
          {
2.           updateC($C$, $i$, $K$, $t$, $L$, $D$)
3.           if ($i \in L \cup D$)
                then
                {
4.                 forall $k \in [i'+1; i-1]$ do insert($K'$, $k$)
5.                 $i' \leftarrow i$
6.                 updateC($C'$, $i'$, $K'$, $t$, $L$, $D$)
                }
                else
                {
7.                 forall $k \in [i'+1; i]$ do insert($K'$, $k$)
8.                 $i' \leftarrow i + 1$
9.                 if ($i' \notin L \cup D$) and ($i' \neq n + 1$)
10.                   then insert($K'$, $i'$)
                }
          }
11.       if ($i \in L \cup D$) and ($C' = t$)
12.          then updateC($C'$, $i'$, $K'$, $t$, $L$, $D$)
}
```

The code of procedure **updateC** is basic: operations are inserted in the given set—in fact a red–black tree—in increasing order of $d_i - p_i$, until we get either an operation which is not yet adjusted or a set with a completion time strictly greater than t.

```
procedure updateC($Completion$, $index$, $Set$, $t$, $L$, $D$)
{
    do
    {
        $index \leftarrow index + 1$
        if ($index \in L \cup D$) or ($index = n + 1$)
            then $Completion \leftarrow +\infty$
            else $Completion \leftarrow$ insert($Set$, $index$)
    }
    while ($Completion \leq t$)
}
```

Finally, procedure **update_L_D** is also easy to state. Please note that sets L and D are implemented as heap data structures: function **top** returns—without removal—the operation with minimum release date for L and with minimum processing time for D.

In the first place, operations that have been delayed to t are considered (lines 1–8): all operation j from D that is not selected in respect to i—such that $t + p_j \leq d_i - p_i$—is removed from D, since it reaches its final adjustment value at t (lines 2–5). If i was previously delayed, we check if this operation is still selected in respect to i'. If it is not the case, operation i also reaches its final adjustment value (lines 6–8).

In the same way, operations from L that become available at time t may either be delayed t (line 13) or simply not adjusted (line 14).

```
    procedure update_L_D($t$, $L$, $D$, $i$, $i'$)
    {
1.      if ($D \neq \varnothing$) then $j \leftarrow$ top($D$)
2.      while ($D \neq \varnothing$) and ($t + p_j \leq d_i - p_i$)
        {
3.          remove($D$, $j$)
4.          $\alpha_j \leftarrow t$
5.          if ($D \neq \varnothing$) then $j \leftarrow$ top($D$)
        }
6.      if $(i \in D)$ and $(t + p_i \leq d_{i'} - p_{i'})$
            then
            {
7.              remove$(D, i)$
8.              $\alpha_i \leftarrow t$
            }
9.      if $(L \neq \varnothing)$ then $j \leftarrow$ top$(L)$
10.     while $(L \neq \varnothing)$ and $(r_j = t)$
        {
11.         remove$(L, j)$
12.         if $((j \neq i)$ and $(t + p_j \leq d_i - p_i))$ or $((j = i)$ and
          $(t + p_j \leq d_{i'} - p_{i'}))$
13.             then insert$(D, j)$
14.             else $\alpha_j \leftarrow r_j$
15.         if $(L \neq \varnothing)$ then $j \leftarrow$ top$(L)$
        }
    }
```

3.3 Proofs

Proposition 2 *Algorithm* ***adjustments*** *is a stable procedure that performs improved immediate selections adjustments.*

Proof. As mentioned in Section 2, improved immediate selections adjustments are monotonic and non-anticipative. This means that any increase of a release date value necessarily induces better adjustments (monotonicity) and that the final adjustment value of any adjustment is only based only previously adjustments made (non anticipation). These properties allow to focus on the chronological study of potential adjustment dates (which correspond to initial release dates, C-values for all operations except operation i and C'-value for operation i) without having to test the in-between values or to go back on earlier decisions. Since the C- and C'-values correspond to the makespan of sets K_i and $K_{i'}$ as defined in Section 2.2, we deduce that **adjustments** procedure is a stable procedure that performs improved immediate selections adjustments. □

Proposition 3 *Algorithm* ***adjustments*** *runs in $O(n \log n)$ time.*

Proof. As mentioned above, the critical time breakpoints correspond to the initial release dates and the potential adjustments dates C and C' that are given by the makespan of sets K and K'. These sets are implemented by mean of red–black trees. In the Appendix, it is shown that **insert** procedure runs in O$(\log n)$ time. These sets only strictly increase during the algorithm, so the overall complexity to insert at most n operations is O$(n \log n)$. In the same way, each operation is inserted and removed at most once in sets D and L.

Clearly, **insert** and **remove** procedures can be done in $O(\log n)$ by mean of a heap data structure. In consequence, the overall complexity for algorithm **adjustments** is $O(n \log n)$. □

4. COMPUTATIONAL EXPERIMENTS

To evaluate the efficiency of our procedure, we performed to kind of test for our procedure: *qualitative* tests to check if improved immediate selection can contribute to get more information, and *performance* tests to see if this procedure can compete with less sophisticated ones.

4.1 Qualitative Test

In table 1, we give some results on classical job-shop scheduling problems. LB1 and LB2 are obtained by bisection search on the value of the makespan until no infeasibility is derived. LB1 corresponds to classical immediate selections whereas LB2 is related to improved immediate selections. Lower bounds LB3 and LB4 are also obtained by bisection search, but global operations (Carlier and Pinson, 1994)—also called shaving (Martin and Shmoys, 1996)—are performed respectively on classical and improved immediate selections.

One can observe that improved immediate selections clearly outperform classical immediate selections in terms of pruning.

4.2 Performance Test

In order to see if the fact that our procedure is stable and in $O(n \log n)$ counterbalance its use of a time consuming red–black tree, we compare the relative performance of two algorithms that both perform improved immediate selection adjustments: the first one is the $O(n \log n)$ stable algorithm proposed in this paper and the second one is a basic non-stable $O(n^2)$ algorithm.

Table 2 reports a comparison between the CPU times obtained for lower bounds given by shaving on 10×10 job shop scheduling problems from the literature. The first column "Stable version" is given as the reference. Table 2 shows that on average the $O(n \log n)$ stable algorithm can favourably be compared with the non-stable one. It seems that the stable algorithm especially allows one to avoid large deviations on non-stable instances.

5. CONCLUSION

In this paper, we have introduced a stable algorithm that improves immediate selections adjustments. This led to a stable procedure that can adjust release dates with these local operations in a single pass in $O(n \log n)$. Computational results confirm that this new algorithm outperforms the classical one in adjustments with comparable CPU time. This algorithm can be introduced

Table 1. Lower bound on hard instances of job-shop scheduling problems

Instance	C^*	n	m	LB1	LB2	LB3	LB4
FT10	930	10	10	750	813	857	880
ABZ5	1234	10	10	975	1083	1138	1183
ABZ6	943	10	10	832	860	918	941
La16	945	10	10	810	817	890	924
La18	848	10	10	725	782	848	848
La19	842	10	10	729	751	823	825
La20	902	10	10	836	843	886	896
La21	1046	15	10	816	934	895	1004
La22	927	15	10	703	800	832	913
La24	935	15	10	787	857	860	905
La25	977	15	10	815	856	900	936
La29	1152	20	10	821	979	923	1114
La36	1268	15	15	1048	1138	1171	1234
La38	1196	15	15	1022	1065	1100	1125
La39	1233	15	15	1039	1137	1137	1221
La40	1222	15	15	1025	1115	1103	1192
ORB01	1059	10	10	792	834	919	971
ORB02	888	10	10	727	785	851	871
ORB03	1005	10	10	760	794	871	923
ORB04	1005	10	10	838	876	957	973
ORB05	887	10	10	695	746	807	840
ORB06	1010	10	10	807	838	913	951
ORB07	397	10	10	335	346	381	397
ORB08	899	10	10	676	690	773	894
ORB09	934	10	10	751	784	874	912
ORB10	944	10	10	777	855	887	930

in the framework given in (Péridy and Rivreau, 2005): indeed improved immediate selection adjustments are proved to be monotonic and non-anticipative. Since all precedence relations found by ascendant sets adjustments are selected by ascendant sets adjustments (from Theorem 1 in (Carlier and Pinson, 1990)), the use of ascendant sets combined with improved immediate selections allows us to perform ascendant set-like adjustments for *both* kinds of selections. Future work may include a domain splitting feature, in order to integrate a better support for Constraint Programming approaches, and an incremental feature.

Table 2. CPU comparison hard instances of Job-Shop Scheduling Problems.

Instance	$O(n \log n)$ Stable version	$O(n^2)$ Classical version
ORB1	100	90.3
ORB2	100	214.8
ORB3	100	90.3
ORB4	100	86.0
ORB5	100	250.0
ORB6	100	88.9
ORB8	100	93.8
ORB9	100	89.6
ORB10	100	106.7
MT1010	100	161.9
Average	100	127.2

APPENDIX. $C(K_i)$ COMPUTATION WITH A RED–BLACK TREE

At each step, we need to compute

$$C(K_i) = \max_{J' \subseteq K_i} \left\{ \min_{j \in J'} \alpha_j + \sum_{j \in J'} p_j \right\}$$

Let us define a red–black data structure to compute $C(K_i)$.

Let $\mathcal{T}$ be a red–black tree. In the tree, we denote for any node k associated with operation k:

- k_l, k_r, k_f, its left successor, its right successor and its predecessor in $\mathcal{T}$.
- $\mathcal{L}_k, \mathcal{R}_k$, its associated left and right subtrees (if $\mathcal{L}_k = \emptyset$ (resp. $\mathcal{R}_k = \emptyset$) then $k_r = 0$ (resp. $k_l = 0$)).
- $\mathcal{F}_k$, the subtree of root k.
- v, the root of $\mathcal{T}$.

$\mathcal{T}$ verifies the property

$$\forall k \leq n, \forall i \in \mathcal{L}_k, \forall j \in \mathcal{R}_k, \quad \alpha_i < \alpha_k \leq \alpha_j$$

Let us assign to any node k, the following quantities:

$$\sigma_k = p_k + \sum_{j \in \mathcal{R}_k} p_j \tag{A.1}$$

$$\zeta_k = \begin{cases} \max_{j \in \mathcal{F}_k} \left\{ \alpha_j + \sum_{i \in \mathcal{F}_k | \alpha_i \geq \alpha_j} p_i \right\} & \text{if } \mathcal{F}_k \neq \emptyset \\ -\infty & \text{otherwise} \end{cases} \tag{A.2}$$

From (A.1)–(A.2), we can deduce the following recursive definitions:

$$\sigma_k = p_k + \tau_{k_r} \tag{A.3}$$

$$\tau_k = p_k + \tau_{k_l} + \tau_{k_r} \tag{A.4}$$

$$\zeta_k = \max\{\sigma_k + \zeta_{k_l}\,;\ \alpha_k + \sigma_k\,;\ \zeta_{k_r}\} \tag{A.5}$$

with the convention that $\sigma_0 = 0$, $\tau_0 = 0$ and $\zeta_0 = 0$.
Indeed, we have

$$\zeta_{k_l} = \max_{j \in \mathcal{L}_k} \left\{ \alpha_j + \sum_{i \in \mathcal{L}_k | \alpha_i \geq \alpha_j} p_i \right\}$$

$$\zeta_{k_r} = \max_{j \in \mathcal{R}_k} \left\{ \alpha_j + \sum_{i \in \mathcal{R}_k | \alpha_i \geq \alpha_j} p_i \right\}$$

In particular, it is straightforward to check that

$$\zeta_v = \max_{j \in \mathcal{T}} \left\{ \alpha_j + \sum_{i \in \mathcal{O} | \alpha_i \geq \alpha_j} p_i \right\} = \max_{J' \subseteq K} \left\{ \min_{j \in J'} \alpha_j + \sum_{j \in J'} p_j \right\} = C(K) \tag{A.6}$$

In a red–black tree, the following property holds: "if a function f for a node x can be computed using only the information in nodes k, k_l and k_r, then we can maintain the values of f in all nodes of $\mathcal{T}$ during insertion and deletion without affecting the O$(\log n)$ performance of these operations." (See Cormen *et al.*, 1989.) The relations (A.3)–(A.5) verifying this property in the red–black tree $\mathcal{T}$, we can compute $C(K_i)$ in O$(\log n)$. The procedure **insert**(K, i) inserts operation i in the red–black tree K and returns the ζ-value of the root, $\zeta_v = C(K_i)$ according to relation (A.6).

References

Brinkkötter, W. and Brucker, P. (2001) Solving open benchmark instances for the job-shop problem by parallel head–tail adjustments. *Journal of Scheduling*, **4**:53–64.

Brucker, P., Jurisch, B. and Krämer, A. (1994) The job-shop problem and immediate selection. *Annals of Operations Research*, **50**:73–114.

Carlier, J. (1975) Thèse de 3e cycle, Paris VI.

Carlier, J. and Pinson, É. (1989) An algorithm for solving the job-shop problem. *Management Science*, **35**:165–176.

Carlier, J. and Pinson, É. (1990) A practical use of Jackson's preemptive schedule for solving the job-shop problem. *Annals of Operations Research*, **26**:269–287.

Carlier, J. and Pinson, É. (1994) Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, **78**:146–161.

Cormen, T., Leiserson, C. and Rivest, R. (1989) Introduction to Algorithms. MIT Press, Cambridge, MA.

Martin, P. and Shmoys, D. B. (1996) A new approach to computing optimal schedules for the job-shop scheduling problem, in: *Proceedings of the 5th International IPCO Conference*, pp. 389–403.

Péridy, L. and Rivreau, D. (2005) Local adjustments: a general algorithm. *European Journal of Operational Research*, **164**:24–38.

Tarski, A. (1955) A lattice-theoretical fixpoint theorem and its applications, *Pacific Journal of Mathematics*, **5**:285–309.