ECOLE NORMALE SUPERIEURE

Disjunctive Scheduling with Task Intervals

Yves CASEAU
François LABURTHE

LIENS - 95 - 25

Département de Mathématiques et Informatique

CNRS URA 1327

Disjunctive Scheduling with Task Intervals

Yves CASEAU
François LABURTHE

LIENS - 95 - 25

July 1995

Laboratoire d'Informatique de l'Ecole Normale Supérieure
45 rue d'Ulm 75230 PARIS Cedex 05

Tel : (33)(1) 44 32 00 00

Adresse électronique : caseau@dmi.ens.fr , laburthe@dmi.ens.fr

Disjunctive Scheduling with Task Intervals

Yves Caseau, François Laburthe
{caseau, laburthe}@dmi.ens.fr

LIENS Technical Report n° 95-25

Laboratoire d'Informatique de l'Ecole Normale Supérieure
Département de Mathématiques et d'Informatique
45 rue d'Ulm, 75230 Paris Cedex 05, FRANCE

Abstract

Task intervals were defined in [CL94] for disjunctive scheduling so that, in a scheduling problem, one could derive much information by focusing on some key subsets of tasks. The advantage of this approach was to shorten the size of search trees for branch&bound algorithms because more propagation was performed at each node.

In this paper, we refine the propagation scheme and use it not only inside a branch&bound algorithm but also within the framework of local moves and priority dispatching rules (greedy algorithm). All these techniques are integrated into a general disjunctive scheduling system which gives lower and upper bounds, finds a solution, refines it up to optimality and proves optimality.

This system is tested on the standard benchmarks from Muth & Thompson, Lawrence, Adams et al, Applegate & Cook and Nakado & Yamada (all available in the OR library). The achievements are the following :

- Window reduction by propagation : for 23 of the 40 problems of Lawrence, the proof of optimality is found with no search, by sole propagation; for typically hard 10×10 problems, the search tree has less than a thousand nodes; hard problems with up to 400 tasks can be solved to optimality and among these, the open problem LA21 is solved within a day.
- Lower bounds that are very quick to compute and outperform by far lower bounds given by cutting planes. The lower bound for the open problem YAM1 is improved from 812 to 826.
- A deterministic greedy heuristic which outperforms classical static selection rules.
- A local optimization algorithm in which several strategies using different neighborhood structures cooperate : it supports very efficient local moves involving very little search.

keywords: Jobshop scheduling, branch and bound, local moves, repair, heuristics, shuffle, propagation, constraints

Résumé

Les intervalles de tâches, définis précédemment dans [CL94] permettent pour les problèmes d'ordonnancement disjonctif de tirer beaucoup d'information de l'examen de certains ensembles critiques de tâches. L'avantage de cette approche est de réduire la taille des arbres de recherche dans des algorithmes de types branch&bound en propageant plus d'information à chaque noeud.
Dans ce rapport, nous raffinons les règles de propagation et les utilisons pour un algorithme de branch&bound, mais aussi dans le cadre de l'optimisation locale et de règles de priorité (algorithme gourmand). Toutes ces techniques sont intégrées dans un système d'ordonnancement disjonctif complet qui produit des bornes supérieures et inférieures, trouve une solution, la répare et la change jusqu'à arriver à l'optimalité, puis prouve l'optimalité de la solution. Ce système est testé sur les problèmes test standards de Muth & Thompson, Lawrence, Adams et al, Applegate & Cook et Nakado & Yamada (tous disponibles dans la "OR library"). Les succès sont les suivants:

- Réduction des fenêtres de temps par propagation : pour 23 des 40 problèmes de Lawrence, la preuve d'optimalité est donnée par simple propagation, sans recherche; pour des problèmes 10×10 réputés difficiles, l'arbre de recherche a moins de mille noeuds; des problèmes difficiles jusqu'à 400 tâches sont résolus et parmi eux le problème ouvert LA21 est résolu en un jour de calcul.
- Des bornes inférieures rapides à calculer et bien plus précises que celles issues des techniques de plans de coupes. La borne inférieure pour le problème ouvert YAM1 est améliorée de 812 à 826.
- Un algorithme gourmand plus performant que les règles de sélection classiques.
- Un algorithme d'optimisation locale dans lequel plusieurs stratégies basées sur différentes structures de voisinages coopèrent, ce qui permet des mouvements locaux efficaces n'impliquant que très peu de recherche.

mots clés: Ordonnancement d'atelier, branch and bound, optimisation locale, réparations, heuristiques, algorithmes gourmands, shuffle, propagation, contraintes

1. Introduction

Disjunctive scheduling problems are combinatorial problems defined as follows : a set of uninterruptible tasks with fixed durations that have to be performed on a set of machines. The problem is constrained by precedence relations between tasks. Moreover, the problem is said to be disjunctive because a resource can handle only one task at a time (as opposed to cumulative scheduling problems). The goal is to order the tasks on the different machines so as to minimize the total makespan of the schedule. These problems have been extensively studied in the past twenty years and many algorithmic approaches have been proposed, including branch & bound ([CP 89], [AC 91], [CP 94]), mixed integer programming with cutting planes ([AC 91]), simulated annealing ([VLA92]), tabu search ([Ta 89]), genetic algorithm ([NY 92], [Po 95]). In this paper, we describe a complete scheduling system which

- finds various lower bounds,
- finds quickly a good solution,
- improves the objective by repairing solutions and moving from a solution to a neighbor one,
- finds the optimal solution,
- proves optimality.

This system is a hybrid algorithm since the algorithmic paradigms are all different. Nevertheless, the common denominator to all these components is the use of the propagation engine.

The paper is organized as follows : Section 2 defines scheduling problems and explains how they can be modelled with time windows and solved with branch and bound, Section 3 explains the notion of task interval and exposes the reduction rules. Finally, Section 4 describes the full algorithm and reports tests on a large set of benchmarks.

2. Disjunctive Scheduling

2.1. Jobshop scheduling

A scheduling problem is defined by a set of tasks T and a set of resources R. Tasks are constrained by precedence relationships, which bind some tasks to wait for other ones to complete before they can start. Tasks that share a resource are not interruptible (non-preemptive scheduling) and mutually exclusive (disjunctive versus cumulative scheduling). The goal is to find a schedule that performs all tasks in the minimum amount of time.

Formally, to each task t, a non-negative duration $d(t)$ and a resource $use(t)$ are associated. For precedence relations, $precede(t_1, t_2)$ denotes that t_2 cannot be performed before t_1 is completed. The problem is then to find a set of starting times $\{time(t)\}$, that minimizes the total makespan of the schedule defined as $Makespan := \max\{time(t) + d(t)\}$ under the following constraints:

$$\forall t_1, t_2 \in T, \quad precede(t_1, t_2) \Rightarrow time(t_2) \geq time(t_1) + d(t_1)$$

$$\forall t_1, t_2 \in T, \quad use(t_1) = use(t_2) \Rightarrow time(t_2) \geq time(t_1) + d(t_1) \vee time(t_1) \geq time(t_2) + d(t_2)$$

Job-shop scheduling is a special case where the tasks are grouped into jobs $j^1, \ldots, j^n$. A job j^i is a sequence of tasks $j^i_1, \ldots, j^i_m$ that must be performed in this order, i.e., for all $\forall k \in \{1, \ldots, m-1\}$, one has $precede(j^i_k, j^i_{k+1})$. Such problems are called $n \times m$ problems, where n is the number of jobs and m the number of resources. The precedence network is thus very simple: it consists of n "chains". The simplification does not come from the matrix structure (one could always add empty tasks to a scheduling problem) but rather from the fact that precedence is a functional relation. It is also assumed that each task in a job needs a different machine. For a task j^i_k, the head will be defined as the sum of the durations of all its predecessors on its job and similarly the tail as the sum of the durations of all its successors on its job, e.g.:

$$head(j^i_k) = \sum_{l=1}^{k-1} d(j^i_l) \quad \text{and} \quad tail(j^i_k) = \sum_{l=k+1}^{m} d(j^i_l).$$

When one allows the precedence relations to form a more complex network, the problem is referred to as a disjunctive scheduling problem. Although general disjunctive scheduling problems are often more appropriate for modelling real-life situations, little work concerning them has been done (they have been studied more by the AI community than by Operations Researchers and most of the published work concerns small instances, like a famous bridge construction problem with 42 tasks [VH89])

The interest of n × m scheduling problems is the attention they have received in the last 30 years. The most famous instance is a 10 × 10 problem of Fisher & Thompson [MT63] that was left unsolved until 1989 when it was solved by Carlier & Pinson [CP89]. Classical benchmarks include problems randomly generated by Adams, Balas & Zawak in 1988 [ABZ88], Applegate & Cook in 1990 [AC91] and by Lawrence in 1984 [La84]. Of the 40 problems published by Lawrence, one is still unsolved (a 20 × 10 referred to as LA29). The typical size of these benchmarks ranges from 10 × 5 to 30 × 10.

2.2 The branch and bound scheme with time windows

Branch and bound algorithms have, however, undergone much study, and the method effectively used in [CP89] to solve MT10 is a branch & bound scheme called "edge-finding". Since a schedule is a set of orderings of tasks on the machines, a natural way to compute them step after step is to order a pair of tasks that share the same resource at each node of the search tree (which corresponds to getting rid of a disjunction in the constraint formulation). There are many variations depending on which pair to pick, how to exploit the disjunctive constraint before the pair is actually ordered, etc., but the general strategy is almost always to order pairs of tasks [AC91].

The domain associated with $time(t_i)$ is represented as an interval : to each task t_i, a window $\left[\underline{t_i}, \overline{t_i} - d(t_i)\right]$ is associated, where $\underline{t_i}$ is the minimal starting date and $\overline{t_i}$ is the maximal completion date (thus the starting date $time(t_i)$ must be between $\underline{t_i}$ and

$\bar{t_i} - d(t_i)$). During the search, a partial ordering (<<) of tasks is built, with the following meaning :

$t_1 << t_2 \Leftrightarrow time(t_1) + d(t_1) \leq time(t_2)$

In order to prune efficiently the search space, one needs to be able to propagate the decisions taken at each node of the search tree. Thus, whenever an ordering is selected, say $t_1 << t_2$, the bounds of the domains can be updated as follows: $\underline{t_2} \geq \underline{t_1} + d(t_1)$ and $\bar{t_1} \leq \bar{t_2} - d(t_2)$. With this model, inconsistency can be detected when one has $\bar{t} - \underline{t} < d(t)$ for some task t (t can no longer fit in its window).

3. Task Intervals and their Application to Scheduling

This part describes a redundant model, called *task intervals*, -introduced in [CL94]- that gives better insight about the feasibility of the scheduling problem than solely information from time windows. The idea is to focus not only on tasks but on sets of tasks sharing the same resource in order to reflect the disjunctive sharing constraints. This additional model has several interests: first, it allows us to propagate more information from an ordering decision between two tasks and so reduces the size of search trees, second, it detects inconsistencies early and third, it is particularly well-suited for a branching scheme using edge-finding.

3.1 Intervals as Sets of Tasks

A trick that is often used with algorithms that shrink domains is to add redundant constraints to improve pruning. The most obvious redundant constraint that is used by all constraint-based schedulers is the resource interval constraint. If we denote by $T(r)$ the set of tasks that use the resource r, $\underline{T(r)} = \min\{\underline{t}, t \in T(r)\}$ the earliest starting time of all tasks in $T(r)$ and $\overline{T(r)} = \max\{\bar{t}, t \in T(r)\}$ the latest completion time of all tasks in $T(r)$, the task constraint

$\bar{t} - \underline{t} \geq d(t)$

also applies to $T(r)$ with $d(T(r)) = \sum\{d(t), t \in T(r)\}$:

$$\overline{T(r)} - \underline{T(r)} \geq d(T(r)) \qquad (1)$$

If the time window $\overline{T(r)} - \underline{T(r)}$ is not sufficient for all tasks in $T(r)$, this additional constraint detects an inconsistency, whereas without this constraint, some ordering on r would have been necessary before an inconsistency for a $t \in T(r)$ could have been detected. The novel idea of task intervals is to apply this constraint (1) to all subsets of tasks that use a common resource. For an n × m problem, this generates m × (2^n -1) constraints. Fortunately, checking them on all subsets of tasks is equivalent to checking it only on task intervals, which are at most m × n^2.

Definition : If t_1 and t_2 are two tasks (possibly the same) satisfying :

use(t_1) = use(t_2) = m (the tasks share the same resource)

$\underline{t_1} \leq \underline{t_2}$ and $\bar{t_1} \leq \bar{t_2}$

then the *task interval* $[t_1, t_2]$ is the set of tasks t such that use(t)=m, $\underline{t_1} \leq \underline{t}$ and $\overline{t} \leq \overline{t_2}$. By convention, if t_1 and t_2 do not verify the order condition ($\underline{t_1} \leq \underline{t_2} \wedge \overline{t_1} \leq \overline{t_2}$), then $[t_1, t_2]$ will denote the empty set. Task intervals that are not empty are called *active.*

Note that to any set of tasks $S = \{t_1, ..., t_i\}$, we can associate t_p and t_q such that $\underline{t_p} = \underline{S}$ and $\overline{t_q} = \overline{S}$. Since $d([t_p, t_q]) \geq d(S)$ by construction, the equation (1) for $[t_p, t_q]$ subsumes that for S. On all following figures, the time will be represented on a horizontal axis, each task t on a horizontal line with two brackets to denote its time window[1] $[\underline{t}, \overline{t}]$ and a box of length $d(t)$ between both brackets. Figure 1 shows an example of a task interval $I = [t_1, t_1]$. The corresponding window is represented by larger brackets.

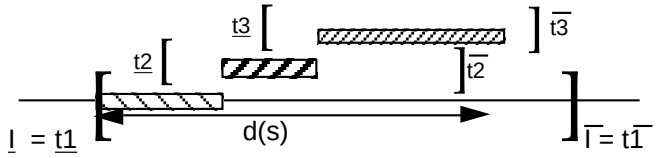

Figure 1: Task Intervals :
$I = [t_1, t_1]$ *represents the set* $\{t_1, t_2, t_3\}$

By construction, there are at most $m \times n^2$ task intervals to consider, which is only n times more than the number of tasks. It is possible to further reduce the number of task intervals that must be considered if we notice that the same time window may be covered by several pairs of tasks (when two tasks have a bound of their time windows in common). We can use a total ordering on tasks to select a unique task interval to represent each time window. However, the maintenance of such "critical" task intervals has shown to be too computationally expensive to gain any benefits from a reduced number of task intervals.

Implementation note:
Each task interval can be seen as the representation of a dynamic constraint. In addition to the two bounds t_1 and t_2 that do not change, we need to store for each interval $[t_1, t_2]$ its set extension (for propagation [Section 4.2] and maintenance [Section 4.3]). To see if an interval is active, we simply check if its extension is not empty. The set extension is a dynamic value that will change throughout the search and that needs to be backtracked. To avoid useless memory allocation, it is convenient to code it with a bit vector mechanism. Task intervals are stored in a matrix with cross-access, so that we have direct access to the set of intervals with t as a left bound, denoted $[t, \_]$ and to the set of intervals with t as a right bound, denoted $[\_, t]$.

In the rest of the paper, we shall call the *slack* of an interval I, written $\Delta(I)$, the value $\overline{I} - \underline{I} - d(I)$.

---

[1] To avoid confusion, we will reserve the word "interval" for task intervals and the expression "time window" for the laps of time during which the task may be performed.

3.2 Reduction with Intervals

In addition to the constraints from the equation (1), we use three sets of reduction rules, corresponding respectively to ordering, edge finding and exclusion.

Ordering rules use the precedence relation among tasks and a dynamic ordering relation $\ll$ that will be build during the search (cf. Section 4.1). The rules are as follows.

$$\forall t_1, t_2,\ \left(precede(t_1, t_2) \wedge \underline{t_2} < \underline{t_1} + d(t_1)\right) \Rightarrow \underline{t_2} := \underline{t_1} + d(t_1)$$

$$\forall t_1, t_2, \left(t_1 \ll t_2 \wedge \underline{t_2} < \underline{t_1} + d(t_1)\right) \Rightarrow \underline{t_2} := \underline{t_1} + d(t_1)$$

Symmetrical rules apply to upper bounds.

The second set of rules, **edge finding**, determines if a task can be the first or the last to be performed in a given task interval. For a task t belonging to a task interval S, the value of $\overline{S} - \underline{t} - d(S)$ is considered. If it is strictly negative, we know that t cannot be first in S, therefore t cannot start before the earliest ending time over all other tasks in S (cf. Figure 2). Thus the rule that we apply is the following :

$$\forall t, S, (t \in S \wedge \overline{S} - \underline{t} - d(S) < 0) \Rightarrow \underline{t} \geq \min\left\{\underline{t_i} + d(t_i), t_i \in S - \{t\}\right\}$$

Here also, a symmetrical rule applies to see if a task can or cannot be the last member of a given interval. Finding the first and last members of task intervals is known as “edge finding” [CP89] [AC91] and is a proven way to improve the search. We will complete these two rules in Section 4.1 with a search strategy that also focuses on edge finding.

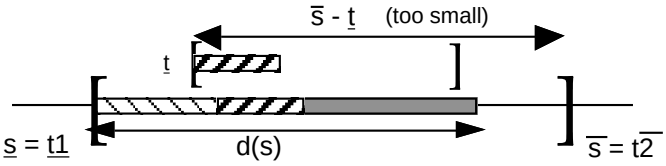

Figure 2: Edge Finding

The previous rule has one drawback, however, because it requires the computation of $\min\left\{\underline{t_i} + d(t_i), t_i \in S - \{t\}\right\}$, which is expensive. If we implement this rule with a rule-based language, this rule is likely to be evaluated for many tasks for which it will not increase $\underline{t}$. Since using a rule-based implementation has many other advantages (readability, maintainability and flexibility), we improve the rule by using $\underline{t_1} + d(t_1)$ (the left bound of the interval s) as an oracle for $\min\left\{\underline{t_i} + d(t_i),\ t_i \in set(S) - \{t\}\right\}$. The rule now becomes :

$$\forall t, S = [t_1, t_2],\ (t \in S \wedge \overline{S} - \underline{t} - d(S) < 0 \wedge \underline{t} < \underline{t_1} + d(t_1)) \Rightarrow \underline{t} \geq \min\left\{\underline{t_i} + d(t_i), t_i \in S - \{t\}\right\}$$

The last set of rules, **exclusion**, tries to order tasks and intervals. More precisely, we check to see if a task can be performed before an interval to which it does not belong (but which uses the same resource). This is done by computing the

value of $\overline{S} - \underline{t} - d(S) - d(t)$ (same as previously but t no longer belongs to S). If it is negative, then t cannot be performed before S, thus t must be performed after some tasks in S. In all cases, t must be performed after the first task in S, but in two special cases, it can be deduced that t must be performed after all tasks in S. Either because the interval S is too tight to allow t to be performed between two tasks of S, or because $\underline{t}$ is already greater than the latest start over all possible latest task in S. The functions packed? and is_after? respectively detect these two situations :

packed?(S,t) := $\left(\overline{S} - \underline{S} > d(S) + d(t)\right)$

is_after?(t,S) := $\left(\underline{t} + d(t) > \max\left\{\overline{t_i} - d(t_i), t_i \in S\right\}\right)$

Finally, we use the following rule (and its symmetrical counterpart):

$\forall t, S = [t_1, t_2],\ (t \notin S \wedge \overline{S} - \underline{t} - d(S) - d(t) < 0) \Rightarrow$

if packed?(S,t) or is_after?(t,S)

then $\underline{t} \geq \underline{S} + d(S) \wedge \forall t_i \in S, \overline{t_i} \leq \overline{t} - d(t)$

else $\underline{t} \geq \min\left\{\underline{t_i} + d(t_i), t_i \in S\right\}$

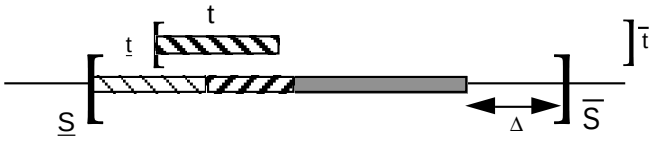

Figure 3: Exclusion - case packed
(packed?(S,t) = (d(t) > Δ))

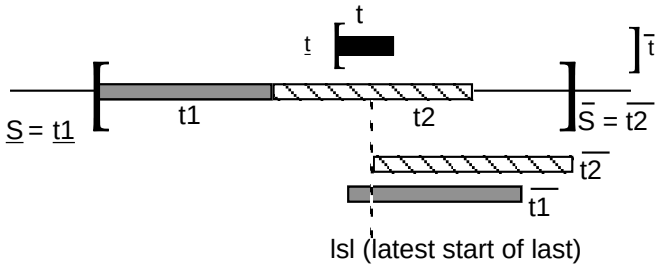

Figure 4 Exclusion - case where t is after S
(is_after?(S,t) = (t + d(t) > lsl))

Finer tuning of propagation

Some refinements can be made for the edge-finding and the exclusion rules (case unpacked). Indeed, more information can be propagated when we reach the conclusion that a task t is performed after some other task in a task interval S. When we come to this conclusion, the quantity $D = d(S \cup \{t\}) - (\overline{S} - \underline{t})$ is strictly positive. And so, there exists a subset V of tasks of S, which durations account for more than D, such that t is performed after V and before $S - V$. The candidates to be this subset V are $\left\{V \subseteq S \text{ such that } \underline{V} + d(S \cup \{t\}) \leq \overline{S}\right\}$ and S itself. If S is the

only candidate, $\underline{t}$ is increased to $\underline{S} + d(S)$, otherwise, $\underline{t}$ can be increased to a value x defined by :

$$x = \min\left\{\underline{V} + d(V) \quad \textit{for } V \subseteq S \textit{ such that } \underline{V} + d(S \cup \{t\}) \leq \overline{S}\right\}$$

Computing such a value x exactly is too long (it amounts to a knapsack problem). However, the first lower bound used (i.e. $\min\left\{\underline{t} + d(t) \textit{ for } t \in S\right\}$) is too gross an estimate of x . We refined it with the value computed by the following algorithm :

```
[after first(S:Task_Interval, t:Task, D:integer)  :  integer
 -> let  v := ∞  in
       (for t' in S -  {t}
          let d' := d(t'), v' := t' + d' in
             (if  (v' > v)  (if (d' ≥ D) v := v'
                            else D := D - d')),
       if (v = ∞) v := S + d(S),
       v)]
```

Note also that the function is_after? can be sharpened to take the dynamic ordering into account. Indeed the computation of the latest start of the last task in S can ignore tasks t' for which the ordering $t' \ll t$ has already selected (at a node of the search tree).

Triggering of these rules

If all these equations were checked each time some new information is drawn, the system would be terribly slow. So, deciding when to trigger the evaluation of these equations is a key point in the algorithm. There is fine tradeoff between too much triggering which brings redundant checks and is too slow, and not enough triggering which is faster but misses consequences that could have been drawn.

- Ordering rules for a task t are triggered upon changes of $\underline{t}$, $\bar{t}$, or $<<$. This corresponds to a classical dynamic reevaluation of the PERT.
- Concerning edge finding, the rule to decide whether t could be scheduled first in S is triggered upon changes of $\underline{t}$, $\overline{S}$ but also $\underline{S}$. Indeed, changes to $\underline{S}$ will not change the triggering condition of the rule, but may change its conclusion, by changing the value of the bound after first(S,t,D). The rule is not triggered upon changes to the set extension of S, because it prunes too little for its cost in time.
- The exclusion rule that decides whether a task t can be scheduled before all tasks of S or not is triggered upon changes to $\underline{t}$, $\overline{S}$ and to the set extension of S, but also upon changes to $\underline{S}$ (which might change the Boolean value of packed?(S,t))

For rapidity purposes, we also preferred to group the triggering upon time window bounds for the two edge finding and the two exclusion rules into a single one. It avoided redundant checks. The code also contains guards to avoid propagating the consequences of a change concerning a task t as soon as one of the consequences is to further reduce the window of t.

Note that unlike many problems solved with propagation rules, these rules may not be confluent. With the simplest version of the exclusion rule -case unpacked- (i.e. without the computations of after_first), these rules are not commutative (since applying them to a task interval I no longer subsumes it for all other subsets of I

[BL95]). Hence, it is not clear whether propagation always leads to the same fix point or not. It remains an open problem to find an appropriate set of reduction rules (subsuming this one) that can be proved to be confluent.

Implementation note

Finally, it must be mentioned that we have implemented these reduction rules with production rules in CLAIRE (following previous examples described in [CK92] and [CGL93]). A CLAIRE production rule contains two parts : an expression exp and a logical condition cond. The system formally differentiates cond w.r.t. the relations involved in cond and specified to be triggers and produces "demons" that watch over updates to these relations in order to evaluate exp when cond becomes true. In the case of scheduling, the conclusion exp always consist in narrowing time windows (via a functional call to increase and decrease, which are described below in section 3.3)

The declarative style of constraint based programming with production rules helped us in keeping the program elegant and small (400 lines without the heuristics). The control over the propagation and triggering mechanisms offered to the user is of great help for tuning the algorithm.

Unlike many problems solved with propagation rules, these rules may not be confluent. With the simplest version of the exclusion rule -case unpacked- (i.e. without the computations of after_first), these rules are not commutative (since applying them to a task interval I no longer subsumes it for all other subsets of I). Hence, it is not clear whether propagation leads to a fix point or not. It remains an open problem to find an appropriate set of reduction rules (subsuming this one) that can be proved to be confluent.

3.3 Interval Maintenance

Taking task intervals into account is a powerful technique that supports focusing very quickly on bottlenecks. However, the real issue is the incremental maintenance of intervals (indeed, recomputing all intervals upon each update on a time window would be much too slow). Resources are interdependent because of the precedence relationship (as displayed in Figure 5, where the arrows symbolize the precedence constraints).

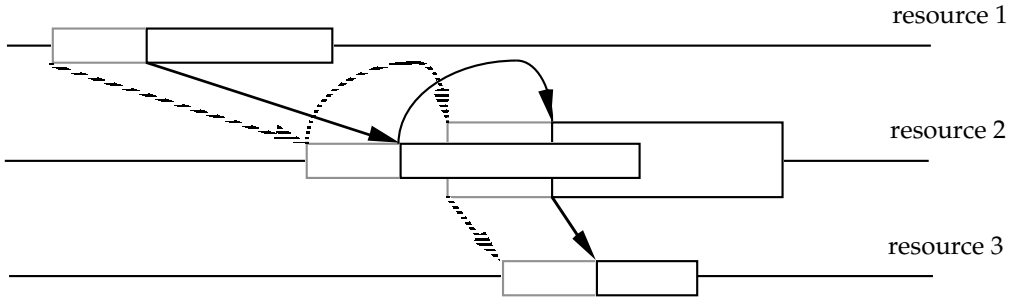

Figure 5 Resource Interdependence

While a resource is being scheduled, the changes to the windows of the tasks are propagated to other resources. We need to be able to compute the changes on task intervals very quickly (new active intervals, intervals that are no longer active and

changes to the set extensions of the intervals). More precisely, there are two types of events that need to be reacted to: the increase of a $\underline{t}$ and the decrease of a $\bar{t}$.

The correct algorithm for updating $\underline{t}$ can be derived from the definition of the set extension of a task interval:

$$[t_1, t_2] = \left\{t,\ \underline{t_1} \le \underline{t} \wedge \bar{t} \le \overline{t_2}\right\}$$

From this definition, we see that, when increasing the values of $\underline{t}$ from n to m,

- we must deactivate intervals $[t,\ t_2]$ if $m > \underline{t_2}$
- we must remove tasks t' from active intervals $[t,\ t_2]$ when $\underline{t'} < m$
- we must create new active intervals $[t_1,\ t]$ if $n < \underline{t_1} \le m$ and $\overline{t_1} < \bar{t}$
- we must add t to active intervals $[t_1,\ t_2]$ if $n < \underline{t_1} \le m$ and $\bar{t} < \overline{t_2}$

The interesting issues are the order in which we need to perform these operations, and the detection that we are in a state stable enough to propagate the changes and trigger the rules. It turns out that negative changes (removing tasks from set extension) do not need to be propagated, because all the rules we use always apply to subsets of intervals (i.e., if a rule can be applied to an interval, it could also be applied to any subset and would not yield more changes). Thus we perform the two "negative" actions first. Then we need to augment the other intervals, which requires triggering rules (as the window or the set extension of the interval changes). This requires that we have set $\underline{t}$ to its new value m (but not propagated this change yet because the intervals are not set up properly yet). After this, rules can legally be triggered, because one can be sure that all intervals have a set extension *that can only be smaller than what it should be*. Therefore, since all rules that are used are monotonic with respect to set extension, we know that any conclusion that might be drawn will be valid.

The last action is then to propagate the change to $\underline{t}$. To avoid duplicate work, we also need to check that $\underline{t}$ was not subsequently changed to a higher value by the propagation of a rule. Therefore, we are using two invariants H0 and H1 (cf. the following algorithm) to make sure that we stop all propagation work if m is no longer the new value for $\underline{t}$.

The algorithm that we use to increase $\underline{t}$ from n to the new value m is therefore as follows :

```
increase(t:Task,n:integer,m:integer) :                    ;; m > n
    for I = [t,t2]  in [t,_]
        if (n ≤ t2 < m) ∧ (t2 ≠ t)   set(I) := ∅          ;; I is no longer active
        else for t' in I
                if (n ≤ t' < m) ∧ (t̄' < t̄2)
                    ( I := I - {t'},    d(I) := d(I) - d(t') ),
    t = m,                                                ;; H0 ⇔ (t = m)
    for all t1 ≠ t such that use(t) = use(t1)
        if (n < t1 ≤ m)                                   ;; H1 ⇔ (t1 ≤ m)
            for I' = [t1, t2] in [t1, _]
                if (t̄ ≤ t̄2) ∧ H0 ∧ H1
                    (I' := I' ∪ {t'},   d(I') := d(I') + d(t') ),
    for I = [t1, t] in [_,t]
        if (t1 ≤ m) ∧ (t̄1 ≤ t̄) ∧ H0
            (I := {...}, d(I) := ... ),
```

if H0 propagate(t = m)

The algorithm for decreasing $\bar{t}$ is exactly symmetrical. We have tried two different variations of this algorithm. First, as mentioned earlier, we tried to restrict ourselves to "critical intervals", using a total ordering on tasks to eliminate task intervals that represented the same time window (and thus the same set). It turns out that the additional complexity does not pay off. Moreover, the maintenance algorithm is so complex that it becomes very hard to prove. The other idea that we tried is to only maintain the extension of task intervals (represented by a bit vector) and to use m pre-computed duration matrices of size 2^n representing the durations of all possible subsets of tasks. It turns out that the duration is used very heavily during the computation and that caching its value improves performance substantially; however, in order to limit the space complexity of the algorithm, we abandoned this feature for problems with more than 15 tasks per machine.

3.4 Comparison with related work

The structure of task intervals together with the reduction rules has two highlights : it is conceptually simple, but gives a very sharp insight of the tightness of the window situation. It provides an elegant unified frame for interpreting many former techniques used by Operational Researchers.

Operationally, the reduction rules presented here are very similar to those presented, in a different terminology, in [CP94]. Carlier and Pinson also do some window reduction, but call it adjusting the heads and tails of the tasks. There are however some real differences due to the fact that they do not maintain an extra structure such as the task intervals. As far as complexity is concerned, the procedure increase which is called every time that one of the bounds of a task has been changed is in $O(n^3)$, whereas theirs is in O(n log(n)), but their triggering is less efficient (since they do not reason about intervals, they have to consider more subsets after each modification to the window bounds of the tasks). As far as the expressiveness is concerned, Carlier and Pinson also include some lookahead in their propagation scheme (trying to replace a time window [a,b] by its left half or its right half and hoping to come to an impossibility in one of the cases). This operation amounts to a one-step breadth exploration of the search tree and explains the relatively smaller number of nodes for their search trees since each node encapsulates a fair amount of search.

The other contribution of task intervals is to give a unified framework that allows the expression complex reduction rules in a simple way. For example, all the sophisticated cutting planes described in [AC91] for semi-definite integer programming are subsumed by the three reduction rules.

3.5 Branch & bound

As we mentioned previously, a classical branching scheme for the job-shop is to order pairs of tasks that share the same resource [AC91]. The search algorithm, therefore, proceeds as follows. It picks a pair of tasks $\{t_1, t_2\}$ and a preferred ordering $t_1 \ll t_2$. The algorithm then explores sequentially the two branches ($t_1 \ll t_2$

and $t_2 \ll t_1$) recursively. The key point is the selection of the pair and of the preferred order. This algorithm produces a feasible schedule within the given makespan. When the makespan is not large enough, the algorithm explores the whole tree without finding any solution. The classical scheme with reduction rules is to iterate the algorithm many times with decreasing makespans to obtain an optimal solution, up to the point when the makespan is one unit too short and the algorithm comes to a dead-end in all branches of the tree, which proves optimality. The search strategy presented here is specially designed for proofs of optimality, therefore we only describe the selection of the pair to order since both branches are visited. This heuristic is well-suited for tight situations and therefore also works correctly to find an optimal (or near-optimal) solution. However, it performs very poorly for finding an initial feasible schedule.

The choice of the task pairs is directly inspired from the edge-finding method described in [AC91], which is itself inspired from the work of Carlier & Pinson [CP89]. This principle consists in considering the set of unscheduled tasks for a given resource, and picking a pair of tasks that could be both first (resp. last) in this set. The choice between first and last is based on cardinality. Our adaptation of this idea is to focus on the most constrained subset of tasks for the resource instead of the set of tasks that are currently unscheduled. This allows faster focusing on bottlenecks and takes advantage of the task intervals that are being carefully maintained.

For all task intervals I, let $\{t_1, \ldots, t_p\}$ be the set of tasks that could be scheduled first in I, let $\{t_1', \ldots, t_q'\}$ be the set of tasks that could be last and let $NC(I) := \min(p,q)$ be the number of choices associated to I. To each resource r, we associate the most critical task interval $Crit(r)$ as the one minimizing (over all tasks intervals using the considered resource) the quantity $\Delta(S) \times NC(S)$. Minimizing the slack forces to concentrate on bottlenecks and minimizing the number of choices insures that there will be much propagation. Indeed, the faster the first task of an interval is known, the faster the exclusion constraints between it and the other tasks in the interval can be propagated. Over all resources, we select the one (and the associated most critical task interval) that minimizes the quantity

$$ff(r) = \Delta(Crit(r)) \times \Delta(r) \times \min(par, NC(Crit(r))).$$

where *par* is a fixed parameter, empirically set around 3. This heuristic combines several criteria into a single numerical objective function :

- first, the slack of the most critical interval of the resource forces to concentrate on bottlenecks,
- second, the slack of r, which denotes the smallest slack over all intervals using r, forces the algorithm to schedule the tightest machines first (if most of the ordering has been done on one machine, it may be worth finishing it before considering other even tighter machines),
- the third quantity is taken into account to force the algorithm to take into account the "first-fail principle" (concentrating first on choices with the smallest number of possibilities), but only for choices offering less than *par* possibilities. As mentioned above, driving the search by first-fail makes sense only when few alternatives need to be considered, therefore *par* is given a relatively small value,
- the final quantity is the product of these three, rather than a linear combination. Indeed, scaling problems arise with linear combinations (linear

coefficients seem to be quiete sensitive to the size of the problem instance). Taking the product showed to be more robust.

Finally, once the task interval I has been selected, suppose that the set of possibly first tasks, $\{t_1, \ldots, t_p\}$ has been picked (this is the case when $p \leq q$); among this set, there remains to pick two tasks t_a and t_b such that both choices ($t_a \ll t_b$) and ($t_b \ll t_a$) will have the maximal impact (we try to reduce the entropy of the scheduling system, in a manner similar to what is described in [CGL93]). The ordering ($t_a \ll t_b$) is evaluated by predicting the consequent changes on the bounds of certain windows. If Δ is the slack of a window and Δ-δ is the slack after the ordering decision is taken, we want to minimize the resulting slack (Δ-δ) and to maximize the change δ. After many attempts, our best evaluation function (to be minimized) is the following:

$f(\Delta, \delta)$ = if (δ = 0) M else if ($\Delta < \delta$) 0 else $(\Delta - \delta)^2 / \Delta$,

where M is the current allowed makespan.

We evaluate the consequences of the ordering $t_a \ll t_b$:

$$t_a \ll t_b \Rightarrow \underline{t_b} := \max(\underline{t_b}, \underline{t_a} + d(t_a)) \quad \wedge \quad \overline{t_a} := \min(\overline{t_a}, \overline{t_b} - d(t_b))$$

$$\delta(t_b) = \max(0, \underline{t_a} + d(t_a) - d(t_b)) \quad \textit{and} \quad \delta(t_a) = \max(0, \overline{t_a} - (\overline{t_b} - d(t_b)))$$

We assess the impact of an ordering by its heaviest consequences :

$$g(t_a \ll t_b) = \min(f(\Delta(t_a), \delta(t_a)),\ f(\Delta(t_b), \delta(t_b)))$$

We always take $t_a = t_1$ (the left bound of the interval) and select t_b by minimizing the following function :

$$h(t_1, t_b) = \max\left(g(t_1 \ll t_b),\ \min\left(g(t_b \ll t_1),\ f(\Delta(S), \delta(S))\right)\right)$$

The function h is a maximum over both branches because one wants both possibilities to perform much propagation. Notice that the change to Δ(S) is taken into account for the branch ($t_b \ll t_1$). We can now summarize how to select the next pair of tasks that will be ordered.

next_pair()

find r such that ff(r) is minimal,

let S = Crit(r), S= $[t_1, t_2]$

S1 := $\left\{t \mid use(t) = r \wedge t \neq t_1 \wedge not(t_1 \ll t) \wedge \underline{t} \leq \underline{t_1} + \Delta(S)\right\}$;; could be first

S2 := $\left\{t \mid use(t) = r \wedge t \neq t_2 \wedge not(t \ll t_2) \wedge \overline{t} \geq \overline{t_2} - \Delta(S)\right\}$;; could be last

if |S1| ≤ |S2| $\quad \delta(S) := \min\left(\underline{t},\ t \in set(S) - \{t_1\}\right) - \underline{t_1}$

find t in S1 such that h(t_1, t) is minimal

return (t_1,t) if $g(t_1 \ll t) \leq g(t \ll t_1)$ and (t,t_1) otherwise

else $\quad \delta(S) := \overline{t_2} - \max\left(\overline{t},\ t \in set(S) - \{t_2\}\right)$

find t in S2 such that h(t, t_2) is minimal

return (t,t_2) if $g(t \ll t_2) \leq g(t_2 \ll t)$ and (t_2,t) otherwise

4. The complete scheduling system

Up to now, we have described a propagation mechanism that can be efficiently used, within a branch and bound scheme, to explore a solution space for a given makespan. However, for large scheduling problems, this exact approach takes too much time and the scheduler is expected to give solutions quickly (even if not optimal) and some guarantees about their quality (by means of lower bounds) within seconds.

Below, we describe a complete scheduling system that attacks the problem from these two angles : from above, it finds an approximate solution to start with, makes local changes and repairs on it to quickly decrease the upper bound and finally, when the upper bound is close to the optimal, performs an exhaustive search for decreasing makespans. From below, it can give good lower bounds (to estimate the distance of a solution to the optimal) and perform proofs of optimality.

An interesting fact is that the propagation mechanism takes part in this whole process, not only for the exhaustive search part of the algorithm.

4.1 An initial solution

The first problem is to find a feasible solution of reasonable cost. In an $m \times n$ problem, any schedule has a makespan less than n times than the optimal makespan. The "quality" of a solution should thus be judged with respect to n. The first thing to mention is that a search algorithm that associates time windows to tasks and tries to reduce them is not adapted to finding a solution. Indeed, such an algorithm needs to know an upper bound of the problem. Moreover, when an upper bound of very poor quality is given, the algorithm produces a solution of very poor quality also. This is due to the fact that ordering decisions are based on the analysis of the tightness of the situation: with too loose an upper bound, the relative slacks of the windows are not relevant and thus ordering decisions are almost taken randomly until the algorithm comes to a tighter situation (because of the previous poor choices) where it starts working properly. Moreover, the whole propagation machinery is too complicated when one just wants a starting solution, obtained without backtracking.

A classical method for obtaining starting solution is to use **priority dispatching rules** : the schedule is constructed chronologically, tasks are selected one after the other and performed as soon as possible. The algorithm works as follows : At each step, a set of "selectable" tasks is kept. In the beginning, this set is initialized to the set of all tasks that are first in a job (tasks that do not require any other task to be performed before them, i.e., tasks with an empty head). One of the tasks in this set is selected and scheduled as soon as possible on its machine. It is then removed from the set of selectable tasks and replaced in this set by its direct successor in the job. This process is repeated until no more tasks are to be selected. The whole algorithm depends on the selection rule. Lawrence reported some experiments in [La 84] of trying out 10 selection rules on 40 problems. The ten priority rules were the following : FIFO (the set of selectable tasks works as a queue), EST (select the task with earliest starting time), LST (select the task with latest starting time), EFT (select the task with earliest finish time), LFT (select the task with latest finish time), SPT (select the task with smallest processing time), LPT (select the task with longest processing time), MTR (most task remaining : select the task having the largest

number of tasks in its tail), MWKR (most work remaining : select the task having the longest tail) and RANDOM. His conclusions were that these methods lead to fair quality solutions, but that none of the criteria dominated the other ones. Indeed, eight of the criteria were best on at least one of the 40 problems in his test set (however, priority dispatching rules are most often used with the criterion SPT). The fact that these rules are static probably accounts for the poor quality of the solutions.

We tried out a more complex dynamic criterion, GREEDY. It gives good quality solutions and it is stable with respect to problem instance. In particular, it outperforms all criteria proposed in [La 84]. This selection rule is greedy in the sense that we consider at each step a lower bound estimate of the solution to be built, which is, in the end, equal to the value of the constructed solution. We then pick the task that will cause this lower bound to increase as little as possible. The lower bound LB is computed as follows :

$$LB := \max\left(lb(t) \mid t \in Task\right)$$

$$\text{where} \quad lb(t) := \underline{t} + \sum_{\substack{use(t') = use(t) \\ \underline{t'} \geq \underline{t}}} d(t')$$

Here, $\underline{t}$ denotes the earliest starting time of t. Indeed, throughout the task selection process, the propagation of the precedence constraints is left active (since only left bounds of the windows are known, this corresponds to dynamically reevaluating the PERT). If the task t_0 is selected next on machine m_0, then for all tasks t using m_0, $\underline{t}$ will be updated to $\max\left(\underline{t},\ \underline{t_0} + d(t_0)\right)$.

To evaluate the change to LB if the task t_0 is selected, we perform the update to all tasks sharing m_0, propagate precedences, evaluate the new lower bound and backtrack one step to undo the changes. The propagation of precedences as well as the evaluation of the lower bound LB take a time $O(n^2m)$, and this bound has to be evaluated $O(n^2m)$ times. Hence, the algorithm runs in $O(n^4m^2)$, whereas static selection rules such as SPT or MWKR run in $O(n^2m)$. The idea of a greedy selection rule was first mentioned in [DT93], and the performances are comparable. There are however a few differences : Dell'Amico and Trubian use a randomized version of a bi-directional algorithm (two semi-schedules are constructed -one from the left and one from the right), while we propose a deterministic unidirectional algorithm with a slightly finer lower bound estimate.

It could seem a good idea to reduce the complexity by computing approximates of the bound (taking the max of lb(t) not over all unselected tasks but only on those with earliest starting time) or by performing the propagation only on m_0. However, restricting the propagation leads to a substantial loss in the quality of the solution obtained and, for the size of the test instances, the time required for obtaining the first solution was very reasonable compared to the time required for the optimal solution or for the proof of optimality. However, for very large problems where optimality is out of reach, it is worthwhile (timewise) to take only estimates of the lower bound as in [DT93].

There are often ties with this rule, so we break them using a second criterion (instead of using randomization as in [DT93]) : we pick the task with the most work remaining (by maximizing tail(t) + d(t) over all concerned tasks t). This corresponds to minimizing the lexicographic pair (GREEDY, -MWKR). We tried many other criteria and this one worked best most of the time. The choice of this second

criterion is of importance : we relate in figure 6 the results with another classical criterion, SPT : the difference is important.

Another idea that we tried out was to make the schedule from both sides (this needs an adaptation of the bound lb(t), but we tried something very similar to what is described in [DT93]). Indeed tasks could be placed according to priority dispatching rules either at the beginning of the schedule or at the end of it. Our first impression was that when there was a situation with many ties in one view (say from left to right), it could be helpful to consider the problem from the other side and make a few choices until the situation on the original side was clearer. However, we encountered two problems : First, there is a natural tendency to drift to one of the sides : when more tasks have been selected at one end of the schedule than to the other, the lower bounds become more accurate and fewer ties arise: so it is necessary to force the system to consider to work on both sides to prevent it from drifting. Second, even if the lower bound remains low during most of the procedure, it rises up considerably in the end when the two partial solutions meet. Because of resource interdependence, it is hard to put back together these pieces of schedules built from the left and from the right, although both pieces, alone, are very well scheduled. Dell'Amico and Trubian also report considerable worsening in the quality of the solution at the end of a unidirectional algorithm, which we did not experience. For all these reasons, we did not find it worthwhile to have a bi-directional algorithm.

	size	optimal	SPT	GREEDY, SPT	GREEDY, -MWKR	BI-DIR [DT93]
MT 06	6×6	55	84	59	55	56
MT 20	20×5	1165	1399	1269	1275	1310
MT 10	10×10	930	1124	1103	1013	1076
ABZ5	10×10	1234	1416	1390	1330	1359
ABZ6	10×10	943	1130	1033	1052	1025
LA 21	15×10	1046	1560	1230	1211	1210
LA 28	20×10	1216	1610	1416	1354	1415
LA 31	30×10	1784	2278	2034	1883	1840
LA 36	15×15	1268	1828	1704	1443	1383

Figure 6: Performance of various selection rules

The conclusion about this part of the algorithm confirms the experiments of Dell'Amico and Trubian in the sense that priority dispatching rules can give systematically good quality solutions, but to the condition that they are dynamic rather than static (see the difference between SPT and GREEDY). Greedy algorithms are good candidates for such constructions and do not necessarily need to be bi-directional nor randomized to give good results. With these features, the incremental construction of a solution by selection rules can now be considered as a technology of choice for finding approximate solutions.

4.2 Local optimization

4.2.1. Repair

This section of the algorithm analyzes a solution and looks for easy improvements. In fact, in some situations, the solution can be very easily repaired: a small modification to the schedule, such as the permutation of two tasks on one machine can lower the value of the total makespan. The idea is to tighten a critical path. A path is a sequence of tasks $\{t_1, ..., t_n\}$ such that for all i in {1, .. , n-1} either t_i is performed just before t_{i+1} on the same machine or t_i and t_{i+1} are linked by the following precedence relationship $precede(t_i, t_{i+1})$. It is hence a sequence of tasks such that no two of them may overlap over time. Because of this mutual exclusion of tasks, one always has along a path $\sum_{i=1}^{n} d(t_i) \leq Makespan$. Moreover, a path is called critical when $\sum_{i=1}^{n} d(t_i) = Makespan$.

Figure 6 and 7 display the case of two tasks A2 and B2 that can be swapped and lead to such an improvement. Such a pair (A2,B2) is called “swappable” when the sequence (A2, B2, B3) is on some critical paths (say k), and when swapping A2 and B2 would allow us to shift B3 to the left by Δ (Δ > 0). Such a swap would then shrink the paths going through B3 by Δ, and thus remove k critical paths. So, when one finds a “swappable” pair (A2,B2) such that the task B3 is located on all critical paths, one knows that the swap will necessarily remove all critical paths and thus lead to an improvement of the global makespan.

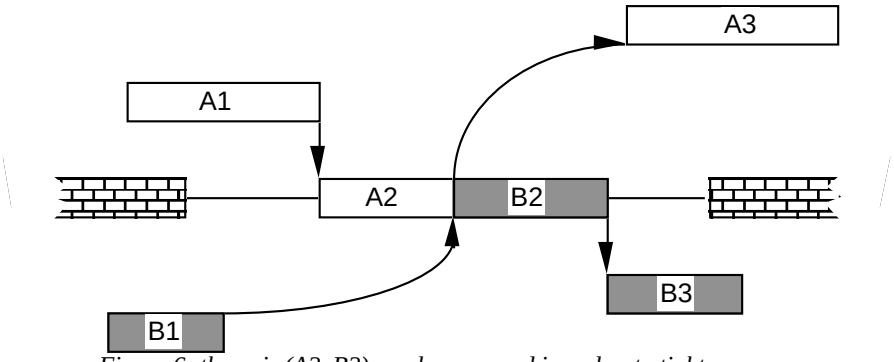

Figure 6: the pair (A2, B2) can be swapped in order to tighten the critical paths going through (A1, A2, B2, B3)

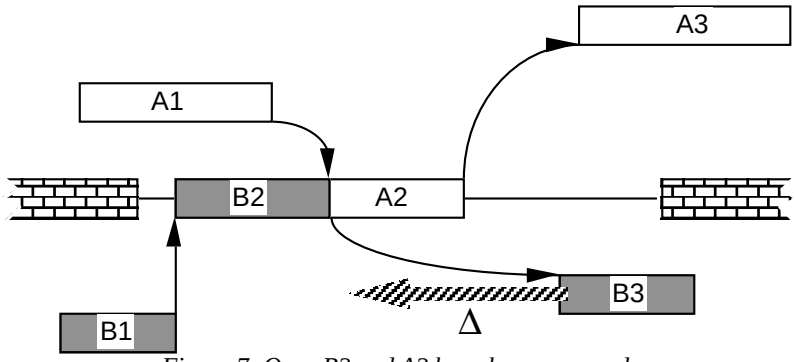

Figure 7: Once B2 and A2 have been swapped
B3 can be scheduled earlier

The repair algorithm goes as follows : A first pass reports all "swappable" pairs (t_i, t'_i) and for each pair the number nb_i of concerned critical paths and the minimum shrinkage Δ_i of these paths. The total number of critical paths is noted NB. If there are some pairs (t_i, t'_i) with $nb_i = NB$, we select among them the one with smallest Δ_i (selecting by smallest Δ_i turned out to work better than by largest Δ_i, because the number of such improving swaps that could be performed in a row was larger than in the other case; however, the difference was small). If no such pair exists, we select the pair with largest nb_i and among those, we take the one with maximum Δ_i. To summarize, the repair algorithm swaps the pair (t_i, t'_i) that maximizes the lexicographic pair

$\left(nb_i, \mathit{if}\ (nb_i = NB)\ \mathit{then}\ (-\Delta_i)\ \mathit{else}\ \Delta_i\right)$

The case $nb_i < NB$ does not improve the total makespan, neither does it necessarily reduce the number of critical paths (some critical paths are suppressed, but other may arise), so this phase may not necessarily lead to an improvement. It can even loop, swapping the same pair over and over again. To avoid this, we limited the number of repair steps that could be performed at a given makespan (we typically limited this number to 1 or 2 at the beginning of the search, when the solution is far from optimal and to 5 or 7 at the end of the search, when the solution is closer from the optimal). The repair algorithm could be further refined, using tabu lists for the swappable tasks, but it might not be worth the effort for the next stage, shuffle is very efficient anyway.

4.2.2. Shuffle

When the repair phase is over, either because the current solution has no more swappable pair, or because enough swaps have already been tried at the same makespan, another local optimization procedure takes over. The idea is similar to the shuffle procedure from [AC91], which is itself inspired from the shifting bottleneck procedure from [ABZ88]. The idea is to keep part of the solution and to recompute the rest. This allows to explore a neighborhood of the current solution. The shuffle procedure from [AC91] consists in freezing the schedule of one or several machines and to recompute the rest of the solution. Such a method is possible because the partial solution can be very quickly completed into a full one by the propagation rules and the edge finder. Actually, the better the propagation, the smaller the search for completing a partial solution and hence, the larger the neighborhood that can be explored. Our optimization method differs from the previous ones in several ways :

• A smaller fragment of the solution is kept. For example, in the shuffle schema, the schedule of just one or two machines are kept (for problems up to 20×20).

• The algorithm is forced to improve the current solution : the allowed makespan is lowered for the completion of the solution. To improve the rate of convergence, the total makespan is decreased by progressively by small steps (10 then 3 then 1).

• The information from the analysis of critical paths and swappable pairs is kept. When the analysis suggests that swapping a pair would eliminate some critical paths, the swap is performed, part of the solution around this pair is kept and the rest is recomputed.

• Only a very limited number of backtracks is allowed. This number starts at very small values (10 or 20 backtracks) and progressively increases

• Other neighborhood structures are explored.

The point is to avoid getting stuck when the search space is dense of solutions. To achieve this, we accept only solutions that can be found very fast. Whenever a better solution cannot be found with almost 0 backtracks, instead of persisting and spending much effort for finding one in this area, we change neighborhood and explore other areas of the tree.

The algorithm goes in several steps :

- SWAPS : when the repair phase is over and there are still some swappable pairs, the system tries to use them to guide the local moves : it selects a swappable pair that touches the most critical paths, swaps this pair, keeps the task order on the resource used by the pair, and tries to recompute the rest of the solution on the other resources (with the branch & bound algorithm described in section 3.5).

- BASIC SHUFFLE : the task order on one machine is kept and the rest of the solution is forgotten; the branch and bound algorithm tries to complete the rest of the schedule.

- VERTICAL SHUFFLE : Two dates t_1 and t_2 are selected in [0,Makespan] and the following part of the solution is kept: if two tasks (sharing the same resource) were both completly scheduled between t_1 and t_2 in the solution, their respective order is kept else it is forgotten (this amounts to keeping only the information in the schedule beween t_1 and t_2). The branch & bound algorithm then tries to complete this partial schedule into a full one.

- HORIZONTAL SHUFFLE : This step is an adaptation of the BASIC SHUFFLE : the task order is kept on a machine, except for the tasks that are on some critical paths (the idea is to keep track of the ordering decisions that did not matter and to allow more freedom in the ordering of the critical tasks)

The algorithm starts with a step of descent of 10 (the makespan is forced to diminish by 10), and an allowed number of backtracks of 10. When one solution that cannot be repaired is known, SWAPS is tried on all swappable pairs. If it does not succeed, basic shuffle is tried on all resources, starting from the one with largest slack, then vertical shuffle is tried at most 10 times, taking at random t_1 in $[0, Makespan]$ and t_2 - t_1 in $[0, Makespan/15]$. If this does not succeed, HORIZONTAL SHUFFLE is tried several times, keeping the respective order of all uncritical tasks on the two machines with largest slack (and for which HORIZONTAL SHUFFLE has not been tried so far). If all attempts to complete the schedule within the allowed number of backtracks failed, the step of descent is lowered (10 then 3 then 1), the number of allowed backtracks is augmented (multiplied by 3) and the

whole cycle of swaps and shuffles is retried. When the step of descent is already 1, a last pass is tried with an increased number of backtracks. If this fails, the algorithm shifts to exhaustive branch and bound.

In order to assess the efficiency of this local optimization procedure, we tested it against the problem set selected in [VAL94]. In 1994, Vaessens et al. compared many different approaches for finding approximate solutions to the job shop problem, including iterative improvement, shifting bottleneck heuristic, simulated annealing, taboo search, genetic algorithms and constraint satisfaction. These approaches were ranked on a set of 13 hard instances by the ratio between the distance of the solution to optimum and the optimum. (some of these problems were open at that time; in this case, the best lower bound is taken as reference instead of the optimum). Taboo search outperformed the other methods, the winner being the tailored taboo algorithm of Nowicki and Smutnicki [NS 93] with a ratio of 0,54% (after this come two other taboos ranking between 1 and 2% and all others candidates are above 2%. Our ratio is exactly the same as the one for Nowicki and Smutnicki (0,54% with the 1994 lower bounds, and 0,36% with the bounds known in 1995). Our procedure comes thus first ex-aequo in this ranking, ahead of the other taboos and far ahead of the simulated annealing and genetic algorithms tested in [VAL 94]. Another procedure based on constraint propagation [BLN95] was also tested on this set of 13 problems and achieved a ratio of 0,8%. Hence, in the case of job-shop scheduling, constraint propagation methods can compete with the best taboos algorithms. Note that some time figures in table 8 are high; they could be significantly lowered if the algorithm was stopped a few units earlier (say within 1% of the solution). Indeed, the closer one gets to optimum, the longer it takes to improve the current solution. For example, on LA29, if the algorithm is given less time, the best solution is 1180 (7000 s.), while if the algorithm is given more time (70 hours), the solution can be improved to 1161 (which brings the ratio below 0,36%).

	size	opt.	best solution	time
MT10	10×10	930	930	68 s.
LA02	10×5	655	655	6 s.
LA19	10×10	842	842	19 s.
LA21	15×10	1046	1046	420 s.
LA24	15×10	935	938	1 500 s.
LA25	15×10	977	977	394 s.
LA27	20×10	1235	1235	8 800 s.
LA29	20×10	1130-57	1168	20 000 s.
LA36	15×15	1268	1268	2 600 s.
LA37	15×15	1397	1397	740 s.
LA38	15×15	1196	1211	5 600 s.
LA39	15×15	1233	1233	578 s.
LA40	15×15	1222	1224	15 000 s.

Figure 8: The local optimization phase, evaluation on 13 hard problems

4.3 Branch and bound

4.3.1. Finding an optimal solution

We use branch and bound for two different purposes: for finding an optimal solution and for proofs of optimality. In both cases, branching is done by edge-finding, and the pair on which to branch is selected with an entropic function, as described in section 3.5. There are however a few shallow differences between these cases. Indeed, for proofs of optimality one needs to visit the whole search tree, whereas for finding an optimal solution one just needs to come to a leaf of the tree. Therefore, in the first case, the branching pair is selected to maximize the minimum entropic change over both branches (since both of them have to be explored), with the function h (cf paragraph 3.5) :

$$h(t_1, t_b) = \max\Big(g(t_1 \ll t_b),\ \min\big(g(t_b \ll t_1), f(\Delta(S), \delta(S))\big)\Big)$$

(recall that small values of g account for important changes, therefore the minimal change over both branches is a max), whereas in the second case, the function is modified as such

$$h'(t_1, t_b) = \min\Big(g(t_1 \ll t_b),\ \max\big(g(t_b \ll t_1), f(\Delta(S), \delta(S))\big)\Big)$$

The next tables contains the number of backtracks and running times for finding an optimal solution when the search is performed within the optimal makespan. We also tried another variation, adding a penalty is given to pairs of tasks which durations are too different (it is often interesting to postpone such ordering decisions). This feature brought susbtantial gains in the size of search trees, (the solution for MT10 went from 363backtracks to 95 backtracks), but was unstable (for ORB3, the figure rose from 6kb. to 11 kb), therefore, we did not keep it.

	backtracks	time
MT 10	363 b.	20 s.
ABZ 5	1 164 b.	61 s.
ABZ 6	67 b.	4,5 s.
La 19	1 008 b.	50 s.
La 20	267 b.	10,6 s.
ORB 1	1 539 b.	89 s.
ORB 2	54 b.	4,7 s.
ORB 3	6 690 b.	347 s.
ORB 4	1 156 b.	60 s.
ORB 5	658 b.	40 s.

Figure 9: Branch and bound for finding an optimal solution

Note that the function h' can also be used for proofs of optimality (it takes 2000 backtracks with h', versus 1575 with h for the proof of optimality of MT10).

4.3.2. Proofs of optimality

The next tables contains the results for proofs of optimality on the same ten problems. Our times are given on a Sparc 10, and those from [AC91] on a Sparc 1.

A good performance is achieved on MT10 with ten times less backtracks than [AC 91] and on La19, La20 and ORB2 which were supposed to be particularly hard 10×10 problems. In fact, performances are stable : 8 out of 10 problems are solved under 1600 backtracks. A few thousands of backtracks or less seems to be the typical measure for 10×10 problems

	[AC 91]	Task Intervals
MT 10	372 s. - 16 kb.	106 s. - 1575 b.
ABZ 5	951 s. - 58 kb.	85 s. - 1350 b.
ABZ 6	91 s. - 1,3 kb.	9,9 s. - 157 b.
La 19	1460 s. - 94 kb.	63 s. - 1109 b.
La 20	1402 s. - 82 kb.	52 s. - 901 b.
ORB 1	1482 s. - 72 kb.	550 s. - 7265 b.
ORB 2	2484 s. - 153 kb.	36 s. - 456 b.
ORB 3	2297 s. - 130 kb.	340 s. - 4323 b.
ORB 4	1013 s. - 44 kb.	82 s. - 1060 b.
ORB 5	526 s. - 23 kb.	61 s. - 799 b.

Figure 9: Proof of optimality for ten 10×10 problems

The table below reports experiments with other classical benchmarks. 8 more 10×10 problems confirm the previous conclusion. The interesting fact is that among the 40 problems published by Lawrence ([La 84]), 23 can be solved with no backtrack and no search. For a time window one unit smaller than the optimal, reduction rules come to a contradiction. Among these problems, large ones are solved (all five 30×10). This indicates that the large problems of Lawrence are not exceptionally hard, as are the 15×10 (such as La21) or 20×10 (such as La29), but are just large. On such normal instances, task intervals perform especially well to detect bottlenecks.

problem	size	time/backtracks		problem	size	time/backtracks
La1,2,3,5	10×5	0,1 s. - 0 b.		La31-35	30×10	1,7 s. - 0 b.
La4	10×5	0,6 s. 16 b.				
La6-10	15×5	0,2 s. - 0 b.		MT20	5×20	0,4 s. - 0 b.
La11-15	20×5	0,3 s. - 0 b.				
La16	10×10	3,7 s. - 54 b.		ORB 6	10×10	202 s. - 2770 b.
La17	10×10	0,4 s. - 5 b.		ORB 7	10×10	44 s. - 631 b.
La18	10×10	4,6 s. - 80 b.		ORB 8	10×10	0,2 s. - 4 b.
La23	15×10	0,6 s. - 0 b.		ORB 9	10×10	8,3 s. - 142 b.
La26,28,30	20×10	0,8 s. - 0 b		ORB 10	10×10	21 s. - 255 b.

Figure 10: Proof of optimality for other classical benchmarks

4.3.3. Optimal solution and proof of optimality within a single search tree

Finding an optimal solution and giving the proof of optimality can also be integrated within a single search tree. The proof of optimality is then performed "on the fly", by dynamically reducing the makespan after the first optimal solution has been found and continuing the exploration of the same search tree. For MT10, starting from 930 (the optimum), it takes a total of 6281 backtracks with h' (and even more with h). This figure may seem surprisingly high since part of the tree has already been discarded when the proof of optimality starts. But this part is located deep in the tree and therefore, the search for optimality is performed in a tree which top nodes have been selected for a larger (feasible) makespan. Since the entropic analysis is very sensitive, the pairs selected for a makespan of 930 are less relevant for 929 and therefore, the relevant alternatives are reconsidered many times, below the less relevant nodes inherited from the first part of the algorithm (930). The price to pay for the acuteness of the entropic analysis is the inability to reuse parts of a tree. When the value of the optimum is known beforehand, it is always worth restarting a search from scratch for the proof of optimality, rather than performing it in the same tree as the one for the optimal solution. However, when the optimal value is still unknown, such trees may be of interest. For example, when this algorithm is ran on MT10, starting with upper bound 940 (recall that the optimum is 930), it takes 779 backtracks to find a first solution at 938; this solution is successively improved to 937, 936, 935, 934 and 930, requiring between each solution another 30 to 80 backtracks. The system then takes 5492 backtracks for the proof of optimality. In order to avoid this dramatic increase of the number of backtracks for the proof of optimality, we decided to systematically stop the search and start it over when the number of backtracks became too large compared to the number of backtracks necessary for finding solutions (the rule was the following : if n_1 is the current average number of backtracks needed for finding a solution, the search is stopped when more than $5n_1$ backtracks have been spent without finding any solution and a search is restarted with the heuristic h (designed for proofs of optimality). In the case of MT10, it allowed to spend 2000 backtracks in the search for solutions from 938 down to 930 and 1569 backtracks for the proof of optimality.

4.4 Lower bounds

The next experiment was to look for lower bounds. We used two techniques that we compared with the methods described in [AC91]. Several methods are presented in [AC91] : the first one "Preempt" gives instantaneously a lower bound by constructing the optimum of the relaxed preemptive scheduling problem (for details, see [Ca 82]) and the other ones ("Cuts 1", "Cuts 2" and "Cuts 3") are cutting plane heuristics with increasingly complex cuts. Our first bound (reported in the column "Task Intervals") is to determine what maximum allowed total time window would cause a contradiction when the reduction rules are applied. The second method is also a common technique, that consists in computing the minimal schedule of one machine, leaving the propagation rules active. That is to say that only one machine is scheduled, but decisions made on this machine are propagated to the other tasks, where contradictions can be raised. This second lower bound (reported in the column "Task Intervals - 1 machine") is more expensive to obtain since some search is involved. The running times are measured on an IBM 3081D for those given by [AC91] and on a Sun Sparc 10 for ours.

	Preempt [AC 91]	Cuts 1 [AC 91]	Cuts 2 [AC 91]	Cuts 3 [AC 91]	Task Intervals	Task Intervals one machine
MT 10 opt = 930	808 (0,1 s.)	823 (5,23 s.)	824 (305 s.)	827 (7552 s.)	868 (0,3 s.)	915 (25 s - 541b.)
ABZ 5 opt = 1234	1029 (0,1 s.)	1074 (5,61 s.)	1076 (611 s.)	1077 (4971 s.)	1127 (0,3 s.)	1208 (18 s. - 426 b.)
ABZ 6 opt = 943	835 (0,1 s.)	835 (4,87 s.)	837 (335 s.)	840 (5257 s.)	890 (0,3 s.)	936 (3,5 s. - 59 b.)
La 19 opt = 842	709 (0,1 s.)	709 (5,57 s.)	716 (917 s.)	not available	763 (0,3 s.)	813 (7,8 s. - 202 b.)
La 20 opt = 902	807 (0,1 s.)	807 (5,13 s.)	807 (806 s.)	not available	851 (0,3 s.)	874 (1,6 s. - 29 b.)
ORB1 opt = 1059	929 (0,1 s.)	930 (7,16 s.)	931 (358 s.)	not available	975 (0,3 s.)	1045 (100 s. - 2k b.)
ORB 2 opt = 888	766 (0,1 s.)	768 (10 s.)	769 (327 s.)	not available	815 (0,3 s.)	867 (9,3 s. - 189 b.)
ORB 3 opt = 1005	865 (0,1 s.)	869 (5,95 s.)	870 (449 s.)	not available	907 (0,3 s.)	971 (19 s. - 436 b.)
ORB 4 opt = 1005	833 (0,1 s.)	891 (5,58 s.)	895 (555 s.)	not available	898 (0,3 s.)	1004 (473 s. 10k b.)
ORB 5 opt = 887	801 (0,1 s.)	801 (6,90 s.)	801 (323 s.)	not available	822 (0,3 s.)	873 (14 s. - 303 b.)

Figure 11: Lower Bounds for ten 10×10 problems

A few remarks need to be made. The Preemptive bound is at an average distance of more than 13% to the optimum, bounds obtained with cutting planes are at an average distance of around 11%, and our two bounds are respectively within 8% and 2% of the optimum. Taking running times into account, the first bound (almost instantaneous) should be related to the preemptive bound. The distance to optimum is almost divided by a factor 2 for the same running times. The second bound is also obtained within reasonable times (less than 25 seconds for 8/10 problems) and is extremely precise. Sophisticated cutting planes are not competitive in terms of quality of the bound (by a factor 5 or 6) and in terms of running times (becoming unreasonable).

These results clearly show the power of task intervals and their reduction rules as a cutting mechanism.

4.6 The complete procedure.

The complete procedure is decomposed as follows : an initial solution is found with priority dispatching rules, using the lexicographic criterion (GREEDY,-MWKR). The current solution is then improved by successive runs of repair, swap and shuffle (trying all different neighborhood structures described earlier). When local moves

no longer manage to improve the solution, a complete search is performed until the algorithm finds an optimal solution and proves its optimality. Below, two examples of famous instances illustrate the behaviour of the algorithm.

4.6.1. an example : MT10

MT10 is a very famous 10×10 job shop instance that remained unsolved for 25 years. It was first solved by Carlier & Pinson, using branch & bound and edge-finding. The shortest proof of optimality reported today is the one in [CP94], which take 35 nodes (but this figure does not account for some lookahead exploration) in a couple of minutes.

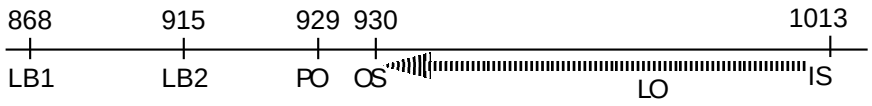

Figure 12 : the run on MT10

LB1	The first lower bound is obtained by simple propagation in 0,3 seconds.
LB2	The second lower bound obtained by trying to schedule only the tightest resource is obtained in 22 s. and 542 bk.
PO	The proof of optimality is obtained in 1575 backtracks and 80 seconds.
OS	The optimal solution is found in 97 backtracks and 5,8 s.
LO	The local optimization consists in 18 moves decomposed into 10 repairs, 1 swap and 7 shuffles. The total takes 70 s.
IS	The initial solution is found in 1 s.

The total procedure has thus taken 151 s.

4.6.2. an example : LA21

The problem LA21 a hard 15×10 instance published by Lawrence in 1984 (for which Applegate & Cook report 1040 as best lower bound and 1053 as best upper bound). This problem was solved early 95 for the first time by three separate teams, one using mixed integer programming and the other two (including ours) using constraint propagation. To our knowledge, our approach yields the best performance.

Figure 13: the run on LA21

LB1	The two lower bounds are obtained by simple propagation in 1 s.
PO	The proof of optimality is obtained in 1,28 million backtracks and 23 hours.
LO	The local optimization consists in 27 moves decomposed into 15 repairs, 2 swaps and 10 shuffles. The total takes 420 s.
IS	The initial solution is found in 10 s.

The total procedure has thus taken 24 hours

This algorithm also allowed us to improve the lower a bounds for two 20×20 open problems, YAM1 and YAM2, which respectively rose from 820 to 826 and from 860 to 861. A complete table with all figures is provided at the end of the paper. For a large set of benchmarks, the following entries are mentionned: the value of the optimum (or the best bounds in the case of open problems), the two lower bounds obtained by propagation and by scheduling one resource, the search trees for the proof of optimality and the optimal solution, the initial solution given by the greedy algorithm, the number of steps of the local optimization algorithm (rp=repairs, sw=swaps, sh=shuffle, vs=vertical shuffle, cs=critical horizontal shuffle) as well as the total time required by the local optimization algorithm and the best value obtained (column "solution"). The last column ("final solution") relates an other pass of local optimization, with a larger number of backtracks per shuffle (starting at 500 or 1000 and rising in the process up to 30000).

5. Conclusion

We have presented in this paper a complete disjunctive scheduling system which has been applied to many standard jobshop problems. The 15×10 problem LA21, unsolved since 1984, was solved for the first time in a day of computations. This scheduling system finds lower bounds, an initial greedy solution, performs local optimization and branch and bound search for finding an optimal solution and proving optimality.

The importance of propagation rules need to be stressed : they play a crucial part in the algorithm in two stages : from below, they allow to perform very efficient cuts for lower bounds and proofs of optimality and from above, they are used in the local optimization procedure. This technique of using propagation for performing local optimization seems very promising and is under invetigation on other combinatorial problems.

For the scheduling problems, a possible future direction of work would be to limit the notion of task intervals to smaller subsets of tasks in order to find good approximate solutions and lower bounds for larger problems (like 50×20).

Acknowledgments

This paper and the work on job-shop scheduling has been strongly influenced by numerous people. We are especially grateful to Bill Cook and Claude Lepape for their insights and their comments throughout our work. We also want to thank Fançois Fages, Clyde Monma, Jean-Francois Puget and Pascal Van Hentenryck for their kind encouragement and their valuable comments. Last, we are grateful to Jacques Carlier and Eric Pinson for an enlightning conversation about disjunctive scheduling.

References

[ABZ88] J.Adams, E. Balas & D. Zawak. *The Shifting Bottleneck Procedure for Job Shop Scheduling.* Management Science 34, p391-401. 1988

[AC91] D. Applegate & B. Cook. *A Computational Study of the Job Shop Scheduling Problem.* Operations Research Society of America vol 3, no 2, 1991

[Ba 69] E. Balas. *Machine Sequencing via Disjunctive Programming: an Implicit Enumeration Algorithm.* Operations Research 17, p 941-957. 1969

[Ba 95] P. Baptiste, rapport de DEA.

[BL 95] P. Baptiste, C. Lepape *A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling*. Proc. of the 14th IJCAI, 1995.

[BLN95] P. Baptiste, C. Lepape, W. Nuijten *Constraint-based Optimization and Approximation for Job Shop Scheduling* , Proc. of the IJCAI 95 orkshop on Intelligent Manufacturing Systems, Montréal, 1995.

[Ca 82] J. Carlier. *The one machine sequencing problem* European Journal of Operations Research 11, p. 42-47, 1982.

[Ca 91] Y. Caseau. *A Deductive Object-Oriented Language*. Annals of Mathematics and Artificial Intelligence, Special Issue on Deductive Databases, March 1991.

[CC88] J. Carlier & P. Chretienne. *Problèmes d'ordonnancement.* col. ERI, Masson, Paris 1988

[CGL93] Y. Caseau, P.-Y. Guillo & E. Levenez. *A Deductive and Object-Oriented Approach to a Complex Scheduling Problem.* Proc. of DOOD'93, Phoenix, December 1993.

[CK92] Y. Caseau & P. Koppstein. *A Cooperative-Architecture Expert System for Solving Large Time/Travel Assignment Problems.* International Conference on Databases and Expert Systems Applications, Valencia, Spain, September 1992.

[CL94] Y. Caseau & F. Laburthe. *Improved CLP Scheduling with Task Intervals.* Proc. of ICLP'94, ed: P. van Hentenryck, The MIT Press, 1994.

[CP89] J. Carlier & E. Pinson. *An Algorithm for Solving the Job Shop Problem.* Management science, vol 35, no 2, february 1989

[CP94] J. Carlier & E. Pinson. *Adjustments of heads and tails for the job-shop problem* , European Journal of Operations Research, vol 78, 1994, p. 146-161.

[DT93] M. Dell'Amico & M. Trubian. *Applying Tabu-Search to the Job-Shop Scheduling Problem.* Annals of Operations Research, vol 41, 1993, p. 231-252

[DW90] M. Dyer & L.A. Wolsey. *Formulating the Single Machine Sequencing Problem with Release Dates as a Mixed Integer Program.* Discrete Applied Mathematics 26, p255-270. 1990

[La84] S. Lawrence. *Resource Constrained Project Scheduling: an Experimental Investigation of Heuristic Scheduling Techniques.* GSIA, Carnegie Mellon University 1984

[MT63] J.F. Muth & G.L. Thompson *Industrial scheduling*. Prentice Hall, Englewood Cliffs, NJ, 1963

[NS 93] E. Nowicki & C. Smutnicki *A fast taboo search algorithm for the job-shop problem*, Preprint 8/93 Institute of Engineering Cybernetics, Technical University of Wroclaw, 1993.

[NY92] Nakado & Yamada *A Genetic Algorithm applicable to Large Scale Job-Shop Problems.* Parallel Problem solving from Nature 2, R Manner 1 B. Manderick eds., Elsevier Science, 1992

[Po88] M.C. Portmann. *Méthodes de Décomposition Spatiale et Temporelle en Ordonnancement de la Production.* RAIRO vol 22, no 5, 1988

[Po95] M.C. Portmann. Communication at FRANCORO 95, Mons, 1995

[Ta89] Taillard. *Parallel Taboo Search Technique for the Jobshop Scheduling Problem.* Internal Report ORPWP 89/11, Ecole Polytechnique Fédérale de Lausanne, 1989

[VLA92] P van Laarhoven, E.Aarts & J.K. Lenstra. *Job Shop Scheduling by Simulated Annealing.* Operations Research vol 40, no 1, 1992

[VAL94] R. Vaessens, E. Aarts & J.K. Lenstra *Job Shop Scheduling by Local Search.*

Benchmarks

	Opt.	LB1	LB2	Proof of optimality	Optimal solution	Greedy	Steps of local opt.					solution	Final solution
							rp	sw	sh	vs	cs		
MT 06 6 × 6	**55**	**55**			0 b.	**55**							
MT 10 10 × 10	**930**	868	915 541 b. 25 s.	1575 b. 80 s.	363 b. 18 s.	1013	10	1	7	0	0	**930** 68 s.	
MT 20 5 × 20	**1165**	**1165**				1275	12	6	6	1	0	**965** 156 s.	
LA 01 10 × 5	**666**	**666**			0 b.	698	3	1	1	0	0	**666** 1,2 s.	
LA 02 10 × 5	**655**	**655**			2 b.	699	2	0	1	2	0	**655** 6 s.	
LA 03 10 × 5	**597**	**597**			0 b.	683	7	1	4	2	0	**597** 6,5 s.	
LA 04 10 × 5	**590**	**590**			0 b.	687	4	2	4	1	0	**540** 4,9 s.	
LA 05 10 × 5	**593**	**593**			0 b.	595	0	0	1	0	0	**593** 0,4 s.	
LA 06 15 × 5	**926**	**926**			0 b.	**926**							
LA 07 15 × 5	**890**	**890**			2 b.	**890**							
LA 08 15 × 5	**863**	**863**			45 b. 3 s.	868	1					**863** 0,7 s.	
LA 09 15 × 5	**951**	**951**			0 b.	**951**							
LA 10 15 × 5	**958**	**958**			3 b.	962	1					**958** 0,8 s.	
LA 11 20 × 5	**1222**	**1222**			0 b.	1257	4	0	1	0	0	**1222** 13 s.	
LA 12 20 × 5	**1039**	**1039**			0 b.	**1039**							
LA 13 20 × 5	**1150**	**1150**			0 b.	**1150**							

	Opt	LB1	LB2	Proof of optimality	Optimal solution	Greedy	Steps of local opt.					solution	final solution
							rp	swsh		vs	cs		
LA 14 20×5	**1292**	**1292**			0 b.	**1292**							
LA 15 20×5	**1207**	**1207**			8831 b. 294 s.	1239	1	3	3	0	0	**1207** 41 s.	
LA 16 10×10	**945**	909	**945** 85 b. 1,8 s.	61 b. 2,4 s.	29 b. 2,5 s.	1034	3	0	3	2	0	**945** 30 s.	
LA 17 10×10	**784**	780	**784** 7 b.		5 b.	836	7	1	2	0	0	**784** 7,7 s.	
LA 18 10×10	**848**	803	835 34 b. 1 s.	102 b. 3,8 s.	37 b. 2,8 s.	913	8	1	2	0	0	**848** 8,7 s.	
LA 19 10×10	**842**	756	807 147 b. 2,5s.	1361 b. 48 s.	1008 b. 50 s.	966	7	0	6	0	0	**842** 19 s.	
LA 20 10×10	**902**	844	870 42 b. 1 s.	2120 b. 67 s.	267 b. 10,6 s.	970	5	1	6	0	0	**902** 60 s.	
LA 21 15×10	**1046**	1033	1033	1,2 Mb. 23 h.	2,2 Mb. 30 h.	1211	16	2	10	0	0	**1046** 420 s.	
LA 22 15×10	**927**	913	**927** 318 b. 23 s.	363 b. 43 s.		1189	30	8	12	0	0	**927** 360 s.	
LA 23 15×10	**1032**	**1032**				1124	12	5	4	0	0	**1032** 83 s.	
LA 24 15×10	**935**	892	906 26 b. 2,5 s.			1017	9	1	6	1	0	938 1500 s.	
LA 25 20×10	**977**	919	960 1172 b. 106 s.			1168	21	5	9 1	0 1	0	979 394 s.	**977** 2300s.
LA 26 20×10	**1218**	**1218**				**1218**							
LA 27 20×10	**1235**	**1235**				1466	37	2	19	4	1	**1235** 8854s.	
LA 28 20×10	**1216**	**1216**				1354	12	2	9	0	0	**1216** 435 s.	

	Opt	LB1	LB2	Proof of optimality	Optimal solution	Greedy	Steps of local opt.					solution	final solution
							rp	sw	sh	vs	cs		
LA 29 20 × 10	**1130** - **1157**	1119				1428	35 2	6 1	14 3	1 3	1 4	1180 7300s.	1161 70 h.
LA 30 20 × 10	**1355**	**1355**				1512	15	3	11	0	0	**1355** 700 s.	
LA 31 30 × 10	**1784**	**1784**				1883	17	6	5	0	0	**1784** 3000s.	
LA 32 30 × 10	**1850**	**1850**				1876	3	0	1	0	0	**1850** 357 s.	
LA 33 30 × 10	**1719**	**1719**				1775	8	1	3	0	0	**1719** 1085s.	
LA 34 30 × 10	**1721**	**1721**				1835	8	4	10	0	0	**1721** 3700s.	
LA 35 30 × 10	**1888**	**1888**				1954	13	5	3	0	0	**1888** 2400s.	
LA 36 15 × 15	**1268**	1233	1267 135 b.			1443	23	4	10	0	0	**1268** 431 s.	
LA 37 15 × 15	**1397**	**1397**				1614	18	5	15	0	0	**1397** 740 s.	
LA 38 15 × 15	**1196**	1106	1161 2470 b.			1380	15	1	13	1	0	1211 5600s.	
LA 39 15 × 15	**1233**	1221	1232 88 b.			1394	11	0	11	1	1	**1233** 578 s.	
LA 40 15 × 15	**1222**	1192	1203			1502	30	6	17	0	1	1229 5033s.	
ABZ 5 10 × 10	**1234**	1126	1190 219 b. 5 s.	1350 b. 61 s.	1164 b. 61 s.	1330	6	2	5	0 2	0	1238 272 s.	**1234** 472 s.
ABZ 6 10 × 10	**943**	889	933 62 b. 2 s.	217 b.	67 b. 4,5 s.	1052	8	6	7	1	0	**943** 48 s.	
ABZ 7 20 × 15	**655** - **665**	651	651	654 4172 b. 1360 s.		781						667	

	Opt	LB1	LB2	Proof of optimality	Optimal solution	Greedy	Steps of local opt.					solution	final solution
							rp	sw	sh	vs	cs		
ABZ 8 20 × 15	**638 - 670**	608				763						681	
ABZ 9 20 × 15	**656 - 668**	630				810	9	7	10	1	2	698 18 s.	
ORB 1 10 × 10	**1059**	975	1041 1791 b. 52 s.	982 b. 48 s.	1539 b. 89 s.	1168	15	0	9	1	0	**1059** 90 s.	
ORB 2 10 × 10	**888**	812	858 143 b. 3,7 s.	487 b. 23 s.	54 b. 4,7 s.	952	1	1	8	0	0	**888** 34 s.	
ORB 3 10 × 10	**998**	907	971 470 b. 13 s.	4641 b. 228 s.	6690 b. 347 s.	1199	17	1	11	0	0	1005 387 s.	
ORB 4 10 × 10	**1005**	898	998 9681 b. 258 s.	1215 b. 53 s.	1156 b. 60 s.	1158	12	2	8	0	0	**1005** 31 s.	
ORB 5 10 × 10	**887**	821	867 236 b. 6 s.	904 b. 43 s.	658 b. 40 s.	957	5	0	4	1	0	**887** 196 s.	
ORB 6 10 × 10	**1010**	946	983 281 b. 7 s.	4131 b. 162 s.	1336 b. 68 s.	1226	13	2	10	1	0	**1010** 125 s.	
ORB 7 10 × 10	**397**	364	382 141 b. 3 s.	838 b. 37 s.	75 b. 4,3 s.	463							
ORB 8 10 × 10	**899**	894	**899** 5 b.	5 b.	30 b. 19 s.	1020	14	3	8	2	0	**899** 89 s.	
ORB 9 10 × 10	**934**	909	**934** 182 b. 5,7 s.	164 b. 7 s.	28 b. 2,2 s.	995	6	1	3	1	0	**934** 16 s.	
ORB10 10 × 10	**944**	923	940 82 b. 3 s.	308 b. 14 s.	22 b. 1,8 s.	1117	16	6	8	0	0	**944** 32 s.	

	Opt	LB1	LB2	Proof of optimality	Optimal solution	Greedy	Steps of local opt.					solution	final solution
							rp	sw	sh	vs	cs		
YAM 1 20×20	**826** **- 888**	784		826 50 kb. 33 ks.									
YAM 2 20×20	**861** **- 912**	825		**861** 420 kb. 267 s.									
YAM 3 20×20	**827** **- 898**	799											
YAM 4 20×20	**918** **- 977**	885		916 454 kb. 258 ks.									