Heuristic Control of a Constraint-Based Algorithm for the Preemptive Job-Shop Scheduling Problem

CLAUDE LE PAPE [1] AND PHILIPPE BAPTISTE [1,2] clp@challenger.bouygues.fr, baptiste@utc.fr
[1]*Bouygues, Direction des Technologies Nouvelles, 1, av. E. Freyssinet, F-78061 Saint-Quentin-en-Yvelines.*
[2]*UMR CNRS 6599 HEUDIASYC, Université de Technologie de Compiègne.*

Abstract. In the recent years, constraint programming has been applied to a wide variety of academic and industrial non-preemptive scheduling problems, *i.e.*, problems in which activities cannot be interrupted. In comparison, preemptive scheduling problems have received almost no attention from both the Operations Research and the Artificial Intelligence community. Motivated by the needs of a specific application, we engaged in a study of the applicability of constraint programming techniques to preemptive scheduling problems. This paper presents the algorithms we developed and the results we obtained on the preemptive variant of the famous "job-shop scheduling problem." Ten heuristic search strategies, combined with two different constraint propagation techniques, are presented, and compared using two well-known series of job-shop scheduling instances from the literature. The best combination, which relies on "limited discrepancy search" and on "edge-finding" techniques, is shown to provide excellent solutions to the preemptive job-shop scheduling problem. A mean relative distance to the optimal solution of 0.32% is achieved in five minutes, on instances with 10 jobs and 10 machines (100 activities).

Keywords: Preemptive scheduling, job-shop scheduling, constraint programming, constraint propagation, resource constraints, timetables, edge-finding, limited discrepancy search, depth-bounded discrepancy search.

1. Introduction

Broadly speaking, constraint programming can be defined as a programming method based on three main principles:

- The problem to be solved is explicitly represented in terms of variables and constraints on these variables. In a constraint-based program, this explicit problem definition is clearly separated from the algorithm used to solve the problem. This separation guarantees that the problem to be solved is precisely defined. In many cases, it also simplifies the revision or the extension of a constraint programming application when the corresponding problem changes. For example, the replacement of old machines by new machines in a manufacturing shop can lead to the introduction of new constraints and to the removal of old constraints; yet, in some cases, the same problem-solving algorithms will continue to apply, with a different problem definition as input.
- Given a constraint-based definition of the problem to be solved and a set of decisions, themselves translated into constraints, a purely deductive process referred to as "constraint propagation" is used to propagate the consequences of the constraints. This process is applied each time a new decision is made, and is clearly separated from the decision-making algorithm *per se*. This allows the software

developer to implement the constraint propagation code and the decision-making code independently of one another. The same constraint propagation code can then be used to propagate decisions made by different decision-making algorithms, as well as decisions made by a human user. In an optimization context, this may lead to the development of several decision-making algorithms, dedicated for example to distinct combinations of optimization criteria.

- The overall constraint propagation process results from the combination of several local and incremental processes, each of which is associated with a particular constraint or a particular constraint class. This "locality" principle (Steele, 1980) is fundamental, since it enables the efficient combination of multiple constraint propagation techniques, associated with different classes of constraints. This allows the developer of a constraint programming application to reuse constraint propagation techniques developed for other applications. It also allows multiple developers to "share" libraries of constraints and augment such libraries with whatever new specific constraints are required for a given application.

In practice, the adherence to these three principles facilitates the development of complex problem-solving applications. However, the success of a constraint programming application depends a lot on the constraint propagation algorithms that are employed and on the heuristics that are used to explore the search space. Obviously, a "good" constraint propagation algorithm is an algorithm that makes a lot of useful deductions with a low CPU cost. In most cases, the "simplest" constraint propagation algorithms, which merely notice when values assigned to variables violate a constraint, are not acceptable as they are absolutely ineffective in guiding and reducing the exploration of the search space. On the other extreme, "complete" constraint propagation algorithms, which, given a constraint language, determine all the consequences of a given set of constraints, are in most cases even less acceptable, because determining all the consequences of a set of constraints (or even determining whether a set of constraints is consistent) is often an NP-hard problem, not known to be solvable in polynomial time. In general, a compromise must be made between the deductive power of the chosen constraint propagation algorithm and the cost of its application. Finally, constraint propagation is not the only component of a constraint programming algorithm. Heuristics are also used to decide how to explore the search space, *e.g.*, on which alternative decisions to branch first, whether all possible branches must be pursued and in which order, etc. In practice, the quality of a heuristic constraint-based algorithm must be judged by the quality of the solutions obtained after a given amount of CPU time, depending on the requirements of the problem-solving application under consideration.

A number of researchers have designed, implemented, and evaluated various constraint programming techniques for the resolution of non-preemptive scheduling problems, *i.e.*, problems in which activities cannot be interrupted. See, for example, (Fox, 1983), (Fox & Smith, 1984), (Rit, 1986), (Smith *et al.*, 1986), (Le Pape & Smith, 1987), (Collinot & Le Pape, 1987), (Burke, 1989), (Prosser, 1990), (Burke & Prosser, 1991), (Erschler *et al*., 1991), (Le Pape, 1991), (Lopez, 1991), (Beck, 1992), (Smith, 1992), (Aggoun & Beldiceanu, 1993), (Varnier *et al*., 1993), (Caseau & Laburthe, 1994), (Cheng & Smith, 1994), (Le Pape, 1994), (Nuijten & Aarts, 1994), (Nuijten, 1994), (Zweben & Fox, 1994), (Baptiste & Le Pape, 1995), (Caseau & Laburthe, 1995), (Cheng & Smith, 1995a), (Cheng & Smith, 1995b), (Caseau & Laburthe, 1996a), (Colombani, 1996), (Lock, 1996), (Nuijten & Aarts, 1996), (Baptiste & Le Pape, 1997), (Colombani, 1997). Many deductive algorithms that can serve as a basis for constraint propagation have also been developed by researchers in the Operations Research community, *e.g.*, (Erschler *et al*., 1976), (Pinson, 1988), (Carlier & Pinson, 1990), (Applegate & Cook, 1991), (Brucker *et al*., 1994), (Carlier & Pinson, 1994), (Brucker & Thiele, 1996), (Martin & Shmoys, 1996), (Péridy, 1996), (Brucker *et al*., 1997).

Coupled with branch and bound backtracking algorithms, these techniques proved to be very successful on both academic and industrial problems. By contrast, preemptive scheduling problems, *i.e.*, problems where activities can be interrupted, and "mixed" problems where some activities can be interrupted and some cannot, have received almost no attention from both the Operations Research and the Artificial Intelligence community (see, for example, (Demeulemeester, 1992) as one of a few exceptions). Motivated by the needs of a specific application, we decided to engage in a significant study of the applicability of constraint programming techniques to preemptive scheduling problems. In particular, we used the preemptive variant of the job-shop scheduling problem as a basis for comparing different constraint propagation and search algorithms. The present paper summarizes our results. It is organized as follows. Section 2 presents the preemptive job-shop scheduling problem and a basic "dominance property" for this problem. In Section 3, we show how this dominance property allows us to define a simple branching rule for the resolution of the preemptive job-shop scheduling problem and we embed this rule in a depth first search (DFS) algorithm. Section 4 discusses constraint propagation: extensions to the preemptive case (and to the mixed case) of two well-known constraint propagation techniques are presented and compared. In Section 5, we propose two additional means of exploiting the dominance property introduced in Section 2, and show that these extensions are, on average, worthwhile. In Section 6, we propose to replace the depth first search algorithm by a limited discrepancy search algorithm (LDS) (Harvey & Ginsberg, 1995). This, again, is shown to be worthwhile. Section 7 summarizes our results and suggests topics for future work.

2. The preemptive job-shop scheduling problem

The preemptive job-shop scheduling problem (PJSSP) can be defined as follows. Given are a set of jobs and a set of machines. Each job consists of a set of activities to be processed in a given order. Each activity is given an integer processing time (duration) and a machine on which it has to be processed. A machine can process at most one activity at a time. Activities may be interrupted at any time, an unlimited number of times. The problem is to find a schedule, *i.e.*, a set of integer execution times for each activity, that minimizes the makespan, *i.e.*, the time at which all activities are finished. (We remark, that since we require integer durations and integer execution times, the total number of interruptions of a given activity is bounded by its duration minus 1.)

In more formal constraint programming terms, a start time variable $start(A)$, an end time variable $end(A)$, an integer duration $duration(A)$ and a set variable $set(A)$ are associated with each activity A. The set variable $set(A)$ represents the set of times at which A executes, $start(A)$ represents the time at which A starts, $end(A)$ the time at which A ends, and $duration(A)$ the number of time units required for A. An additional variable, *makespan*, represents the makespan of the schedule. The following constraints apply:

- For every activity A, $duration(A) = |set(A)|$.
- For every activity A, $start(A) = \min_{t \in set(A)}(t)$ and $end(A) = \max_{t \in set(A)}(t + 1)$. This implies $end(A) \geq start(A) + duration(A)$. We remark that in a given solution we have $end(A) = start(A) + duration(A)$ if and only if A is not interrupted.
- For every job $J = (A_1, A_2, ..., A_m)$ and i in $\{1, 2, ..., m - 1\}$, $end(A_i) \leq start(A_{i+1})$.
- For every machine M, if $acts(M)$ denotes the set of activities to be processed on M, then, for every pair of activities (A, B) in $acts(M)$, $set(A)$ and $set(B)$ are disjoint.
- For every activity A, $0 \leq start(A)$.
- For every activity A, $end(A) \leq makespan$.

The goal is to minimize the value of the *makespan* variable. In constraint programming, such a minimization objective is generally satisfied by solving the decision variant of the problem (Garey & Johnson, 1979), with different bounds imposed on the makespan. At each iteration, an additional constraint $makespan \leq v$ (where v is a given integer) is imposed, and the problem consists of determining a value for each variable such that all the constraints, including $makespan \leq v$, are satisfied. If such a solution is found, its makespan can be used as a new upper bound for the optimal makespan. On the contrary, if it is proven (for example, by exhaustive search) that no such solution exists, $v + 1$ can be used as a new lower bound. The "decision variant" of the PJSSP, *i.e.*, the problem of determining whether there exists a solution with $makespan \leq v$, is NP-complete in the strong sense (Garey & Johnson, 1979).

The search space for the PJSSP is larger than the search space of the non-preemptive job-shop scheduling problem. Indeed, each $set(A)$ variable *a priori* accepts up to $(v * (v - 1) * ... * (v - duration(A) + 1)) / (1 * 2 * ... * duration(A))$ values. However, the dominance criterion introduced below allows the design of branching schemes which in a sense "order" the activities that require the same machine, and thus explore a reduced search space. The basic idea is that it does not make sense to let an activity A interrupt an

activity B by which it was previously interrupted. In addition, A shall not interrupt B if the successor of A (in its job) starts after the successor of B. The following definitions and theorem provide a formal characterization of the dominance property.

DEFINITION 1 *For any schedule S and any activity A, we define the "due date of A in S" $d_S(A)$ as:*

- *the makespan of S if A is the last activity of its job;*
- *the start time of the successor of A (in its job) otherwise.*

DEFINITION 2 *For any schedule S, an activity A_k has priority over an activity A_l in S ($A_k <_S A_l$) if and only if either $d_S(A_k) < d_S(A_l)$ or $d_S(A_k) = d_S(A_l)$ and $k \leq l$. Note that $<_S$ is a total order.*

THEOREM 1 *For any schedule S, there exists a schedule J(S) such that:*

1. ***J(S) meets the due dates**: $\forall A$, the end time of A in J(S) is at most $d_S(A)$.*
2. ***J(S) is "active"**: $\forall M$, $\forall t$, if some activity $A \in acts(M)$ is available at time t, M is not idle at time t (where "available" means that the predecessor of A is finished and A is not finished).*
3. ***J(S) follows the $<_S$ priority order**: $\forall M$, $\forall t$, $\forall A_k \in acts(M)$, $\forall A_l \in acts(M)$, $A_l \neq A_k$, if A_k executes at time t, either A_l is not available at time t or $A_k <_S A_l$.*

Proof: We construct $J(S)$ chronologically. At any time t and on any machine M, the available activity that is the smallest (according to the $<_S$ order) is scheduled. $J(S)$ satisfies properties 2 and 3 by construction. Let us suppose $J(S)$ does not satisfy property 1. Let A denote the smallest activity (according to $<_S$) such that the end time of A in $J(S)$ exceeds $d_S(A)$. We claim that:

- the schedule of A is not influenced by the activities A_k with $A <_S A_k$ (by construction);
- for every activity $A_k <_S A$, the time at which A_k becomes available in $J(S)$ does not exceed the time at which A_k starts in S (because the predecessor of A_k is smaller than A).

Let M be the machine on which A executes. The activities $A_k \in acts(M)$ such that $A_k <_S A$ are, in $J(S)$, scheduled in accordance with Jackson's rule (*cf.* (Carlier & Pinson, 1990)) applied to the due dates $d_S(A_k)$. Since $d_S(A)$ is not met, and since Jackson's rule is guaranteed to meet due dates whenever it is possible to do so, we deduce that it is **impossible** to schedule the activities $A_k \in acts(M)$ such that $A_k <_S A$ between their start times in S and their due dates in S. This leads to a contradiction, since in S these activities **are** scheduled between their start times and their due dates. □

We call $J(S)$ the "Jackson derivation" of S. Since the makespan of $J(S)$ does not exceed the makespan of S, at least one optimal schedule is the Jackson derivation of another schedule. Thus, in the search for an optimal schedule, we can impose the characteristics of a Jackson derivation to the schedule under construction. This results in a significant reduction of the size of the search space.

Example: Figure 1 displays a schedule S and its "Jackson derivation" $J(S)$.

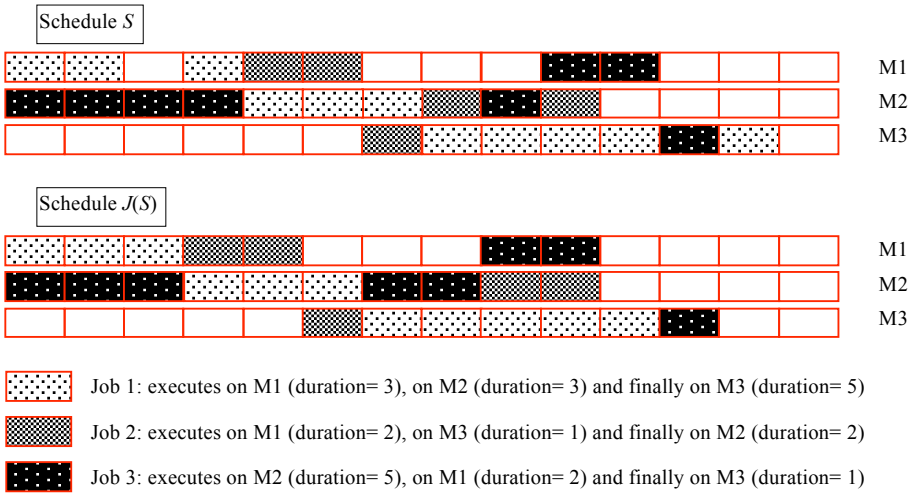


Figure 1. A preemptive schedule and its Jackson derivation

3. A branching scheme for the preemptive job-shop scheduling problem

The dominance criterion of the previous section led us to develop the following branching scheme (which heavily exploits the dominance criterion):

1. Let t be the earliest date such that there is an activity A available (and not scheduled yet!) at t.
2. Compute K, the set of activities available at t on the same machine than A.
3. Compute NDK, the set of activities which are not "dominated" in K (as explained below).
4. Select an activity A_k in NDK. Schedule A_k to execute at t. Propagate the decision and its consequences according to the dominance criterion (as explained below). Keep the other activities of NDK as alternatives to be tried upon backtracking.
5. Iterate until all the activities are scheduled or until all alternatives have been tried.

Needless to say, the power of this branching scheme highly depends on the rules that are used to (a) eliminate "dominated" activities in step 3 and (b) propagate "consequences" of the choice of A_k in step 4. The dominance criterion is exploited as follows:

- Whenever $A_k \in acts(M)$ is chosen to execute at time t, it is set to execute either up to its earliest possible end time or up to the earliest possible start time of another activity $A_l \in acts(M)$ which is not available at time t.
- Whenever $A_k \in K$ is chosen to execute at time t, any other activity $A_l \in K$ can be constrained not to execute between t and the end of A_k. At times $t' > t$, this reduces

the set of candidates for execution (A_l is "dominated" by A_k). In step 4, "redundant" constraints can also be added: $end(A_k) + rp_t(A_l) \leq end(A_l)$, where $rp_t(A_l)$ is the remaining processing time of A_l at time t; $end(A_k) \leq start(A_l)$ if A_l is not started at time t.

- Let $A_k \in acts(M)$ be the last activity of its job. Let $A_l \in acts(M)$ be another activity such that either $l < k$ or A_l is not the last activity of its job. Then, if A_l is available at time t, A_k is not candidate for execution at time t (A_k is dominated by A_l).

The above branching scheme defines a search tree which is, by default, explored in a depth-first fashion. Yet several "points of flexibility" remain in the resulting depth first search (DFS) algorithm: the constraint propagation algorithms used to propagate the decision to execute A_k at time t (as well as the resulting "redundant" constraints); the heuristic used to select activity A_k in *NDK*; and the course of action to follow when a solution with $makespan \leq v$ has been found. The first point, *i.e.*, constraint propagation, is discussed in Section 4. Section 5 discusses the two other points. Finally, Section 6 proposes an alternative to depth first search, *i.e.*, limited discrepancy search (LDS).

4. Constraint propagation

We distinguish three categories of constraints in the PJSSP. The first category includes all the constraints that relate the variables of a given activity A: $duration(A) = |set(A)|$, $start(A) = \min_{t \in set(A)}(t)$ and $end(A) = \max_{t \in set(A)}(t + 1)$. These constraints are easily propagated by maintaining a "lower bound" and an "upper bound" for the set variable $set(A)$. The lower bound of $set(A)$ is a series of disjoint intervals ILB_i such that each ILB_i is constrained to be included in $set(A)$. The upper bound is a series of disjoint intervals IUB_j such that $set(A)$ is constrained to be included in the union of the IUB_j. If the size of the lower bound (*i.e.*, the sum of the sizes of the ILB_i) becomes larger than $duration(A)$ or if the size of the upper bound (*i.e.*, the sum of the sizes of the IUB_j) becomes smaller than $duration(A)$, a contradiction is detected and a backtrack occurs. If the size of the lower bound (or of the upper bound) becomes equal to $duration(A)$, $set(A)$ receives the lower bound (respectively, the upper bound) as its final value. Minimal and maximal values of $start(A)$ and $end(A)$, *i.e.*, earliest and latest start and end times, are also maintained. Each of the following rules, considered independently one from another, is used to update the bounds of $set(A)$, $start(A)$ and $end(A)$.

- $\forall t, t < start(A) \Rightarrow t \notin set(A)$
- $\forall t, t \in set(A) \Rightarrow start(A) \leq t$
- $\forall t, end(A) \leq t \Rightarrow t \notin set(A)$
- $\forall t, t \in set(A) \Rightarrow t < end(A)$
- $\forall t, [\forall u < t, u \notin set(A)] \Rightarrow t \leq start(A)$
- $\forall t, [\forall u \geq t, u \notin set(A)] \Rightarrow end(A) \leq t$
- $start(A) \leq \max\{t \mid \exists S \subseteq set(A)$ such that $|S| = duration(A)$ and $\min(S) = t\}$
- $end(A) \geq \min\{t \mid \exists S \subseteq set(A)$ such that $|S| = duration(A)$ and $\max(S) = t - 1\}$

Needless to say, whenever any of these rules leads to a situation where the lower bound of a variable is not smaller than or equal to its upper bound, a contradiction is detected, and a backtrack immediately occurs.

The second category of constraints is composed of temporal constraints, of the form $var_i + d_{ij} \leq var_j$ where var_i and var_j are either start and end times of activities, or the *makespan* variable, or even the constant 0, and d_{ij} is an integer, *i.e.*, the minimal delay between the two time points var_i and var_j. These constraints, to which we add the redundant inequalities $start(A) + duration(A) \leq end(A)$, are easily propagated through an incremental version of Ford's algorithm (Gondran & Minoux, 1984), (Le Pape, 1988), (Cesta & Oddi, 1996). This $O(n * m)$ algorithm, where n is the number of activities and m the number of temporal constraints, guarantees that the earliest and latest start and end times of activities are always consistent with respect to the temporal constraints. In other terms, if no contradiction is detected, then the earliest start and end times of activities satisfy all the temporal constraints, and the latest start and end times of activities satisfy all the temporal constraints.

The third category of constraints includes the resource constraints, stating that each machine M can execute at most one activity at a time. These constraints are the most complex to propagate. Two constraint propagation techniques, inspired by previous work on non-preemptive scheduling, have been developed. The first technique relies on resource timetables (Le Pape, 1994) and applies to both "disjunctive" resources, which can execute only one activity at a time, and "cumulative" resources, of capacity $m > 1$. A timetable is maintained for each resource, in order to keep track of the required and available capacity at any time t. Propagation occurs both from activities to resource timetables, and from resource timetables to activities.

- From activities to resources
  When an activity is known to execute at time t, this activity requires its resources at time t. The timetable of each required resource can consequently be updated.
- From resources to activities
  The resource timetable is used to incrementally update time-bounds of activities. The earliest end time of each activity is updated to ensure that between its earliest start time and its earliest end time, there are enough "free" time intervals on the resource to let the activity execute. Needless to say, if the resource timetable proves that the activity cannot start before time t, its earliest start time is updated and the timetable mechanism is iterated. A similar technique is used to update the latest start time and the latest end time of the activity.

The second resource constraint propagation technique relies on edge-finding. It reasons about the order in which several activities can execute on a given resource. Edge-finding, initially developed for non-preemptive disjunctive scheduling (Carlier & Pinson, 1990), (Nuijten, 1994), consists of determining whether an activity A can, cannot, or must, execute before (or after) a set of activities Ω which require the same resource. Two types of conclusions can then be drawn: new ordering relations ("edges" in the graph representing the possible orderings of activities) and new time-bounds, *i.e.*, strengthened earliest and latest start and end times of activities. The preemptive case is more complex since several activities can preempt one another. Then edge-finding consists of determining whether an activity A can, cannot, or must, start or end before (or after) a set

of activities Ω. A mixed edge-finding algorithm, running in a quadratic number of steps, is proposed in (Le Pape & Baptiste, 1996). Given a machine M and the current time-bounds of activities in $acts(M)$, it computes, for each activity A in $acts(M)$:

- when A is not interruptible: the earliest time at which A could start and the latest time at which A could end, if all the other activities in $acts(M)$ were interruptible;
- when A is interruptible: the earliest time at which A could end and the latest time at which A could start, if all the other activities in $acts(M)$ were interruptible.

The edge-finding algorithm enables the deduction of better bounds, *i.e.*, more precise earliest and latest start and end times, but it is expensive in CPU time and, in particular, much less incremental than the timetable-based algorithm. Indeed, for a given machine M, each run of the edge-finding algorithm involves all the activities in $acts(M)$. Table 1 provides the average time per node we have observed for a few PJSSP instances of different sizes, using the ten problem-solving strategies described in the next sections. Only the hardest instances have been considered because on easier instances most of the time is spent proving that the optimal solution has been found. In this table, n denotes the number of jobs, m the number of machines, $c(tt)$ the average cost (CPU time in milliseconds on a PC Dell at 200MHz running Windows NT) per node when the timetable mechanism is used, and $c(ef)$ the average cost per node when the edge-finding algorithm is used.

Several important things appear. First, the table confirms that the edge-finding algorithm is costly: on the largest instances, the time spent per node is multiplied by a factor between 4 and 5.5 when the edge-finding algorithm is used. Second, and more surprisingly, $c(ef)$ increases linearly (and even a little less than linearly) with the size of the problem. The $c(ef) / (n * m)$ ratio varies between 0.035, for the smallest problems, and 0.029. The $c(tt) / (n * m)$ ratio seems to depend on the number of machines: it varies between 0.012 and 0.015 for the instances with 5 machines, and between 0.0056 and 0.0081 for the larger instances.

Needless to say, many factors impact the measured time per node. First, the edge-finding algorithm tends to find dead ends sooner than the timetable propagation algorithm, and thus is more often in a situation where many activities are unscheduled. Second, the jobs tend to compete less for the machines when the number of machines is high (because the propagation of the temporal constraints results in spreading the competing activities in time). This impacts both the time needed for constraint propagation to reach quiescence (at a given node) and the shape of the search tree. When constraint propagation is restricted to the use of timetables, the making of one decision tends to trigger resource constraint propagation on a limited number of machines, which may explain the behavior of the $c(tt) / (n * m)$ ratio. A more systematic computational study would be necessary to further explain this behavior.

n	m	*instances*	$c(tt)$	$c(tt) / (n * m)$	$c(ef)$	$c(ef) / (n * m)$
10	5	LA02, LA03	0.62	0.0124	1.76	0.0352
15	5	LA07, LA08	0.99	0.0132	2.27	0.0303
20	5	LA12, LA15	1.49	0.0149	3.13	0.0313
10	10	FT10, ABZ5, ORB3	0.81	0.0081	3.36	0.0336
15	10	LA24, LA25	0.94	0.0063	4.71	0.0314
20	10	LA27, LA29	1.40	0.0070	6.12	0.0306
15	15	LA36, LA38, LA40	1.25	0.0056	6.85	0.0304
30	10	LA31, LA35	2.16	0.0072	8.70	0.0290

Table 1. CPU time per node with timetables *vs*. edge-finding.

5. Heuristic control of the DFS algorithm

Several points of flexibility remain in the DFS algorithm. Let us first consider the course of action to follow when a new solution has been found by the branch and bound algorithm. The alternative is either to "*continue*" the search for a better solution in the current search tree (with a new constraint stating that the makespan must be smaller than the current one) or to "*restart*" a brand new branch and bound procedure. The main advantage of restarting the search is that the heuristic choices can rely on the result of the new propagation (based on the new upper bound), which shall lead to a better exploration of the search tree. The drawback is that parts of the new search tree may have been explored in a previous iteration, which results in redoing the same unfruitful work.

As far as the PJSSP is concerned, the restart strategy brings another point of flexibility, concerning the selection of an activity A_k in *NDK*. A basic strategy consists of selecting A_k according to a specific heuristic rule. In our case, selecting the activity with the smallest latest end time (Earliest Due Date rule) seems reasonable since it corresponds to the rule which optimally solves the preemptive one-machine problem (see, for instance, (Carlier & Pinson, 1990)). However, we can also use a strategy which relies on the best schedule S computed so far. We propose to select the activity A_k with minimal $d_S(A_k)$. Our hope is that this should help to find a better schedule when there exists one that is "close" to the previous one.

In addition, we can use the Jackson derivation operator J and its symmetric counterpart K to improve the current schedule. Whenever a new schedule S is found, derivations J and K can be applied to improve the current schedule prior to restarting the search. Several strategies can be considered, *e.g.*, apply only J, apply only K, apply a sequence of Js and Ks. In a previous paper (Le Pape & Baptiste, 1997a), we applied systematically J to S and occasionally K to $J(S)$. After further experimentation, we decided to focus on the following scheme, which performs much better on average:

- compute $J(S)$ and $K(S)$;

- replace S with the best schedule among $J(S)$ and $K(S)$, if this schedule is strictly better than S (in our implementation, $J(S)$ is chosen if $J(S)$ and $K(S)$ have the same makespan);
- if S has been replaced by either $J(S)$ or $K(S)$, iterate.

Globally, this leads to five strategies based on depth first search: DFS-C-E, DFS-R-E, DFS-R-E-JK, DFS-R-B and DFS-R-B-JK, where C, R, E, B, JK stand respectively for "Continue search in the same tree", "Restart search in a new tree", "select activities according to the Earliest due date rule", "select activities according to their position on the Best schedule met so far" and "apply JK derivation operators". We remark that, in fact, three other strategies, DFS-C-E-JK, DFS-C-B and DFS-C-B-JK could also be considered, but with a more complex implementation (*e.g.*, in DFS-R-E-JK, the same data structures are used to perform the depth-first search and apply the J and K operators; to implement DFS-C-E-JK, we would need to duplicate the schedule).

Table 2 provides the results obtained on the preemptive variant of the ten 10∗10 instances used by Applegate and Cook (1991) in their computational study of the non-preemptive job-shop scheduling problem. Each line of the table corresponds to a given "constraint propagation + search" combination, and provides the mean relative error (MRE, in percentage) obtained after 1, 2, 3, 4, 5, and 10 minutes of CPU time. For each instance, the relative error is computed as the difference between the obtained makespan and the optimal value, divided by the optimal value. The MRE is the average relative error over the ten instances. The optimal values have been obtained by running an exact algorithm, described in (Le Pape & Baptiste, 1998), with an average CPU time of 3.4 hours, and a maximum of 27 hours (for the ORB3 instance), on a PC Dell at 200MHz running Windows NT.

Table 3 provides the results obtained on the thirteen instances used by Vaessens, Aarts, and Lenstra (1994) to compare local search algorithms for the non-preemptive job-shop scheduling problem. As these instances differ in size, we allocated to each instance an amount of time proportional to the square of the number of activities in the instance. This means that column 1 corresponds to the allocation of 1 minute to a 10∗10 problem, 15 seconds for a 10∗5 problem, 4 minutes for a 20∗10 problem, etc. We used the square of the number of activities, because the time spent per node is approximately linear in the number of activities, and the number of decisions necessary to construct a schedule (*i.e.*, the depth of the search tree) is also proportional to the number of activities (hence the time necessary to reach the first solution tends to increase as the square of the number of activities). To our knowledge, five of the thirteen instances are, in the preemptive case, open (and we have been unable to solve them with our exact algorithm). For these instances, we applied a variant of edge-finding and "shaving" (Carlier & Pinson, 1994), (Martin & Shmoys, 1996), (Péridy, 1996) to obtain a lower bound, and thus to estimate the relative error.

These tables show that the use of the edge-finding technique enables the generation of good solutions in a limited amount of time. In addition, the DFS-R-B-JK variant clearly outperforms the other algorithms, especially when the edge-finding technique is used.

Propagation algorithm	Search strategy	1	2	3	4	5	10
Timetable	DFS-C-E	16.74	16.37	16.25	16.25	16.25	16.18
	DFS-R-E	16.74	16.42	16.40	16.37	16.37	16.18
	DFS-R-E-JK	8.95	8.95	8.95	8.95	8.95	8.33
	DFS-R-B	14.67	14.48	14.48	14.48	14.13	13.72
	DFS-R-B-JK	8.32	8.16	7.74	7.73	7.73	7.34
Edge-finding	DFS-C-E	5.23	4.64	3.80	3.09	2.94	1.55
	DFS-R-E	5.70	5.26	4.99	4.47	4.09	2.73
	DFS-R-E-JK	4.29	3.67	3.17	2.55	2.42	1.62
	DFS-R-B	4.23	3.68	3.41	2.82	2.80	1.41
	DFS-R-B-JK	1.69	1.32	0.86	0.80	0.79	0.65

Table 2. DFS results on the ten instances used in (Applegate & Cook, 1991)

Propagation algorithm	Search strategy	1	2	3	4	5	10
Timetable	DFS-C-E	16.28	16.04	16.03	15.96	15.96	15.96
	DFS-R-E	16.34	16.08	16.06	16.05	16.04	16.02
	DFS-R-E-JK	9.22	9.22	9.22	9.22	9.22	9.04
	DFS-R-B	14.55	14.36	14.28	14.28	14.28	14.25
	DFS-R-B-JK	8.82	8.70	8.60	8.59	8.59	7.84
Edge-finding	DFS-C-E	4.33	3.98	3.62	3.52	3.47	3.15
	DFS-R-E	4.99	4.80	4.49	4.25	3.96	3.72
	DFS-R-E-JK	4.02	3.74	3.32	3.26	3.22	3.03
	DFS-R-B	3.96	3.64	3.42	3.42	3.42	3.04
	DFS-R-B-JK	2.26	2.12	1.94	1.77	1.77	1.72

Table 3. DFS results on the thirteen instances used in (Vaessens *et al*., 1994)

6. Limited discrepancy search

Limited discrepancy search (LDS) (Harvey and Ginsberg, 1995) is an alternative to the classical depth first search algorithm. This technique relies on the intuition that heuristics make few mistakes through the search tree. Thus, considering the path from the root node of the tree to the first solution found by a DFS algorithm, there should be few "wrong turns" (*i.e.*, few nodes which were not immediately selected by the heuristic). The basic idea is to restrict the search to paths which do not diverge more than w times from the choices recommended by the heuristic. When $w = 0$, only the leftmost branch of the search tree is explored. When $w = 1$, the number of paths explored is linear in the depth of the search tree, since only one alternative turn is allowed for each path. Each

time this limited search fails, w is incremented and the process is iterated, until either a solution is found or it is proven that there is no solution. It is easy to prove that when w gets large enough, LDS is complete. At each iteration, the branches where the discrepancies occur close to the root of the tree are explored first (which makes sense when the heuristics are more likely to make mistakes early in the search). See (Harvey and Ginsberg, 1995) for details.

Several variants of the basic LDS algorithm can be considered:

- When the search tree is not binary, it can be considered that the i^{th} best choice according to the heuristic corresponds either to 1 or to $(i - 1)$ discrepancies. In the following, we consider it represents $(i - 1)$ discrepancies because the second best choice is often much better than the third, etc. In practice, this makes the search tree equivalent to a binary tree where each decision consists of either retaining or eliminating the best activity according to the heuristic.
- The first iteration may correspond either to $w = 0$ or to $w = 1$. In the latter case, one can also modify the order in which nodes are explored during the first iteration (*i.e.*, start with discrepancies far from the root of the tree). The results reported below are based on a LDS algorithm which starts with $w = 0$.
- (Korf, 1996) proposes an improvement based on an upper bound on the depth of the search tree. In our case, the depth of the search tree can vary a lot from a branch to another (even though it remains linear in the size of the problem), so we decided not to use Korf's variant. This implies that, to explore a complete tree, our implementation of LDS has a very high overhead over DFS.
- (Walsh, 1997) proposes a variant called "Depth-bounded Discrepancy Search" (DDS), in which any number of discrepancies is allowed, provided that all the discrepancies occur up to a given depth. This variant is recommended when the heuristic is very unlikely to make mistakes in the middle and at the bottom of the search tree (*i.e.*, when almost all mistakes occur at low depth). We tried both classical LDS and DDS, and decided to focus on classical LDS, which appears to perform better in our case. Some experimental results with DDS are provided below.

Table 4 provides the results obtained by the five LDS variants, LDS-C-E, LDS-R-E, LDS-R-E-JK, LDS-R-B and LDS-R-B-JK, on the ten instances used by Applegate and Cook. Table 5 provides the results for the thirteen instances used by Vaessens, Aarts, and Lenstra. These tables clearly show that the LDS algorithms provide better results on average than the corresponding DFS algorithms. Figures 2 and 3 present the evolution of the mean relative error for the eight "constraint propagation + search" combinations in which the J and K operators are used. The combination of the edge-finding constraint propagation algorithm with LDS-R-B-JK appears to be the clear winner.

Propagation algorithm	Search strategy	1	2	3	4	5	10
Timetable	LDS-C-E	9.55	9.43	9.16	9.08	8.95	8.52
	LDS-R-E	10.46	9.87	9.22	8.98	8.98	8.52
	LDS-R-E-JK	7.68	6.75	6.75	6.75	6.75	5.98
	LDS-R-B	5.75	4.95	4.42	4.16	4.16	3.61
	LDS-R-B-JK	6.14	5.67	5.59	5.14	5.07	4.20
Edge-finding	LDS-C-E	3.20	2.70	2.42	2.08	1.77	1.41
	LDS-R-E	3.52	2.90	2.67	2.39	2.25	1.66
	LDS-R-E-JK	2.17	2.03	1.86	1.71	1.38	1.24
	LDS-R-B	1.10	0.95	0.75	0.74	0.60	0.39
	LDS-R-B-JK	0.64	0.64	0.55	0.36	0.32	0.23

Table 4. LDS results on the ten instances used in (Applegate & Cook, 1991)

Propagation algorithm	Search strategy	1	2	3	4	5	10
Timetable	LDS-C-E	11.43	11.12	11.08	10.89	10.53	10.43
	LDS-R-E	11.53	11.24	11.12	10.95	10.59	10.40
	LDS-R-E-JK	7.98	7.97	7.95	7.89	7.87	7.40
	LDS-R-B	6.58	5.92	5.64	5.44	5.26	4.68
	LDS-R-B-JK	5.93	5.82	5.78	5.66	5.66	4.60
Edge-finding	LDS-C-E	3.57	3.02	2.85	2.77	2.57	2.27
	LDS-R-E	4.33	3.37	3.14	2.91	2.78	2.39
	LDS-R-E-JK	2.43	2.20	2.03	1.82	1.73	1.61
	LDS-R-B	2.25	1.80	1.76	1.74	1.58	1.13
	LDS-R-B-JK	1.75	1.28	1.03	0.92	0.92	0.79

Table 5. LDS results on the thirteen instances used in (Vaessens *et al*., 1994)

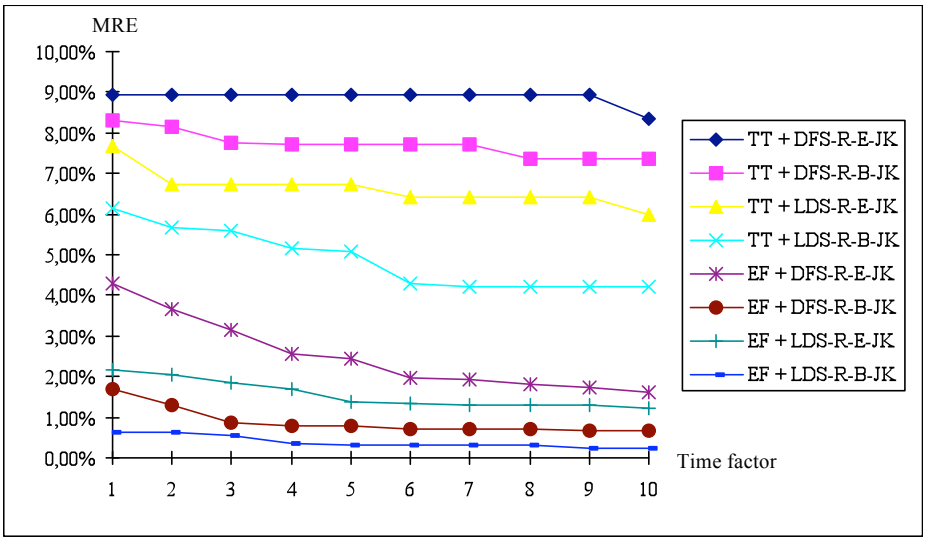


Figure 2. Results on the ten instances used in (Applegate & Cook, 1991)

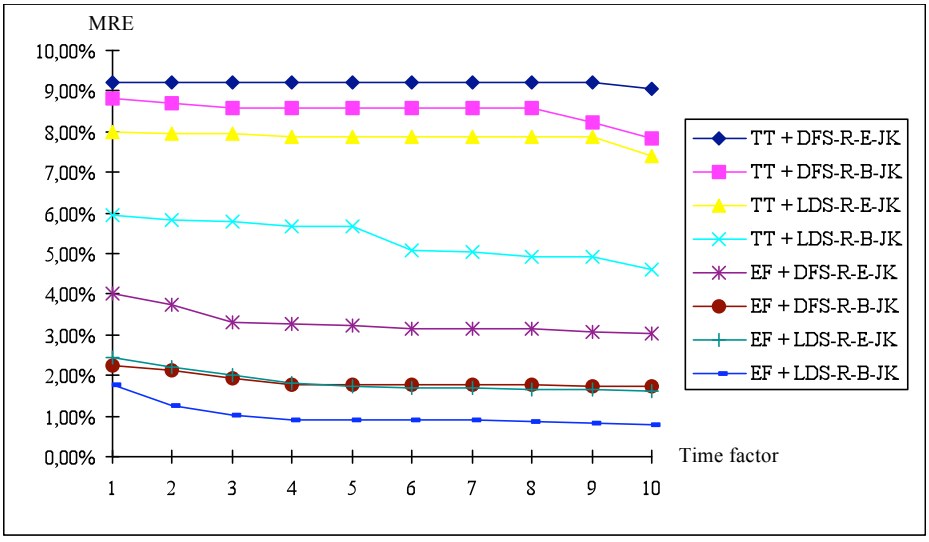
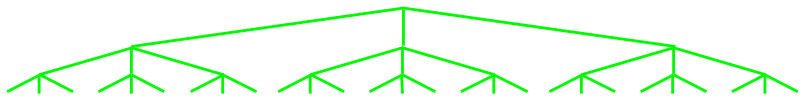


Figure 3. Results on the thirteen instances used in (Vaessens *et al.*, 1994)

Tables 6 and 7 provide the results obtained using DFS and variants of LDS with the edge-finding constraint propagation algorithm and the R-E-JK and R-B-JK heuristic search strategies. In these tables, LDS^N and DDS^N refer to the application of the LDS and DDS principles on the N-ary tree, where each son of a node corresponds to scheduling one candidate activity on the corresponding machine. LDS and DDS (without the N) correspond to the application of LDS and DDS to the binary tree in which each decision consists of either scheduling or postponing the best candidate. Figure 4 illustrates the difference between the five algorithms, DFS, LDS, LDS^N, DDS, and DDS^N, on a ternary tree of depth three. "ITE" shows at which iteration each leaf of the tree is attained (starting with iteration number 0) and "ORD" shows in which order the leaves are visited. It shall be noted that, to explore a complete tree, LDS and DDS have a high overhead over DFS. Hence, LDS and DDS must quickly repair the bad decisions to beat DFS. LDS^N and DDS^N have a much smaller overhead on a complete tree. Hence, their performance is less dependent on the quick repair of the "worst" decisions. However, in LDS^N and DDS^N, the 2^{nd}, 3^{rd}, ... and N^{th} sons of a node are considered equal, which can lead to the early exploration of unpromising branches.



ITE																										
DFS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LDS	0	1	2	1	2	3	2	3	4	1	2	3	2	3	4	3	4	5	2	3	4	3	4	5	4	5	6
LDS^N	0	1	1	1	2	2	1	2	2	1	2	2	2	3	3	2	3	3	1	2	2	2	3	3	2	3	3
DDS	0	3	4	2	4	5	3	4	5	1	4	5	3	5	6	4	5	6	2	4	5	3	5	6	4	5	6
DDS^N	0	3	3	2	3	3	2	3	3	1	3	3	2	3	3	2	3	3	1	3	3	2	3	3	2	3	3
ORD																										
DFS	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	&
LDS	A	D	J	C	I	Q	H	P	W	B	G	O	F	N	V	M	U	Z	E	L	T	K	S	Y	R	X	&
LDS^N	A	F	G	D	P	Q	E	R	S	B	J	K	H	T	U	I	V	W	C	N	O	L	X	Y	M	Z	&
DDS	A	E	I	C	J	P	F	K	Q	B	L	R	G	S	X	M	T	Y	D	N	U	H	V	Z	O	W	&
DDS^N	A	J	K	D	L	M	E	N	O	B	P	Q	F	R	S	G	T	U	C	V	W	H	X	Y	I	Z	&

Figure 4. The behavior of the five algorithms on a balanced ternary tree.

Propagation algorithm	Search strategy	1	2	3	4	5	10
Edge-finding	DFS-R-E-JK	4.29	3.67	3.17	2.55	2.42	1.62
	LDS-R-E-JK	2.17	2.03	1.86	1.71	1.38	1.24
	LDS^N-R-E-JK	2.66	2.13	1.93	1.74	1.62	1.27
	DDS-R-E-JK	2.81	2.20	1.93	1.71	1.67	1.34
	DDS^N-R-E-JK	3.17	2.50	2.33	2.25	1.63	1.51
	DFS-R-B-JK	1.69	1.32	0.86	0.80	0.79	0.65
	LDS-R-B-JK	0.64	0.64	0.55	0.36	0.32	0.23
	LDS^N-R-B-JK	1.10	0.91	0.70	0.64	0.53	0.47
	DDS-R-B-JK	2.01	1.68	1.29	1.11	0.89	0.72
	DDS^N-R-B-JK	1.98	1.63	1.34	1.21	1.03	0.81

Table 6. LDS variants on the ten instances used in (Applegate & Cook, 1991)

Propagation algorithm	Search strategy	1	2	3	4	5	10
Edge-finding	DFS-R-E-JK	4.02	3.74	3.32	3.26	3.22	3.03
	LDS-R-E-JK	2.43	2.20	2.03	1.82	1.73	1.61
	LDS^N-R-E-JK	2.61	2.32	2.21	1.93	1.86	1.81
	DDS-R-E-JK	3.46	3.10	2.98	2.30	2.29	2.16
	DDS^N-R-E-JK	3.69	2.96	2.90	2.79	2.61	2.39
	DFS-R-B-JK	2.26	2.12	1.94	1.77	1.77	1.72
	LDS-R-B-JK	1.75	1.28	1.03	0.92	0.92	0.79
	LDS^N-R-B-JK	1.50	0.99	0.87	0.86	0.86	0.78
	DDS-R-B-JK	2.68	2.47	2.20	1.56	1.51	1.26
	DDS^N-R-B-JK	2.68	2.40	1.87	1.59	1.56	1.32

Table 7. LDS variants on the thirteen instances used in (Vaessens *et al*., 1994)

LDS on the binary tree appears to be the best overall alternative. LDS^N-R-B-JK also performs very well, in particular on the thirteen instances of (Vaessens *et al*., 1994). By contrast, DDS and DDS^N do not perform that well. In particular, DDS-R-B-JK and DDS^N-R-B-JK do not perform better than DFS-R-B-JK in Table 6 and for the first values of the time factor in Table 7.

Globally, it appears that the edge-finding technique is the most crucial of the techniques we used to provide good solutions to the PJSSP. On the 13 instances used in (Vaessens *et al*., 1994), all the algorithms that use the edge-finding technique perform better than all the algorithms that do not. Tables 8 and 9 provide the average gains in MRE over the 8 DFS-R-∗-∗ and LDS-R-∗-∗ scenarios, when each of the other techniques is added. When edge-finding is not used, LDS appears as the second most

important technique. When edge-finding is used, the contributions of the different techniques are closer one to the other. This confirms the observation in (Beck *et al*., 1997) that LDS appears to help the weaker algorithms to a greater extent than the stronger algorithms.

Propagation algorithm	Search strategy	1	2	3	4	5	10
Timetable	LDS	4.66	5.19	5.40	5.62	5.56	5.82
	B	2.24	2.18	2.27	2.38	2.49	2.53
	JK	4.13	4.05	3.87	3.85	3.79	4.04
Edge-finding	LDS	2.12	1.85	1.65	1.36	1.39	0.72
	B	2.00	1.82	1.78	1.60	1.41	1.14
	JK	1.44	1.29	1.35	1.25	1.21	0.61

Table 8. Average gains on the ten instances used in (Applegate & Cook, 1991)

Propagation algorithm	Search strategy	1	2	3	4	5	10
Timetable	LDS	4.23	4.35	4.41	4.55	4.69	5.01
	B	2.30	2.43	2.51	2.53	2.48	2.87
	JK	4.26	3.97	3.89	3.84	3.71	4.12
Edge-finding	LDS	1.12	1.41	1.31	1.33	1.34	1.40
	B	1.39	1.32	1.20	1.10	1.00	1.02
	JK	1.26	1.07	1.12	1.13	1.03	0.78

Table 9. Average gains on the thirteen instances used in (Vaessens *et al*., 1994)

Tables 10 and 11 detail the results obtained by the "Edge-finding + LDS-R-B-JK" combination, and provide the values of the optimal solutions, when these are known, or of the best lower and upper bounds we have for the five open instances. The upper bounds for LA24 (909) and LA38 (1168) have been obtained by LDS^N-R-B. The upper bound for LA29 (1145) has been obtained by LDS^N-R-B-JK. The upper bound for LA40 has been obtained by a variant of LDS-R-B-JK, in which $K(J(S))$ and $J(K(S))$ are calculated in place of $J(S)$ and $K(S)$. The upper bounds for ABZ5, ORB3 and LA25 have been obtained by a systematic depth-first-search algorithm (with long CPU times).

In these tables, a bold font is used to indicate when an optimal solution has been reached. For example, we see that, for seven of the ten Applegate and Cook instances, the optimal solution is reached in less than 5 minutes.

Instance	LB / UB	1	2	3	4	5	10
FT10	900	908	908	907	907	907	**900**
ABZ5	1203	1206	1206	1206	1206	1206	1206
ABZ6	924	**924**	**924**	**924**	**924**	**924**	**924**
LA19	812	818	818	**812**	**812**	**812**	**812**
LA20	871	**871**	**871**	**871**	**871**	**871**	**871**
ORB1	1035	1039	1039	1039	1039	**1035**	**1035**
ORB2	864	**864**	**864**	**864**	**864**	**864**	**864**
ORB3	973	1013	1013	1013	994	994	993
ORB4	980	**980**	**980**	**980**	**980**	**980**	**980**
ORB5	849	**849**	**849**	**849**	**849**	**849**	**849**

Table 10. Detailed results on the ten instances used in (Applegate & Cook, 1991)

Instance	LB / UB	1	2	3	4	5	10
FT10	900	908	908	907	907	907	**900**
LA02	655	**655**	**655**	**655**	**655**	**655**	**655**
LA19	812	818	818	**812**	**812**	**812**	**812**
LA21	1033	1056	1056	1034	**1033**	**1033**	**1033**
LA24	909	931	915	915	915	915	915
LA25	940 / 947	972	967	966	966	966	966
LA27	1235	1239	1239	1239	1239	1239	**1235**
LA29	1119 / 1145	1183	1158	1158	1158	1158	1152
LA36	1250 / 1252	1262	1253	1252	1252	1252	1252
LA37	1397	**1397**	**1397**	**1397**	**1397**	**1397**	**1397**
LA38	1141 / 1168	1185	1174	1174	1173	1173	1173
LA39	1221	**1221**	**1221**	**1221**	**1221**	**1221**	**1221**
LA40	1198 / 1199	1225	1225	1225	1210	1210	1209

Table 11. Detailed results on the thirteen instances used in (Vaessens *et al*., 1994)

7. Conclusion and perspectives

In this paper, we have presented the algorithms we developed for the preemptive job-shop scheduling problem, and the results we obtained. Ten heuristic search strategies, combined with two different constraint propagation techniques, have been presented and compared. The best combination, which relies on "limited discrepancy search" and on "edge-finding" techniques, provides excellent solutions to the preemptive job-shop scheduling problem, in a reasonable amount of time.

All the algorithms presented in this paper have been implemented in CLAIRE, a high-level object-oriented programming language with backtracking and rule processing

capabilities (Caseau & Laburthe, 1996b). The constraint propagation techniques presented in Section 4 have been integrated in CLAIRE SCHEDULE, a library for preemptive and non-preemptive scheduling (Le Pape & Baptiste, 1997b). The source code of the DFS and LDS algorithms is available as an example of use of CLAIRE SCHEDULE, on the CLAIRE web site (http://www.dmi.ens.fr/users/laburthe/claire.html).

The constraint propagation techniques presented in Section 4 are also used in an industrial project scheduling application. This application solves mixed project scheduling problems (half of the activities are interruptible) involving up to 40 activities, to be executed on 3 to 5 resources. About 70 additional constraints apply. Half of these additional constraints are preferences (with different levels of importance) which are more or less conflicting depending on the problem data. Experiments have shown that the use of the edge-finding technique results in a better satisfaction of the preferences, in a few minutes of CPU time.

Several extensions of this work could be considered in the future. First, slightly more complex algorithms, based on the continuation of a unique search tree, could be implemented. Given our current results, LDS-C-B-JK would be a good candidate. Second, it would be interesting to test other techniques on the preemptive job-shop scheduling problem, in particular genetic algorithms and tabu search, since good results have been obtained with these techniques in the non-preemptive case (Mattfelfd, 1996), (Nowicki & Smutnicki, 1996), (Vaessens *et al*., 1994). "Shifting bottleneck" and "shuffle" algorithms (Adams *et al*., 1988), (Applegate & Cook, 1991), which are more easily amenable to a constraint programming implementation (Baptiste *et al*., 1995), (Caseau & Laburthe, 1995), could also be studied. Given a preemptive schedule, one could try to re-optimize each machine while keeping the schedules of the other machines fixed. Also, since our approach relies on ordering the activities (in terms of priority), one could try to randomly select some ordering relations to be relaxed, and re-optimize these relations, as in (Baptiste *et al*., 1995).

Also, the PJSSP in itself is not that practically relevant. Industrial applications tend to include both interruptible and non-interruptible activities, both disjunctive and cumulative resources, optimization criteria that differ from makespan minimization, and additional constraints such as setup times, minimal and maximal delays between activities, etc. Even though some of the techniques described in the paper, in particular the constraint propagation techniques, can be generalized to more complex problems, it is unclear how well more general problems can be tackled. Hence, systematic experimental studies ought to be performed on multiple extensions of the PJSSP. Two simple extensions that could be looked at are the mixed job-shop scheduling problem, in which some activities are interruptible and some are not, and the preemptive resource-constrained project scheduling problem (Demeulemeester, 1992). We believe that efficient constraint propagation algorithms could be developed for the preemptive cumulative case, and hope our results on the preemptive job-shop scheduling problem will encourage other researchers to pursue work in this area.

Acknowledgments

The authors would like to thank the referees for many constructive comments on this paper, as well as the referees of a previous version, presented at the CP'97 workshop on industrial constraint-directed scheduling. We also received many comments at the workshop itself, which led to significant improvements of this experimental study.

References

Adams, J., Balas, E, & Zawack, D. (1988). "The Shifting Bottleneck Procedure for Job-Shop Scheduling," Management Science 34, 391-401.

Aggoun, A. & Beldiceanu, N. (1993). "Extending CHIP in Order to Solve Complex Scheduling and Placement Problems," Mathematical and Computer Modelling 17, 57-73.

Applegate, D. & Cook, W. (1991). "A Computational Study of the Job-Shop Scheduling Problem," ORSA Journal on Computing 3, 149-156.

Baptiste, Ph. & Le Pape, C. (1995). "A Theoretical and Experimental Comparison of Constraint Propagation Techniques for Disjunctive Scheduling," Proc. 14th International Joint Conference on Artificial Intelligence, 600-606, Morgan Kaufmann.

Baptiste, Ph., Le Pape, C., & Nuijten, W.P.M. (1995). "Constraint-Based Optimization and Approximation for Job-Shop Scheduling," Proc. AAAI-SIGMAN Workshop on Intelligent Manufacturing Systems, 5-16.

Baptiste, Ph. & Le Pape, C. (1997). "Constraint Propagation and Decomposition Techniques for Highly Disjunctive and Highly Cumulative Project Scheduling Problems," Proc. 3rd International Conference on Principles and Practice of Constraint Programming, 375-389, Springer-Verlag.

Beck, H. (1992). "Constraint Monitoring in TOSCA," Proc. AAAI Spring Symposium on Practical Approaches to Planning and Scheduling.

Beck, J.C., Davenport, A.J., Sitarski, E.M., & Fox, M.S. (1997). "Texture-Based Heuristics for Scheduling Revisited," Proc. 14th National Conference on Artificial Intelligence, MIT Press.

Brucker, P., Jurisch, B., & Sievers, B. (1994). "A Branch and Bound Algorithm for the Job-Shop Scheduling Problem," Discrete Applied Mathematics 49, 107-127.

Brucker, P. & Thiele, O. (1996). "A Branch and Bound Method for the General-Shop Problem with Sequence-Dependent Setup Times," OR Spektrum 18, 145-161.

Brucker, P., Knust, S., Schoo, A., & Thiele, O. (1997). "A Branch and Bound Algorithm for the Resource-Constrained Project Scheduling Problem," Working Paper, University of Osnabrück, Germany.

Burke, P. (1989). "Scheduling in Dynamic Environments," PhD Thesis, University of Strathclyde, Glasgow, United Kingdom.

Burke, P. & Prosser, P. (1991). "A Distributed Asynchronous System for Predictive and Reactive Scheduling," International Journal for Artificial Intelligence in Engineering 6, 106-124.

Carlier, J. & Pinson, E. (1990). "A Practical Use of Jackson's Preemptive Schedule for Solving the Job-Shop Problem," Annals of Operations Research 26, 269-287.

Carlier, J. & Pinson, E. (1994). "Adjustment of Heads and Tails for the Job-Shop Problem," European Journal of Operational Research 78, 146-161.

Caseau, Y. & Laburthe, F. (1994). " Improved CLP Scheduling with Task Intervals," Proc. 11th International Conference on Logic Programming, MIT Press.

Caseau, Y. & Laburthe, F. (1995). "Disjunctive Scheduling with Task Intervals," Technical Report, Ecole Normale Supérieure, Paris, France.

Caseau, Y. & Laburthe, F. (1996a). "Cumulative Scheduling with Task Intervals," Proc. Joint International Conference and Symposium on Logic Programming.

Caseau, Y. & Laburthe, F. (1996b). "CLAIRE: A Parametric Tool to Generate C++ Code for Problem Solving," Working Paper, Bouygues, Direction Scientifique, Saint-Quentin-en-Yvelines, France.

Cesta, A. & Oddi, A. (1996). "Gaining Efficiency and Flexibility in the Simple Temporal Problem," Proc. 3rd International Workshop on Temporal Representation and Reasoning, 45-50.

Cheng, C.-C. & Smith, S.F. (1994). "Generating Feasible Schedules under Complex Metric Constraints," Proc. 12th National Conference on Artificial Intelligence, 1086-1091, MIT Press.

Cheng, C.-C. & Smith, S.F. (1995a). "Applying Constraint Satisfaction Techniques to Job-Shop Scheduling," Technical Report, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Cheng, C.-C. & Smith, S.F. (1995b). "A Constraint-Posting Framework for Scheduling under Complex Constraints," Proc. AAAI-SIGMAN Workshop on Intelligent Manufacturing Systems, 64-75.

Collinot, A. & Le Pape, C. (1987). "Controlling Constraint Propagation," Proc. 10th International Joint Conference on Artificial Intelligence, 1032-1034, Morgan Kaufmann.

Colombani, Y. (1996). "Constraint Programming: An Efficient and Practical Approach to Solving the Job-Shop Problem," Proc. 2nd International Conference on Principles and Practice of Constraint Programming, 149-163, Springer-Verlag.

Colombani, Y. (1997). "Un modèle de résolution de contraintes adapté aux problèmes d'ordonnancement," PhD Thesis, Université de la Méditerranée, Aix-Marseille II, France (in French).

Demeulemeester, E. (1992). "Optimal Algorithms for Various Classes of Multiple Resource-Constrained Project Scheduling Problems," PhD Thesis, Katholieke Universiteit Leuven, Leuven, Belgium.

Erschler, J., Roubellat, F., & Vernhes, J.-P. (1976). "Finding Some Essential Characteristics of the Feasible Solutions for a Scheduling Problem," Operations Research 24, 774-783.

Erschler, J., Lopez, P., & Thuriot, C. (1991). "Raisonnement temporel sous contraintes de ressource et problèmes d'ordonnancement," Revue d'Intelligence Artificielle 5, 7-32 (in French).

Fox, M.S. (1983). "Constraint-Directed Search: A Case Study of Job-Shop Scheduling," PhD Thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Fox, M.S. & Smith, S.F. (1984). "ISIS: A Knowledge-Based System for Factory Scheduling," Expert Systems 1, 25-49.

Garey, M.R. & Johnson, D.S. (1979). "Computers and Intractability. A Guide to the Theory of NP-Completeness," W. H. Freeman and Company.

Gondran, M. & Minoux, M. (1984). "Graphs and Algorithms," John Wiley and Sons.

Harvey, W.D. & Ginsberg, M.L. (1995). "Limited Discrepancy Search," Proc. 14th International Joint Conference on Artificial Intelligence, 607-613, Morgan Kaufmann.

Korf, R.E. (1996). "Improved Limited Discrepancy Search," Proc. 13th National Conference on Artificial Intelligence, 286-291, MIT Press.

Le Pape, C. & Smith, S.F. (1987). "Management of Temporal Constraints for Factory Scheduling," Proc. IFIP TC 8/WG 8.1 Working Conference on Temporal Aspects in Information Systems, 159-170, North-Holland.

Le Pape, C. (1988). "Des systèmes d'ordonnancement flexibles et opportunistes," PhD Thesis, University Paris XI, Orsay, France (in French).

Le Pape, C. (1991). "Constraint Propagation in Planning and Scheduling," Technical Report, Stanford University, Palo Alto, California.

Le Pape, C. (1994). "Implementation of Resource Constraints in ILOG SCHEDULE: A Library for the Development of Constraint-Based Scheduling Systems," Intelligent Systems Engineering 3, 55-66.

Le Pape, C. & Baptiste, Ph. (1996). "Constraint Propagation Techniques for Disjunctive Scheduling: The Preemptive Case," Proc. 12th European Conference on Artificial Intelligence, 619-623, John Wiley and Sons.

Le Pape, C. & Baptiste, Ph. (1997a). "An Experimental Comparison of Constraint-Based Algorithms for the Preemptive Job-Shop Scheduling Problem," Proc. CP Workshop on Industrial Constraint-Directed Scheduling.

Le Pape, C. & Baptiste, Ph. (1997b). "A Constraint Programming Library for Preemptive and Non-Preemptive Scheduling," Proc. 3rd International Conference and Exhibition on the Practical Application of Constraint Technology, 237-256, The Practical Application Company.

Le Pape, C. & Baptiste, Ph. (1998). "Resource constraints for preemptive job-shop scheduling," Constraints, to appear.

Lock, H.C.R. (1996). "An Implementation of the Cumulative Constraint," Working Paper, University of Karlsruhe, Karlsruhe, Germany.

Lopez, P. (1991). “Approche énergétique pour l'ordonnancement de tâches sous contraintes de temps et de ressources,” PhD Thesis, Université Paul Sabatier, Toulouse, France (in French).

Martin, P. & Shmoys, D.B. (1996). “A New Approach to Computing Optimal Schedules for the Job-Shop Scheduling Problem,” Proc. 5th International Conference on Integer Programming and Combinatorial Optimization.

Mattfeld, D.C. (1996). “Evolutionary Search and the Job-Shop: Investigations on Genetic Algorithms for Production Scheduling,” Physica-Verlag.

Nowicki, E. & Smutnicki, C. (1996). “A Fast Taboo Search Algorithm for the Job-Shop Problem,” Management Science 42, 797-813.

Nuijten, W.P.M. & Aarts, E.H.L. (1994). “Constraint Satisfaction for Multiple Capacitated Job-Shop Scheduling,” Proc. 11th European Conference on Artificial Intelligence, 635-639, John Wiley and Sons.

Nuijten, W.P.M. (1994). “Time and Resource Constrained Scheduling: A Constraint Satisfaction Approach,” PhD Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands.

Nuijten, W.P.M. & Aarts, E.H.L. (1996). “A Computational Study of Constraint Satisfaction for Multiple Capacitated Job-Shop Scheduling,” European Journal of Operational Research 90, 269-284.

Péridy, L. (1996). “Le problème de job-shop : arbitrages et ajustements,” PhD Thesis, Université de Technologie de Compiègne, Compiègne, France (in French).

Pinson, E. (1988). “Le problème de job-shop,” PhD Thesis, University Paris VI, Paris, France (in French).

Prosser, P. (1990). “Distributed Asynchronous Scheduling,” PhD Thesis, University of Strathclyde, Glasgow, United Kingdom.

Rit, J.-F. (1986). “Propagating Temporal Constraints for Scheduling,” Proc. 5th National Conference on Artificial Intelligence, 383-388, MIT Press.

Smith, S.F., Fox, M.S., & Ow, P.S. (1986). “Constructing and Maintaining Detailed Production Plans: Investigations into the Development of Knowledge-Based Factory Scheduling Systems,” AI Magazine 7, 45-61.

Smith, S.F. (1992). “Knowledge-Based Production Management: Approaches, Results and Prospects,” Production Planning and Control 3, 350-380.

Steele, G.L., Jr. (1980). “The Definition and Implementation of a Computer Programming Language Based on Constraints,” PhD Thesis, Massachusetts Institute of Technology.

Vaessens, R.J.M., Aarts, E.H.L, & Lenstra, J.K. (1994). “Job-Shop Scheduling by Local Search,” COSOR Memorandum 94-05, Eindhoven University of Technology, Eindhoven, The Netherlands.

Varnier, C., Baptiste, P., & Legeard, B. (1993). “Le traitement des contraintes disjonctives dans un problème d'ordonnancement : exemple du Hoist Scheduling Problem,” Proc. 2èmes journées francophones de programmation logique, 343-363 (in French).

Walsh, T. (1997). “Depth-bounded Discrepancy Search,” Proc. 15th International Joint Conference on Artificial Intelligence, Morgan Kaufmann.

Zweben, M. & Fox, M., editors (1994). “Intelligent Scheduling,” Morgan Kaufmann.