Constraint-Based Scheduling: A Tutorial

Claude Le Pape

ILOG S. A.
9, rue de Verdun
F-94253 Gentilly Cedex
Tel: +33 1 49 08 29 67
Fax: +33 1 49 08 35 10
Email: clepape@ilog.fr

Given a set of resources with given capacities, a set of activities with given processing times and resource requirements, and a set of temporal constraints between activities, a "pure" scheduling problem consists of deciding when to execute each activity, so that both temporal constraints and resource constraints are satisfied. Most scheduling problems can easily be represented as instances of the constraint satisfaction problem (Kumar, 1992): given a set of variables, a set of possible values (domain) for each variable, and a set of constraints between the variables, assign a value to each variable, so that all the constraints are satisfied.

The diversity of scheduling problems, the existence of many specific constraints or preferences in each problem, and the emergence of efficient constraint-based scheduling algorithms in the mid-90s (Aggoun & Beldiceanu, 1993) (Nuijten, 1994) (Caseau & Laburthe, 1994) (Baptiste & Le Pape, 1995) (Colombani, 1996), have made constraint programming a method of choice for the resolution of complex industrial problems. In this tutorial, the main principles of constraint programming are discussed in terms of the corresponding advantages and drawbacks for the resolution of industrial scheduling problems. The modeling of scheduling problems and the use of specific constraint propagation techniques are then discussed. The development of practical heuristic search procedures is illustrated through an example, the preemptive job-shop scheduling problem, and, as a practical extension, a daily construction site scheduling problem. In the conclusion, the usefulness of mixing constraint programming with other techniques (linear programming, local search) is briefly discussed.

Principles and interest of constraint programming applied to scheduling problems

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.
- 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.
- 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.

Explicit problem definition

The main advantage of separating problem definition from problem-solving is obvious: it guarantees that the problem to be solved is precisely defined. Actually, a significant part of the design of a constraint-based scheduling system consists of eliminating any ambiguity from the problem statement. This can be very demanding. Indeed, some pieces of knowledge about the problem can be easier to integrate in a problem-solving algorithm than in a declarative specification of the problem. For example, knowledge of the current and usual practice (i.e., knowledge of the form "generally, we do that") is easy to incorporate in the form of heuristics in a decision-making procedure, while its status in terms of constraints and preferences can lead to long (and passionate!) debates about the qualities and the defects of the current practice. Such debates are often necessary to ensure that the projected software will deal with the right problem; yet they may as well result in pure cancellation of the software project if the participants cannot agree.

Another advantage of separating problem definition from problem-solving concerns the revision or the extension of the scheduling system when the 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 preferences, and to the removal of old constraints and preferences. In some cases, the same problem-solving algorithms will continue to apply, with a different problem definition as input. This is in huge contrast with the case in which the problem definition is "diluted" in lines and lines of problem-solving code.

Constraint propagation and search

The second important principle of constraint programming consists of distinguishing constraint propagation and decision-making search. Constraint propagation is a deductive activity which consists in deducing new constraints from existing constraints. For example, if an activity A must precede an activity B, and if A cannot be finished before 12:00 noon, constraint propagation will (normally) deduce that B cannot start before 12:00. Such information can prove very useful for the decision-maker (algorithm or human being) since it allows more informed decisions to be made. In addition, it can propagate in turn to other variables of the problem: if the minimal duration of B is two hours, then B cannot end before 2:00 p.m., etc. In some cases, this leads to determining that the problem (maybe augmented by some decisions) is insoluble. In such a case, either some constraints or some decisions must be removed. Unfortunately, constraint propagation cannot be perfect. A well-known conjecture on combinatorial problems and algorithms states that some combinatorial problems cannot be solved in an amount of time that grows as a polynomial function of the size of the problem. These problems are called "NP-hard" (Garey & Johnson, 1979). The problem of determining whether activities submitted to resource constraints can be executed within given deadlines (earliest start times and latest end times) is NP-hard, even if only one resource of capacity 1 is considered. As a result, constraint propagation cannot detect all inconsistencies between problem constraints (or it will take too much time to do so), and cannot provide perfect information about the earliest and latest start and end times of activities. To determine if there exists a schedule that meets the deadlines, one must search the possible combinations of scheduling decisions. The information provided by constraint propagation is extremely useful to guide this search. Yet one must be prepared to remove decisions when conflicts are (eventually) discovered.

Separating constraint propagation and search has multiple advantages. First, it allows the system 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 a decision-making algorithm as well as decisions made by a human user. Distinct decision-making algorithms can also be implemented and combined, if they rely on the same constraint propagation process. In an optimization context, this may lead to the development of several decision-making algorithms, dedicated for example to distinct combinations of optimization criteria. When preferences are considered, the same algorithm can also be launched several times, after activating different sets of preferences corresponding to different levels of importance.

Another important advantage of this separation is that precise conditions exist under which a constraint propagation + decision-making search algorithm is guaranteed to find a solution if one exists (Le Pape, 1992). It is important to note that these conditions do not enforce the instantiation of all the variables of the problem. Hence, a decision-making algorithm may just generate a set of constraints which (1) are proved compatible one with the others and (2) represent not a single schedule but a set of possible schedules. This can become useful when the "solution" is put into execution. While unforeseen events (e.g., activities that last longer than expected) would normally invalidate the current (unique) schedule, the same events may only reduce the set of schedules represented by the current constraints (i.e., the constraints derived from predictive scheduling and those arising from execution). If at least one schedule remains in this set, execution can continue without revising the solution (see, for example, (Collinot & Le Pape, 1991) (Lesaint, 1993) and, out of the constraint programming world, (Le Gall, 1989) (Le Gall & Roubellat, 1992) (Billaut, 1993).

Last, but not least, the separation of constraint propagation and decision-making allows the developer of a constraint-based application to reuse constraint propagation techniques developed for other applications. It is even current practice for application developers to use constraint-solving tools marketed by software houses. The main advantage of such practice is that the tool providers have invested significant effort in selecting, designing, and implementing powerful constraint propagation algorithms. For example, the "cumulative constraint" of CHIP (Aggoun & Beldicanu, 1993) and the specific constraint propagation algorithms of ILOG SCHEDULER (Le Pape, 1995) have offered a level of performance that is difficult to attain "from scratch" at a reasonable cost. In addition, some constraint-solving tools (e.g., ILOG SOLVER (Puget & Leconte, 1995)) offer a lot of facilities to mix different types of variables (e.g., integer and Boolean variables as in $BV(p) = true \Leftrightarrow end(A) \leq start(B)$), to define new constraints, to create disjunctions of constraints, etc. The main drawback is that the user of the tool cannot control all of what happens "in the box" and optimize the constraint propagation process with respect to his or her specific needs. Similarly, the user of the tool cannot easily maintain a trace of the propagation, which would allow the precise identification of those constraints that participate in a conflict, as well as intelligent forms of backtracking (e.g., (Stallman & Sussman, 1977) (Latombe, 1979) (Collinot & Le Pape, 1991) (Xiong et al., 1992) (Ginsberg, 1993) (Prosser, 1993)). This can be penalizing, even though the usefulness of intelligent forms of backtracking appears to be reduced when powerful constraint propagation methods are used.

Locality and incrementality of the constraint propagation process

The third important principle of constraint programming is that the constraint propagation process shall be as "local" and as "incremental" as possible. The "locality principle" (Steele, 1980) states that each constraint or each class of constraint is propagated independently of the existence or non-existence of other constraints. "Incrementality" means that new variables and constraints can be added at any time, without re-computing all the consequences of the new constraint set. Let us consider again the case of activity A which must precede activity B and cannot be finished before 12:00. Constraint propagation "deduces" that B cannot start before 12:00. If we now add a constraint stating that the minimal duration of B is two hours, constraint propagation immediately combines this new constraint with the fact that B cannot start before 12:00, to deduce that B cannot end before 2:00 p.m. Previous propagation results (the fact that B cannot start before 12:00) are exploited locally by the new constraint, without being recomputed. Ideally, one would also like the process of removing variables and constraints to be incremental. This, however, is not always feasible at low cost.

The locality and incrementality principle is fundamental as it enables the efficient combination of multiple constraint propagation techniques, associated with different classes of constraints. In particular, it allows multiple programmers to share libraries of constraints and augment such libraries with whatever new specific constraints are required for a given application. This, however, requires a general framework (programming language or library) designed to facilitate both (1) the integration of multiple classes of constraints in the same application and (2) the integration of constraints with the rest of the application. From a software engineering point of view, one must add "integrability" to the locality and incrementality principle, i.e., ensure that it will be possible to integrate all the components of an application in the same software system.

The locality and incrementality principle is sometimes hard to follow. First, because taking a global view often allows more powerful deductions to be made: the integration of "global" constraints is often required for efficiency reasons, but it is not an easy task. Second, because the principle a priori forbids the use of "dominance" arguments within constraint propagation. For example, let us imagine a scheduling problem in which an activity A is totally independent of the remainder of the problem, except for the fact that A requires a resource R, also required by other activities. Let us suppose that A lasts two hours, that A can execute between 12:00 noon and 2:00 p.m., and that no other activity can execute on R between 12:00 and 2:00. In that situation, one would like the system to automatically schedule A between 12:00 and 2:00, since if the problem is soluble, there will always be a solution (or even an optimal solution) such that A executes between 12:00 and 2:00. However, this cannot be done by propagation (even if there is an additional constraint stating that R must be used between 12:00 and 2:00) because a new activity B may be added to the problem later and chosen to execute between 12:00 and 2:00, thereby preventing A from executing between 12:00 and 2:00. In fact, pure deduction can often be cast as a local and incremental process, but default reasoning rules of the form "as long as X is possible and independent of the remainder of the problem, X is true" cannot be efficiently integrated in such a process. From a logical point of view, closed-world meta-constraints of the form "there cannot be more variables and constraints with such characteristics" (e.g., "there cannot be more activities requiring resource R") provide a solution to that problem. Yet a significant loss of incrementality is, in fact, incurred: after the statement of the meta-constraint, one cannot assign a new activity to R.

Representation of scheduling problems with variables and constraints

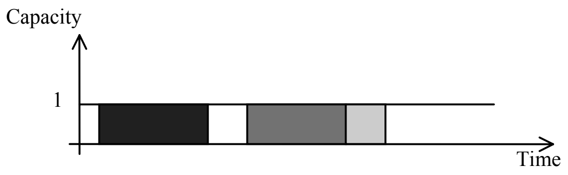
Given a set of resources with given capacities, a set of activities with given processing times and resource requirements, and a set of temporal constraints between activities, a "pure" scheduling problem consists of deciding when to execute each activity, so that both temporal constraints and resource constraints are satisfied. Most scheduling problems can easily be represented as instances of the constraint satisfaction problem (Kumar, 1992): given a set of variables, a set of possible values (domain) for each variable, and a set of constraints between the variables, assign a value to each variable, so that all the constraints are satisfied.

Several types of scheduling problems can be distinguished:

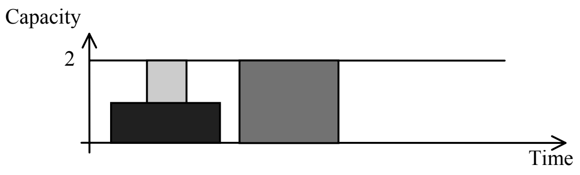
- In *disjunctive* scheduling, each resource can execute at most one activity at a time. In *cumulative* scheduling, a resource can run several activities in parallel, provided that the resource capacity is not exceeded.
- In *non-preemptive* scheduling, activities cannot be interrupted. Each activity *A* must execute without interruption from its start time to its end time. In *preemptive* scheduling, activities can be interrupted at any time, e.g., to let some other activities execute.

Many real-life scheduling problems are complex combinations of these basic problems.

First, real scheduling problems often include both disjunctive resources (e.g., a specific machine in a manufacturing shop, a crane on a construction site) and cumulative resources (e.g., groups of identical machines, teams of people with similar capabilities).



A disjunctive resource



A cumulative resource

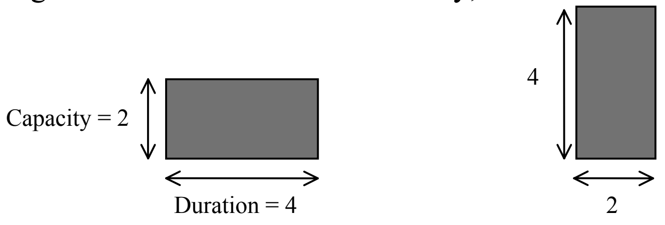
Second, some scheduling problems include both interruptible and non-interruptible activities. In many cases, technical or organizational rules limit the possibilities to interrupt an activity. In particular, it is often the case that an activity can be interrupted for a break (lunch, week-end) but not in favor of another activity.

A non-interruptible activity

An interruptible activity

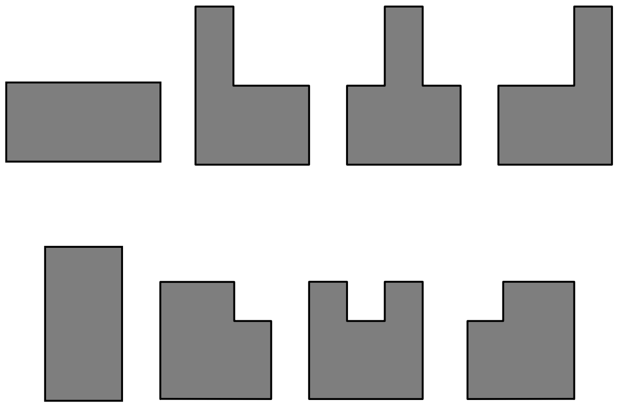
Third, there often exists some flexibility in the amount of capacity (e.g., number of workers) that can be assigned to some activities. Some activities require a predefined amount of resource capacity over their execution (e.g., two workers). For other activities, the capacity may be allowed to take several values, between which the scheduler has to make a choice. Yet, in such a case, the amount of energy (e.g., the number of man-hours necessary to complete the activity) is often given, or allowed to vary in a given range (e.g., between 2 and 4). More precisely, given the energy required by an activity, two cases can occur:

- either the scheduler has to assign a value to the required capacity (either 2, or 3, or 4) which will apply throughout the execution of the activity;



All possible configurations corresponding to an activity requiring
an energy of 8 and a constant amount of resource in [2, 4]

- or, at any execution point, the amount of resource to be used is an unknown value in a given interval (i.e., the capacity required by the resource can vary over time).



All possible configurations corresponding to an activity requiring an energy of
8 which can use at any execution time a capacity in [2, 4]

Finally, a variety of additional constraints (setups, variations of productivity during the day) often need to be taken into account as well. Such constraints will not be considered in the remainder of this section.

Representation of interruptible and non-interruptible activities

A non-preemptive scheduling problem can be encoded efficiently as a constraint satisfaction problem: two variables, start(A) and end(A), are associated with each activity A; they represent the start time and the end time of A. The smallest values in the domains of start(A) and end(A) are called the earliest start time and the earliest end time of A (EST_A and EET_A). Similarly, the greatest values in the domains of start(A) and end(A) are called the latest start time and the latest end time of A (LST_A and LET_A). The duration of the activity is an additional variable, defined as the difference between the end time and the start time of the activity.

A preemptive scheduling problem is more difficult to represent: one can either associate a set variable (i.e., a variable the value of which will be a set) set(A) with each activity A, or define a 0-1 variable w(A, t) for each activity A and time t; set(A) represents the set of times at which A executes, while w(A, t) assumes value 1 if and only if A executes at time t. The processing time pt(A), defined as the size of set(A), can be smaller than end(A) – start(A). Ignoring implementation details, let us note that:

- the value of w(A, t) is 1 if and only if t belongs to set(A).
- assuming time is discretized, start(A) and end(A) can be defined, in both the preemptive and the non-preemptive case, by $\text{start}(A) = \min_{t \in \text{set}(A)}(t)$ and $\text{end}(A) = \max_{t \in \text{set}(A)}(t + 1)$; in the preemptive case, these variables are often needed to connect activities together by temporal constraints.
- in the non-preemptive case, set(A) = [start(A) end(A)), with the interval [start(A) end(A)) closed on the left and open on the right so that |set(A)| = end(A) – start(A) = duration(A) = pt(A).

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 pt(A) or if the size of the upper bound (i.e., the sum of the sizes of the IUB_j) becomes smaller than pt(A), a contradiction is detected and a backtrack occurs. If the size of the lower bound (or of the upper bound) becomes equal to pt(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 from one another, can be used to update the bounds of set(A), start(A) and end(A).

- $\forall t,\ t < \text{start}(A) \Rightarrow t \notin \text{set}(A)$
- $\forall t,\ t \in \text{set}(A) \Rightarrow \text{start}(A) \leq t$
- $\forall t,\ \text{end}(A) \leq t \Rightarrow t \notin \text{set}(A)$
- $\forall t,\ t \in \text{set}(A) \Rightarrow t < \text{end}(A)$
- $\forall t,\ [\forall u < t,\ u \notin \text{set}(A)] \Rightarrow t \leq \text{start}(A)$
- $\forall t,\ [\forall u \geq t,\ u \notin \text{set}(A)] \Rightarrow \text{end}(A) \leq t$

- $\text{start}(A) \leq \max\{t \mid \exists S \subseteq \text{set}(A) \text{ such that } |S| = \text{pt}(A) \text{ and } \min(S) = t\}$
- $\text{end}(A) \geq \min\{t \mid \exists S \subseteq \text{set}(A) \text{ such that } |S| = \text{pt}(A) \text{ and } \max(S) = t - 1\}$

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 can immediately occur.

Representation and propagation of temporal constraints

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 a special variable representing the end time of the complete schedule, or the lateness of an activity subjected to a due date, 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, occur frequently in scheduling. These constraints are easily propagated through an incremental version of Ford's algorithm (Gondran & Minoux, 1984), (Le Pape, 1988), (Cesta & Oddi, 1996). This O(nm) 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.

Representation of resource constraints

Disjunctive resource constraints are obviously the easiest to represent: if A and B require the same disjunctive resource, set(A) and set(B) cannot intersect. In cumulative scheduling, it also often occurs that an activity A can execute at different rates, depending on the assigned resource capacity. In the most extreme case (referred to as the "fully elastic" case (Baptiste et al., 1999)), the amount of resource R assigned to an activity A can, at any time t, pick any value w(A, R, t) between 0 and the resource capacity C(R) (with $w(A, R, t) = 0 \Leftrightarrow w(A, t) = 0$), provided that the sum over time of the assigned capacity w(A, R, t) equals a given amount of energy w(A, R). The fully elastic case is interesting because the constraint propagation techniques that are developed for this case can be applied to any fully elastic, partially elastic, or non-elastic scheduling problem (because it is the less constrained).

Table 1 summarizes different classes of scheduling problems and the relevant constraints for each activity A and each (activity, resource) pair (A, R). The resource constraints concerning resource R state that at any time t, Σ_A w(A, R, t) cannot exceed the resource capacity C(R). The following section presents the most significant resource constraint propagation techniques applicable to each of these classes.

Table 1: Summary of problem definitions

Constraints common to all problems (fully elastic case)	$w(A, t) = 1 \Leftrightarrow t \in set(A)$ $start(A) = \min_{t \in set(A)}(t)$ $end(A) = \max_{t \in set(A)}(t + 1)$ $pt(A) = \lvert set(A) \rvert$ $w(A, R, t) = 0 \Leftrightarrow w(A, t) = 0$ $\Sigma_t\, w(A, R, t) = w(A, R)$ $\Sigma_A\, w(A, R, t) \leq C(R)$	
Additional constraints	Cumulative resource $C(R) \geq 1$	Disjunctive resource $C(R) = 1$
Preemptive	$w(A, R, t) = c(A, R) * w(A, t)$	$w(A, R, t) = w(A, t)$
Non preemptive	$w(A, R, t) = c(A, R) * w(A, t)$ $set(A) = [start(A), end(A))$	$w(A, R, t) = w(A, t)$ $set(A) = [start(A), end(A))$

Propagation of resource constraints

Each of the resource constraint propagation rules presented in this section is identified with a two-part name XX-∗, where the first part XX identifies the most general case in which the rule applies (FE for Fully Elastic, CP for Cumulative Preemptive, CNP for Cumulative Non-Preemptive, DP for Disjunctive Preemptive, and DNP for Disjunctive Non-Preemptive). The following diagram shows the relationships between these five cases. In this diagram, an arrow between two cases means that the techniques that apply to the first case can be applied to the second case (the corresponding deductions remain valid). To simplify the presentation, we assume that, for every activity A, the processing time pt(A) and the energy requirement w(A, R) are constrained to be strictly positive.

$$\begin{tikzcd}
FE \arrow[r] & CP \arrow[r] \arrow[d] & CNP \arrow[d] \\
 & DP \arrow[r] & DNP
\end{tikzcd}$$

The following notations are used:

EST_A	Earliest Start Time of activity A, *i.e.*, the smallest value in the domain of start(A).
LST_A	Latest Start Time of activity A, *i.e.*, the largest value in the domain of start(A).
EET_A	Earliest End Time of activity A, *i.e.*, the smallest value in the domain of end(A).
LET_A	Latest End Time of activity A, *i.e.*, the largest value in the domain of end(A).
p_A	Processing time of activity A, *i.e.*, the value of pt(A) when pt(A) is bound. When pt(A) is not bound, constraint propagation rules can usually be applied with p_A equal to the minimal value in the domain of pt(A).
c_A	Capacity (of the resource R under consideration) required by activity A, *i.e.*, the value of c(A, R) when c(A, R) is bound. When c(A, R) is not bound, constraint propagation rules can usually be applied with c_A equal to the minimal value in the domain of c(A, R).

w_A	Energy (of the resource R under consideration) required by activity A, *i.e.*, the value of w(A, R) when w(A, R) is bound. When w(A, R) is not bound, constraint propagation rules can usually be applied with w_A equal to the minimal value in the domain of w(A, R).
EST_Ω	Earliest start time of the set Ω. $EST_\Omega = \min_{A\in\Omega} EST_A$.
LET_Ω	Latest end time of the set Ω. $LET_\Omega = \max_{A\in\Omega} LET_A$.
p_Ω	Processing time of set Ω. $p_\Omega = \Sigma_{A\in\Omega}\, p_A$.
w_Ω	Energy required by set Ω. $w_\Omega = \Sigma_{A\in\Omega}\, w_A$.

<u>Constraint propagation based on timetables</u>

FE-TT, CP-TT, and CNP-TT

This is the most commonly used resource constraint propagation technique. It consists of (a) maintaining arc-B-consistency (Lhomme, 1993)[1] on the formula $\Sigma_A\, w(A, R, t) \le C(R)$ and (b) using the maximal values $w_{max}(A, R, t)$ of w(A, R, t) to compute the earliest and latest start and end times of activities (Le Pape, 1994) (Le Pape & Baptiste, 1996), (Le Pape & Baptiste, 1997):

$$\text{start}(A) \ge \min\{t \text{ such that } w_{max}(A, R, t) > 0\}$$
$$\text{end}(A) \ge \min\{t \text{ such that } \Sigma_{u<t}\, w_{max}(A, R, u) \ge w(A, R)\}$$
$$\text{start}(A) \le \max\{t \text{ such that } \Sigma_{u\ge t}\, w_{max}(A, R, u) \ge w(A, R)\}$$
$$\text{end}(A) \le \max\{t \text{ such that } w_{max}(A, R, u) > 0\} + 1$$

The FE, CP and CNP cases must be distinguished because the relations between w(A, R, t) and the start and end time variables differ.

- In the non-preemptive case, w(A, R, t) is equal to c(A, R) for every t in $[LST_A, EET_A)$.
- In the preemptive case, either the user or a search procedure must explicitly decide at which time points each activity executes. However, when start(A), pt(A), and end(A) are bound to values such that start(A) + pt(A) = end(A), w(A, R, t) is equal to c(A, R) for every t in [start(A), end(A)).
- In the fully elastic case, the relation is even weaker because, even when t is known to belong to set(A), w(A, R, t) can take any value between 1 and C(R).

In practice, the above rules are applied through the use of specialized algorithms, which can be made very efficient in the CNP case by relying on the fact that set(A) = [start(A), end(A)).

[1] Given a constraint c over n variables $v_1 \dots v_n$ and a domain D_i for each variable v_i, c is "arc-consistent" if and only if for any variable v_i and any value val_i in the domain of v_i, there exist values $val_1 \dots val_{i-1}\ val_{i+1} \dots val_n$ in $D_1 \dots D_{i-1}\ D_{i+1} \dots D_n$ such that $c(val_1 \dots val_n)$ holds. Arc-B-consistency, where B stands for bounds, guarantees only that $val_1 \dots val_{i-1}\ val_{i+1} \dots val_n$ exist for val_i equal to either the smallest or the greatest value in D_i.

Disjunctive constraints

Disjunctive constraints deal with cases in which two activities cannot overlap in time. Such a situation is of course common in disjunctive scheduling, but also occurs in cumulative scheduling, when the sum of the capacities required by two activities exceeds the capacity of the resource. Let us note that the following rules can *a priori* be generalized to triples, quadruples, *etc.*, of activities, but with a significant increase in the number of constraints and in the number of disjuncts per constraint. In practice, they are used only for pairs of activities.

CNP-DISJ: Disjunctive constraints in the non-preemptive case

Let A and B be two activities with $c_A + c_B > C(R)$. In the non-preemptive case, the disjunctive constraint propagation technique consists of maintaining arc-B-consistency on the formula:

$$[\text{end}(A) \leq \text{start}(B)] \text{ or } [\text{end}(B) \leq \text{start}(A)].$$

In the DNP case, all the pairs of activities that require the same resource R are related by such a disjunction. This is the origin of the term "disjunctive scheduling." Extensions of this rule are described in (Baptiste & Le Pape, 1996a), including the case in which c(A, R) and c(B, R) are not bound (and may satisfy $c(A, R) + c(B, R) > C(R)$) and the case in which the domain of pt(A) contains 0.

CP-DISJ: Disjunctive constraints in the preemptive case

Let A and B be two activities with $c_A + c_B > C(R)$. In the preemptive case, the disjunctive constraint propagation technique consists of maintaining arc-B-consistency on the formula:

$$\begin{aligned}
&[\text{start}(A) + \text{pt}(A) + \text{pt}(B) \leq \text{end}(A)]\\
&\text{or } [\text{start}(A) + \text{pt}(A) + \text{pt}(B) \leq \text{end}(B)]\\
&\text{or } [\text{start}(B) + \text{pt}(A) + \text{pt}(B) \leq \text{end}(A)]\\
&\text{or } [\text{start}(B) + \text{pt}(A) + \text{pt}(B) \leq \text{end}(B)].
\end{aligned}$$

We remark that, if only *some* activities can be interrupted, a "mixed" rule is obtained by removing the first (fourth) disjunct when A (respectively B) cannot be interrupted.

Edge-finding

"Edge-finding" constraint propagation techniques reason about the order in which activities execute on a given resource. In the DNP case, edge-finding consists of determining whether a given activity A must execute before (or after) a given set of activities Ω. 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 (earliest and latest start and end times). Cumulative and preemptive cases are more complex since several activities can overlap (on a cumulative resource) or preempt one another. Then edge-finding consists of determining whether an activity A must start or end before (or after) a set of activities Ω (adding edges in the graph representing the possible orderings of start and end times of activities).

DNP-EF: Edge-finding in the disjunctive preemptive case

Let A « B (A » B) mean that A is before (after) B and A « Ω (A » Ω) mean that A is before (after) all the activities in Ω. In the disjunctive non-preemptive case, the edge-finding technique consists in applying the following rules, and their symmetric counterparts:

$$A \notin \Omega \text{ and } \mathrm{LET}_{\Omega} - \mathrm{EST}_{\Omega \cup \{A\}} < p_{\Omega \cup \{A\}} \Rightarrow A \gg \Omega$$
$$A \gg \Omega \Rightarrow \text{start}(A) \geq \max_{\emptyset \neq \Omega' \subseteq \Omega}(\mathrm{EST}_{\Omega'} + p_{\Omega'})$$

Carlier and Pinson (1994) have shown that all the corresponding deductions can be done in O(n ∗ log(n)), where n is the number of activities requiring the resource.[2]

DP-EF: Edge-finding in the disjunctive preemptive case

In the preemptive case, the edge-finding rules no longer order activities but start and end times of activities. If A ⟩⟩ Ω means "A ends after all activities in Ω" then the following rules (and their symmetric counterparts) are obtained:

$$A \notin \Omega \text{ and } \mathrm{LET}_{\Omega} - \mathrm{EST}_{\Omega \cup \{A\}} < p_{\Omega \cup \{A\}} \Rightarrow A \rangle\rangle \Omega$$
$$A \rangle\rangle \Omega \Rightarrow \text{end}(A) \geq \max_{\Omega' \subseteq \Omega}(\mathrm{EST}_{\Omega' \cup \{A\}} + p_{\Omega' \cup \{A\}})$$
$$A \rangle\rangle \Omega \text{ and } (\mathrm{LET}_{\Omega} - \mathrm{EST}_{\Omega} = p_{\Omega}) \text{ and } (\mathrm{EST}_{\Omega} \leq \mathrm{EST}_{A}) \Rightarrow \text{start}(A) \geq \mathrm{LET}_{\Omega}$$

In the "mixed" case, the DNP-EF rules can be applied whenever A is not interruptible, even if the activities in Ω are interruptible (Le Pape & Baptiste, 1996). On the other hand, if A is interruptible, the DNP-EF rules are not valid and the (weaker) DP-EF rules must

[2] O(n ∗ log(n)) corresponds to applying the rules once to each pair (A, Ω). In constraint programming, one applies these rules (and other rules, used to propagate other constraints) until the domains of all the problem variables become stable. In the worst case, this can lead to executing the edge-finding algorithm O(Dn) times where D is a bound on the size of the domains of the variables involved in the propagation. In most cases, however, this does not occur, *i.e.*, each propagation algorithm is applied only a couple of times.

be applied. It is shown in (Le Pape & Baptiste, 1996) that the corresponding deductions can be done in $O(n^2)$.

FE-EF: Edge-finding in the fully elastic case

The following rules (and their symmetric counterparts) can be used in the fully elastic case.

$$A \notin \Omega \text{ and } C(R) * (\text{LET}_\Omega - \text{EST}_{\Omega \cup \{A\}}) < w_{\Omega \cup \{A\}} \Rightarrow A \rangle\rangle \Omega$$

$$A \rangle\rangle \Omega \Rightarrow \text{end}(A) \geq \max_{\Omega' \subseteq \Omega}(C(R) * \text{EST}_{\Omega' \cup \{A\}} + w_{\Omega' \cup \{A\}}) / C(R)$$

$$A \rangle\rangle \Omega \text{ and } (C(R) * (\text{LET}_\Omega - \text{EST}_\Omega) = w_\Omega) \text{ and } (\text{EST}_\Omega \leq \text{EST}_A) \Rightarrow \text{start}(A) \geq \text{LET}_\Omega$$

It is shown in (Baptiste et al., 1999) that the corresponding deductions can be done in $O(n^2)$. Note that when applied in the CP and in the CNP cases, these rules are sensitive to the use of fake activities to represent intervals during which the resource capacity is c with $0 < c < C(R)$ (because the fake activities are considered as fully elastic).

CNP-EF1: Edge-finding in the cumulative non-preemptive case

Let $\phi(\Omega')$ be true when $w_{\Omega'} > (C(R) - c_A) * (\text{LET}_{\Omega'} - \text{EST}_{\Omega'})$. The following rules (and their symmetric counterparts) summarize Section 4.4.1 of (Nuijten, 1994).

$$A \notin \Omega \text{ and } C(R) * (\text{LET}_\Omega - \text{EST}_{\Omega \cup \{A\}}) < w_{\Omega \cup \{A\}} \Rightarrow A \rangle\rangle \Omega$$

$$A \rangle\rangle \Omega \text{ and } \emptyset \neq \Omega' \subseteq \Omega \text{ and } \phi(\Omega')$$
$$\Rightarrow \text{start}(A) \geq \text{LET}_{\Omega'} - (C(R) * (\text{LET}_{\Omega'} - \text{EST}_{\Omega'}) - w_{\Omega'}) / c_A.$$

According to (Nuijten & Aarts, 1996), the corresponding deductions can be done in $O(n^2)$.

CNP-EF2: Edge-finding in the cumulative non-preemptive case

Let $\phi(\Omega')$ be true when $w_{\Omega'} > (C(R) - c_A) * (\text{LET}_{\Omega'} - \text{EST}_{\Omega'})$, as above. The following rules (and their symmetric counterparts) summarize Section 4.4.5 of (Nuijten, 1994).

$$A \notin \Omega \text{ and } \text{EST}_A \leq \text{EST}_\Omega < \text{EET}_A \text{ and}$$
$$C(R) * (\text{LET}_\Omega - \text{EST}_\Omega) < w_\Omega + c_A * (\text{EET}_A - \text{EST}_\Omega)$$
$$\Rightarrow A \rangle\rangle \Omega.$$

$$A \rangle\rangle \Omega \text{ and } \emptyset \neq \Omega' \subseteq \Omega \text{ and } \phi(\Omega')$$
$$\Rightarrow \text{start}(A) \geq \text{LET}_{\Omega'} - (C(R) * (\text{LET}_{\Omega'} - \text{EST}_{\Omega'}) - w_{\Omega'}) / c_A.$$

According to (Nuijten & Aarts, 1996), the corresponding deductions can be made in $O(n^3)$ for every pair (A, Ω) with $\text{EST}_A < \text{EST}_\Omega$.

“Not-first” and “not-last” rules

“Not-first” and “not-last” rules have been developed as a “negative” counterpart to edge-finding rules. They deduce that an activity A cannot be the first (or the last) to execute in $\Omega \cup \{A\}$.

DNP-NFNL: Disjunctive “not-first” and “not-last”
The following rule (and its symmetric counterpart) can be applied in the disjunctive non-preemptive case:

$$A \notin \Omega \text{ and } \mathrm{LET}_\Omega - \mathrm{EST}_A < p_{\Omega \cup \{A\}} \Rightarrow \mathrm{start}(A) \geq \min_{B \in \Omega}(\mathrm{EET}_B)$$

It is shown in (Baptiste & Le Pape, 1996b) that the corresponding deductions can be made in $O(n^2)$.

CNP-NFNL: Cumulative “not-first” and “not-last” rules
The following rule (and its symmetric counterpart) summarizes Section 4.4.3 of (Nuijten, 1994).

$$\begin{aligned} & A \notin \Omega \text{ and } \mathrm{EST}_\Omega \leq \mathrm{EST}_A < \min_{B \in \Omega}(\mathrm{EET}_B) \text{ and} \\ & C(R) * (\mathrm{LET}_\Omega - \mathrm{EST}_\Omega) < w_\Omega + c_A * (\min(\mathrm{LET}_\Omega, \mathrm{EET}_A) - \mathrm{EST}_\Omega) \\ & \qquad \Rightarrow \mathrm{start}(A) \geq \min_{B \in \Omega}(\mathrm{EET}_B) \end{aligned}$$

According to (Nuijten & Aarts, 1996), the corresponding deductions can be made in $O(n^3)$ for every pair (A, Ω) with $\mathrm{EST}_\Omega < \mathrm{EST}_A$.

Energetic reasoning

The edge-finding and the “not-first” and “not-last” rules are such that the activities that do not belong to Ω do not contribute to the analysis of the resource usage between $\mathrm{EST}_{\Omega \cup \{A\}}$ and $\mathrm{LET}_{\Omega \cup \{A\}}$. On the contrary, energetic reasoning rules compute the minimal contribution $W(B, t_1, t_2)$ of each activity B to a given interval $[t_1, t_2)$. In the preemptive case, the required energy consumption of B over $[t_1, t_2)$ can be evaluated as follows:

$$W_{PE}(B, t_1, t_2) = c_B * \max(0, p_B - \max(0, t_1 - \mathrm{EST}_B) - \max(0, \mathrm{LET}_B - t_2))$$

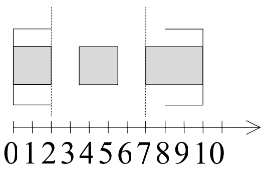


Figure 1: The required energy consumption of an activity B (earliest start time 0, latest end time 10, processing time 7 and resource requirement 2) over [2, 7).

Figure 1 illustrates this evaluation. The Gantt chart displays the fact that at least 2 time units of B have to be executed in [2, 7); which corresponds to $W_{PE}(B, 2, 7) = 2 * (7 - (2 - 0) - (10 - 7)) = 4$. The notation W_{PE} introduced in (Baptiste et al., 1999) refers to the fact that this value corresponds to a particular relaxation of the cumulative resource constraint, identified as the "Partially Elastic" relaxation.

In the non-preemptive case, a stronger value can be used:

$$W_{Sh}(B, t_1, t_2) = c_B * \min(t_2 - t_1, p_B^+(t_1), p_B^-(t_2))$$
with $p_B^+(t_1) = \max(0, p_B - \max(0, t_1 - \mathrm{EST}_B))$
and $p_B^-(t_2) = \max(0, p_B - \max(0, \mathrm{LET}_B - t_2))$.

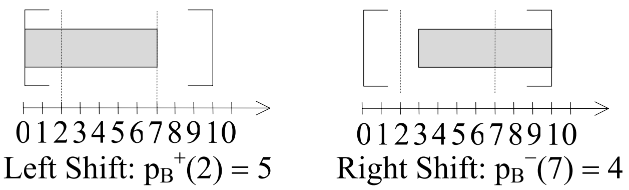


Figure 2: The required energy consumption of an activity B (earliest start time 0, latest end time 10, processing time 7 and resource requirement 2) over [2, 7). At least 4 time units of B must execute in [2, 7) which corresponds to $W_{Sh}(B, 2, 7) = 2 * \min(5, 5, 4) = 8$.

Figure 2 illustrates this evaluation. This time, the notation W_{Sh} refers to the fact that the best value is obtained by shifting the activity either to the left (*i.e.*, to its earliest possible execution interval) or to the right (to its latest possible execution interval). See (Erschler et al., 1991) (Lopez et al. 1992).

Many rules have been developed based on energetic reasoning. Only the most significant rules are presented below. We remark that these rules are in practice applied only to *some* intervals $[t_1, t_2)$, chosen with respect to the earliest and latest start and end times of activities (with no full formal justification, except for complexity reasons).

CP-ER: Energetic reasoning in the preemptive case

Let $\Delta^+(A, t_1, t_2) = \Sigma_{B \neq A} W_{PE}(B, t_1, t_2) + c_A * p_A^+(t_1) - C(R) * (t_2 - t_1)$ and $\Delta^-(A, t_1, t_2) = \Sigma_{B \neq A} W_{PE}(B, t_1, t_2) + c_A * p_A^-(t_2) - C(R) * (t_2 - t_1)$. The following rules can be applied:

$\Sigma_B W_{PE}(B, t_1, t_2) - C(R) * (t_2 - t_1) > 0 \Rightarrow$ inconsistency
$\Delta^+(A, t_1, t_2) > 0 \Rightarrow \mathrm{end}(A) \geq t_2 + \Delta^+(A, t_1, t_2) / c_A$.
$t_1 \leq \mathrm{EST}_A$ and $\Sigma_{B \neq A} W_{PE}(B, t_1, t_2) = C(R) * (t_2 - t_1) \Rightarrow \mathrm{start}(A) \geq t_2$.
$\Delta^-(A, t_1, t_2) > 0 \Rightarrow \mathrm{start}(A) \leq t_1 - \Delta^-(A, t_1, t_2) / c_A$.
$\mathrm{LET}_A \leq t_2$ and $\Sigma_{B \neq A} W_{PE}(B, t_1, t_2) = C(R) * (t_2 - t_1) \Rightarrow \mathrm{end}(A) \leq t_1$.

CNP-ER: Energetic reasoning in the non-preemptive case
Let $\Delta^+(A, t_1, t_2) = \Sigma_{B \neq A} W_{Sh}(B, t_1, t_2) + c_A * p_A^+(t_1) - C(R) * (t_2 - t_1)$ and $\Delta^-(A, t_1, t_2) = \Sigma_{B \neq A} W_{Sh}(B, t_1, t_2) + c_A * p_A^-(t_2) - C(R) * (t_2 - t_1)$. The following rules can be applied:

$\Sigma_B W_{Sh}(B, t_1, t_2) - C(R) * (t_2 - t_1) > 0 \Rightarrow$ inconsistency
$\Delta^+(A, t_1, t_2) > 0 \Rightarrow \text{end}(A) \geq t_2 + \Delta^+(A, t_1, t_2) / c_A$.
$t_1 \leq \text{EST}_A$ and $\Sigma_{B \neq A} W_{Sh}(B, t_1, t_2) = C(R) * (t_2 - t_1) \Rightarrow \text{start}(A) \geq t_2$.
$\Delta^-(A, t_1, t_2) > 0 \Rightarrow \text{start}(A) \leq t_1 - \Delta^-(A, t_1, t_2) / c_A$.
$\text{LET}_A \leq t_2$ and $\Sigma_{B \neq A} W_{Sh}(B, t_1, t_2) = C(R) * (t_2 - t_1) \Rightarrow \text{end}(A) \leq t_1$.

CNP-ER-DISJ: Energetic reasoning in the non-preemptive case
If $t_1 = \text{EST}_A$ and $t_2 = \text{LET}_B$ and $t_1 < t_2$ and $c_A + c_B > C(R)$ and $\Sigma_{C \neq A \,\&\, C \neq B} W_{Sh}(C, t_1, t_2) + c_A * p_A + c_B * p_B > C(R) * (t_2 - t_1)$, then $\text{end}(B) \leq \text{start}(A)$.

<u>Comparison</u>

The rules above differ from many perspectives.

- Their deductive power, i.e., the sets of values their application removes from the domains of variables, are different. For example, in the fully elastic case FE-EF strictly dominates FE-TT, but in the non-preemptive disjunctive case CNP-TT is not dominated by the edge finding rules. An extensive comparison of these rules in terms of deductive power can be found in (Baptiste et al., 2001).
- The complexity and incrementality of the algorithms that have been developed to apply these rules differ a lot. As a result, a rule that performs well on small instances might just take too long when the problem size becomes bigger.
- Some of these rules are not monotonic. FE-TT, CP-TT, CNP-TT, CP-DISJ, CNP-DISJ, FE-EF, DP-EF, DNP-EF, CNP-EF1, DNP-NFNL, CP-ER and CNP-ER have monotonic conditions, while CNP-EF2, CNP-NFNL and CNP-ER-DISJ have non-monotonic conditions. This means that better time-bounds for some activities may prevent the application of the CNP-EF2, CNP-NFNL or CNP-ER-DISJ, and hence the drawing of the corresponding conclusion. The main drawback of non-monotonic propagation rules is that they make debugging and performance tuning much more difficult.

A consequence of these differences is that the set of propagation rules to be used in a given practical application must be chosen with respect to the characteristics of this application.

An Example of Heuristic Search: Preemptive Job-Shop Scheduling

Scheduling problems are NP-hard. Even with the best constraint propagation techniques, it is in general impossible to aim for the optimal solution of a practical problem. The goal then is to obtain solutions as good as possible in a given "allowable" amount of computational time. Only very general principles apply in this domain:

- Avoid searching parts of the search space where no interesting solution lies. As already mentioned, constraint propagation can often be complemented with dominance rules to further reduce the search space.
- Intensify search around good solutions, in search of a better solution.
- Diversify search, i.e., do not leave significant promising regions of the search space unexplored.

In the following, we choose the preemptive job-shop scheduling problem (PJSSP) to illustrate these principles. 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 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 constraint programming, minimizing a given objective (here, the makespan) is generally done by solving the decision variant of the problem (Garey & Johnson, 1979) with different bounds imposed on the objective. 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 the additional constraint 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 very large. Indeed, each set(A) variable a priori accepts up to $(v * (v - 1) * ... * (v - pt(A) + 1)) / (1 * 2 * ... * pt(A))$ 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 (proven in (Baptiste & Le Pape, 1999)) 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.

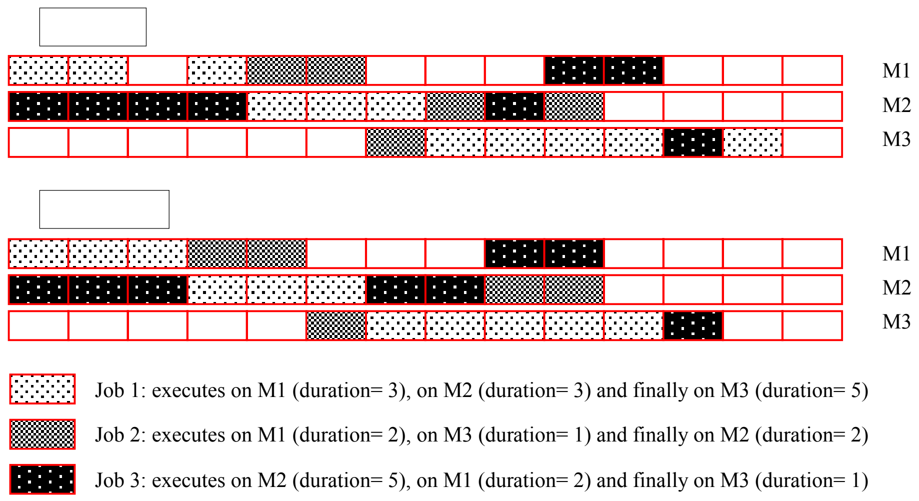
THEOREM 1

Let acts(M) denote the set of activities to be processed on machine M. 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 $A_k <_S A_l$.

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: A schedule *S* and its "Jackson derivation" *J*(*S*).


M1
M2
M3
M1
M2
M3
Job 1: executes on M1 (duration= 3), on M2 (duration= 3) and finally on M3 (duration= 5)
Job 2: executes on M1 (duration= 2), on M3 (duration= 1) and finally on M2 (duration= 2)
Job 3: executes on M2 (duration= 5), on M1 (duration= 2) and finally on M3 (duration= 1)

A branching scheme for the preemptive job-shop scheduling problem

The dominance criterion leads to the following branching scheme:

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: $\text{end}(A_k) + \text{rp}_t(A_l) \le \text{end}(A_l)$, where $\text{rp}_t(A_l)$ is the remaining processing time of A_l at time t; $\text{end}(A_k) \le \text{start}(A_l)$ 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 $\le$ v has been found.

Constraint propagation for the preemptive job-shop scheduling problem

Three constraint propagation techniques can be considered: timetables (CP-TT), disjunctive constraints (CP-DISJ) and edge-finding (DP-EF). In fact, we know from (Baptiste et al., 2001) that DP-EF dominates all other rules in terms of deductive power but it is also more time consuming than CP-TT. CP-DISJ does not deduce much even though it is not dominated by CP-TT. In the end, two alternatives, CP-TT and DP-EF are worth considering.[3]

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. 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.

[3] Note that we could consider applying CP-TT to some resources and DP-EF to others.

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 (10 jobs 10 machines) 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 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.

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)

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). On the PJSSP, LDS appeared to work better than DDS.

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 3 and 4 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)

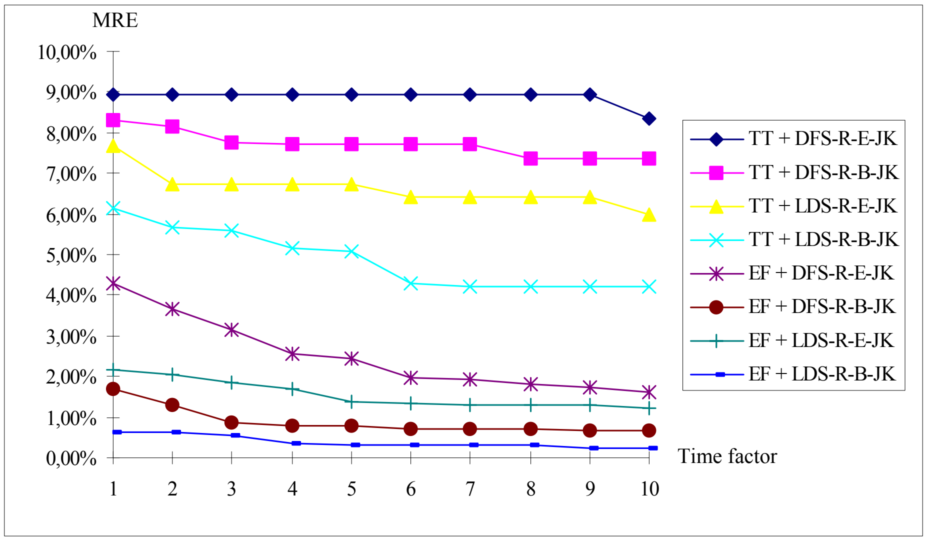


Figure 3: Results on the ten instances used in (Applegate & Cook, 1991)

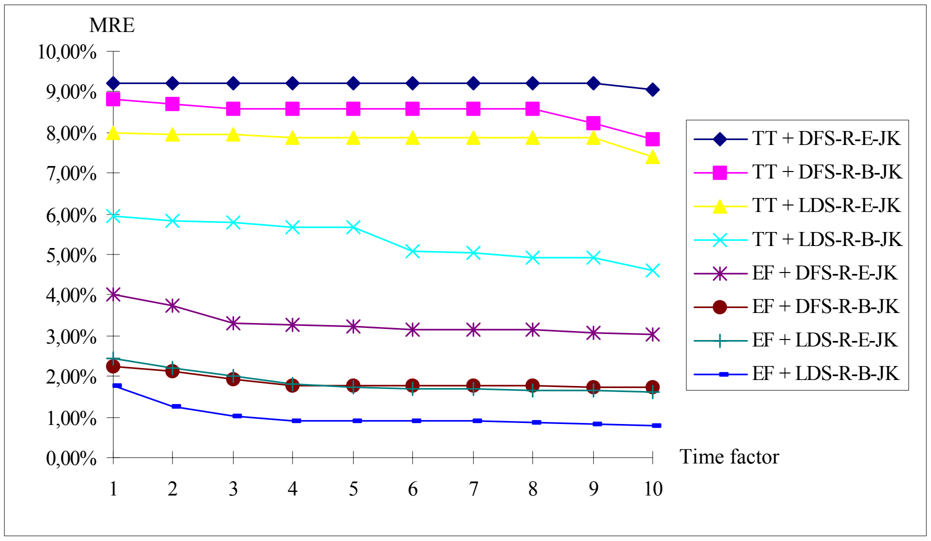


Figure 4: Results on the thirteen instances used in (Vaessens et al., 1994)

From the preemptive job-shop scheduling problem to a practical application

Practical problems are never as pure as the preemptive job-shop scheduling problem. Disjunctive and cumulative resources coexist, interruptible and non-interruptible activities coexist, specific constraints and preferences must often be added. An example of extension of the preemptive job-shop scheduling problem is the daily construction site scheduling problem. A typical instance includes 3 to 5 resources, one disjunctive (the crane) and the others cumulative (teams of people), 15 interruptible activities, 15 activities subjected to time-versus-capacity tradeoffs, and 10 activities for which the required capacity can vary over time. About 70 additional constraints (temporal constraints, synchronization constraints) apply. Half of these constraints are preferences, with different levels of importance, which are more or less conflicting depending on the problem data. The search algorithm implemented to solve this problem attempts to satisfy as many preferences as possible, starting from the most important ones. This is implemented as a "shuffle" of the preferences: at each iteration, a set of preferences is selected and the system tries to find a solution which satisfies all the constraints of the set. Eight search strategies are applied in turn and a limited number of backtracks allowed for each strategy. If the search succeeds, the system further biases the shuffle towards larger sets of constraints. If the search fails, the system biases the shuffle towards smaller sets of constraints. On most instances, the system satisfies all the constraints but two or three in 10 to 15 iterations of 15 seconds each, which compares favorably with the manual solutions we have seen, which typically violate eight or nine constraints.

Conclusion

The principles of constraint programming have been widely applied to scheduling problems, enabling the implementation of flexible and extensible scheduling systems. With constraint programming all the specific constraints of a given problem can be represented and actually used as a guide toward a solution. This “flexibility” can be contrasted with the attention that has been paid in operations research to rather “pure” scheduling problems, based on relatively simple mathematical models, the combinatorial structure of which can be exploited to improve performance. We could say that a traditional operations research approach often aims at achieving a high level of “efficiency” in its algorithms, at the expense of the “generality of application” of these algorithms.

As the number of applications grew, the need emerged to reconcile the flexibility offered by constraint programming with the efficiency of specialized operations research algorithms. The first step consisted in adapting well-known operations research algorithms to the constraint programming framework, mostly by incorporating these algorithms within constraint propagation. As a second step, the success of the resulting tools opened a new area of research aimed at the design and implementation of efficient algorithms embeddable in constraint programming applications and tools. This includes the design of efficient constraint propagation techniques for specific optimization criteria, the application of linear programming to well-behaved subproblems, and the combination of constraint programming with various forms of local search to generate “good” schedules in a limited amount of time.

Resource constraints and optimization criteria

Propagating the objective constraint (that defines the optimization criterion) and the resource constraints independently is not a problem when the optimization criterion is a “maximum” such as the makespan or the maximal tardiness of a given set of jobs. Indeed, an upper bound on the optimization criterion is directly propagated on the completion time of the jobs under consideration, i.e., the latest end times of these jobs are tightened efficiently.

The situation is much more complex for “sum” functions such as the weighted number of late jobs or the sum of setup times between jobs. For such objective functions, efficient constraint propagation techniques must take into account the resource constraints and the objective constraint simultaneously.

Excellent results with new approaches combining constraint programming with deductive algorithms targeted toward specific objective functions appear in (Baptiste et al., 1998) for the weighted number of late jobs and (Focacci et al., 2000} for the sum of setup times. However, many other objective functions (e.g., total tardiness, total flow-time) still have to be studied. An important research challenge is to design generic lower-bounding techniques and constraint propagation algorithms that could work for many criteria.

Linear programming and constraint programming

Three areas in which the integration of linear programming and constraint programming is promising are identified:

- For cumulative scheduling problems, the complexity of specific constraint propagation algorithms tends to raise (e.g., to $O(n^3)$), which suggests that lower-bounding and constraint propagation algorithms based on linear programming might be competitive. For example, several lower bounds based on linear programming formulations of a relaxed resource-constrained project scheduling problem have shown to be very accurate (Bruckner & Knust, 2000) (Mingozzi et al., 1998). Unfortunatley, the size of the linear models used is large, so even with complex column generation techniques hours of CPU time are sometimes required to get a lower bound. Recently, Carlier and Néron (2001) have shown that, for each resource, an efficient lower bound based on linear programming can be tabulated, for each fixed value of the resource capacity. Then the computation of the lower bounds requires much less CPU at run time. This is probably one of the most promising areas of research for the next few years.
- Linear programming can also be a strong "ingredient" when the objective function is a sum or a weighted sum of scheduling variables like the end times of activities. A key research issue here is the design of techniques combining the power of efficient constraint propagation algorithms for the resource constraints and the power of linear programming for bounding the objective function. In some simple cases, mixed integer programming can also be used to improve solutions found by constraint programming (cf., for example (Danna, 2004)).
- In real-life applications, scheduling issues are often mixed with resource allocation, capacity planning, or inventory management issues for which mixed integer programming is a method of choice. Several examples have been reported where a hybrid combination of constraint programming and mixed integer programming was shown to be more efficient than pure constraint programming or mixed integer programming models (cf., for example, (El Sakkout & Wallace, 2000)). The generalization of these examples into a principled approach is another important research issue for the forthcoming years.

Local search and constraint programming

Various forms of local search, such as simulated annealing, tabu search, genetic algorithms, etc., provide excellent results when one can define a compact representation of the solution space that is consistent with the objective function. In real life, problems often incorporate side constraints that tend to disable the local search approach. This led several researchers to integrate constraint programming and local search techniques. For example, Caseau and Laburthe (1995) describe an algorithm for the job-shop scheduling problem which combines constraint programming and local search. The overall algorithm finds an approximate solution to start with, makes local changes and repairs on it to quickly decrease the makespan and, finally, performs an exhaustive search for decreasing makespans. Experiments in combining branch and bound search with genetic algorithms (see, for example, (Portmann et al., 1998)) suggest that constraint programming algorithms could also be combined with algorithms optimizing populations of solutions, in particular when these algorithms can be adapted to respect constraints imposed at a given node of a branch and bound tree or deduced through constraint propagation.

Globally, the integration of local search and constraint programming is promising whenever local search operators provide a good basis for the exploration of the search space and either side constraints or effective constraint propagation algorithms can be used to prune the search space. The examples presented in the literature represent a significant step toward the understanding of the possible combinations of local search and constraint programming. Yet the definition of a general approach and methodology for integrating local search and constraint programming remains an important area of research.

References

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," Proceedings 14th International Joint Conference on Artificial Intelligence, 600-606, Morgan Kaufmann.

Baptiste, Ph. & Le Pape, C. (1996a). "Disjunctive Constraints for Manufacturing Scheduling: Principles and Extensions," International Journal of Computer Integrated Manufacturing, 9, 306-310.

Baptiste, Ph. & Le Pape, C. (1996b). "Edge-Finding Constraint Propagation Algorithms for Disjunctive and Cumulative Scheduling," Proceedings 15th Workshop of the U.K. Planning Special Interest Group.

Baptiste, Ph., Le Pape, C., & Péridy, L. (1998). "Global Constraints for Partial CSPs: A Case Study of Resource and Due-Date Constraints," Proceedings 4th International Conference on Principles and Practice of Constraint Programming.

Baptiste, Ph., Le Pape, C., & Nuijten, W. (1999). "Satisfiability Tests and Time-Bound Adjustments for Cumulative Scheduling Problems," Annals of Operations Research 92, 305-333.

Baptiste, Ph., Le Pape, C., & Nuijten, W. (2001). "Constraint-Based Scheduling," Kluwer Academic Publishers, 2001.

Billaut, J.-C. (1993). "Prise en compte des ressources multiples et des temps de préparation dans les problèmes d'ordonnancement en temps réel," PhD Thesis, University Paul Sabatier (in French).

Brucker, P. & Knust, S. (2000). "A Linear Programming and Constraint Propagation-Based Lower Bound for the RCPSP," European Journal of Operational Research 127, 355-362.

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.

Carlier, J. & Néron, E. (2001). “A New LP Based Lower Bound for the Cumulative Scheduling Problem,” Research Report, University of Technology of Compiègne.

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

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

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.

Collinot, A. & Le Pape, C. (1991). “Adapting the Behavior of a Job-Shop Scheduling System,” Decision Support Systems 7.

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

Danna, E. (2004). “Intégration des techniques de recherche locale à la programmation linéaire en nombre entiers,” PhD Thesis, Université d’Avignon (in French).

El Sakkout, H., & Wallace, M. (2000). “Probe Backtrack Search for Minimal Perturbation in Dynamic Scheduling,” Constraints 5, 359-388.

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).

Focacci, F., Laborie, Ph., & Nuijten, W. (2000). “Solving Scheduling Problems with Setup Times and Alternative Resources,” Proc. 5th International Conference on Artificial Intelligence Planning and Scheduling.

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

Ginsberg, M. L. (1993). “Dynamic Backtracking,” Journal of Artificial Intelligence Research 1.

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.

Korf, R. E. (1996). “Improved Limited Discrepancy Search,” Proc. 13th National Conference on Artificial Intelligence.

Kumar, V. (1992). “Algorithms for Constraint Satisfaction Problems: A Survey,” AI Magazine 13, 32-44.

Latombe, J.-C. (1979). “Failure Processing in a System for Designing Complex Assemblies,” Proceedings 6th International Joint Conference on Artificial Intelligence.

Le Gall, A. (1989). “Un système interactif d’aide à la décision pour l’ordonnancement et le pilotage en temps réel d’atelier,” PhD Thesis, University Paul Sabatier (in French).

Le Gall, A. & and Roubellat, F. (1992). “Caractérisation d’un ensemble d’ordonnancements avec contraintes de ressources de type cumulatif,” RAIRO Automatique, Productique et Informatique Industrielle 26 (in French).

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

Le Pape, C. (1992). “Using Constraint Propagation in Blackboard Systems: A Flexible Software Architecture for Reactive and Distributed Systems,” IEEE Computer 25.

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. (1995). “Three Mechanisms for Managing Resource Constraints in a Library for Constraint-Based Scheduling,” Proceedings INRIA/IEEE Conference on Emerging Technologies and Factory Automation.

Le Pape, C. & Baptiste, Ph. (1996). “Constraint Propagation Techniques for Disjunctive Scheduling: The Preemptive Case,” Proceedings 12th European Conference on Artificial Intelligence.

Le Pape, C. & Baptiste, Ph. (1997). “A Constraint Programming Library for Preemptive and Non-Preemptive Scheduling,” Proceedings 3rd International Conference and Exhibition on the Practical Application of Constraint Technology.

Le Pape, C. & Baptiste, Ph. (1998). “Resource constraints for preemptive job-shop scheduling,” Constraints 3, 263-287.

Le Pape, C. & Baptiste, Ph. (1999). “Heuristic Control of a Constraint-Based Algorithm for the Preemptive Job-Shop Scheduling Problem,” Journal of Heuristics 5, 305-325.

Lesaint, D. (1993). “Specific Sets of Solutions for Constraint Satisfaction Problems,” Proceedings 13th International Workshop on Expert Systems and Applications.

Lhomme, O. (1993). “Consistency Techniques for Numeric CSPs,” Proceedings 13th International Joint Conference on Artificial Intelligence.

Lopez, P., Erschler, J., & Esquirol, P. (1992). “Ordonnancement de tâches sous contraintes : une approche énergétique,” RAIRO APII 26, 453-481 (in French).

Mingozzi, A., Maniezzo, V., Ricciardelli, S., & Bianco, L. (1998). “An exact algorithm for project scheduling with resource constraints based on a new mathematical formulation,” Management Science 44, 714-729.

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.

Portmann, M.-C., Vignier, A., Dardilhac, D., & Dezalay, D. (1998). “Branch and Bound crossed with GA to solve Hybrid Flowshops,” European Journal of Operational Research 107, 389-400.

Prosser, P. (1993). “Hybrid Algorithms for the Constraint Satisfaction Problem,” Computational Intelligence 9.

Puget, J.-F. & Leconte, M. (1995). “Beyond the Glass Box: Constraints as Objects,” Proceedings International Symposium on Logic Programming.

Stallman, R. M. & Sussman, G. J. (1977). “Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis,” Artificial Intelligence 9.

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.

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

Xiong, Y., Sadeh, N., & Sycara, K. (1992). “Intelligent Backtracking Techniques for Job-Shop Scheduling,” Proceedings 3rd International Conference on Principles of Knowledge Representation and Reasoning.