

Beyond Cmax: an optimization-oriented framework for constraint-based scheduling

Arnaud Malapert, Sophie Demassey, Jean-Charles Régin

▶ **To cite this version:**

Arnaud Malapert, Sophie Demassey, Jean-Charles Régin. Beyond Cmax: an optimization-oriented framework for constraint-based scheduling. 2012. hal-00976994

HAL Id: hal-00976994
https://hal.science/hal-00976994

Submitted on 10 Apr 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.





LABORATOIRE



INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR7271

Beyond *Cmax*: an optimization-oriented framework for constraint-based scheduling

Arnaud Malapert, Sophie Demassey, Jean-Charles Régin

Équipe CeP

Rapport de Recherche
ISRN I3S/RR-2012-07-FR

Avril 2012

Laboratoire d'Informatique, Signaux et Systèmes de Sophia-Antipolis (I3S) - UMR7271 - UNS CNRS
2000, route des Lucioles - Les Algorithmes - bât. Euclide B 06900 Sophia Antipolis - France
http://www.i3s.unice.fr

Beyond *Cmax*: an optimization-oriented framework for constraint-based scheduling

Arnaud Malapert [1] , Sophie Demassey [2], Jean-Charles Régin [1]

Équipe CeP

ISRN I3S/RR-2012-07-FR
Avril 2012 - 10 pages

Abstract: This paper presents a framework taking advantage of both the flexibility of constraint programming and the efficiency of operations research algorithms for solving scheduling problems under various objectives and constraints. Built upon a constraint programming engine, the framework allows the use of scheduling global constraints, and it offers, in addition, a modular and simplified way to perform optimality reasoning based on well-known scheduling relaxations. We present a first instantiation on the single machine problem with release dates and lateness minimization. Beyond the simplicity of use, the optimizationoriented framework appears to be, from the experiments, effective for dealing with such a pure problem even without any ad-hoc heuristics.

Key-words: Combinatorial optimization; Artificial intelligence; Constraints satisfaction; Scheduling

[1] Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France – arnaud.malapert@unice.fr, jean-charles.REGIN@unice.fr
[2] TASC project, Mines Nantes-INRIA, LINA CNRS UMR 6241 – sophie.demassey@mines-nantes.fr

Beyond C_{max}: an optimization-oriented framework for constraint-based scheduling

Arnaud Malapert[1], Sophie Demassey[2], and Jean-Charles Régin[1]

[1] I3S CNRS UMR 7271, UNSA
[2] TASC project, Mines Nantes-INRIA, LINA CNRS UMR 6241,
{firstname.lastname@{unice,mines-nantes}.fr

Abstract. This paper presents a framework taking advantage of both the flexibility of constraint programming and the efficiency of operations research algorithms for solving scheduling problems under various objectives and constraints. Built upon a constraint programming engine, the framework allows the use of scheduling global constraints, and it offers, in addition, a modular and simplified way to perform optimality reasoning based on well-known scheduling relaxations. We present a first instantiation on the single machine problem with release dates and lateness minimization. Beyond the simplicity of use, the optimization-oriented framework appears to be, from the experiments, effective for dealing with such a pure problem even without any ad-hoc heuristics.

All scheduling problems are defined by specific combinations of resource environments, job characteristics, objectives and side constraints which make them all slightly different to solve and most of them highly combinatorial. A standard approach for solving a given practical scheduling problem is to adapt and combine different algorithms (relaxations, heuristics, search algorithms) that have been designed for simpler mathematical models. As the peculiar aspects of the problem are rarely compatible with these algorithms, they are simply ignored during the solution process to be heuristically repaired afterwards. This results in complex developments of a non-exact solution method dedicated to one problem.

As an alternative, a constraint programming (CP) engine lets the user specify the different components of his problem separately, then automatically solves the components jointly. The efficiency of the engine strongly depends on the management of each independent component. In the context of scheduling, attention has been paid on temporal networks [1] and, mainly, on resource conflicts given various filtering algorithms (edge-finding, not first/not last, etc) for `disjunctive` and `cumulative` constraints [2]. These algorithms are mostly effective when minimizing the makespan, *i.e.* the schedule duration. However, makespan minimization is not a realistic criterion in many contexts where other objective functions, like the maximum lateness, the total tardiness or the weighted sum of the completion times, are preferred in practice [3]. Such regular objectives may be set as additional constraints and variables in the model, but no effective propagation can be obtained without considering the resource constraints and the temporal

contraints at the same time. This is probably the main limitation of current CP systems to their wide application to practical scheduling [4].

Our goal is to design a generic constraint-based scheduling framework, combining the flexibility of CP for specifying complex scheduling problems with various objectives and constraints, with the efficiency of ad-hoc branch-and-bound (B&B) algorithms for solving the simplest models. These algorithms include accelerating features such as (for a minimization criterion): (*i*) relaxations computing tight lower bounds at each node, (*ii*) filtering techniques discarding either infeasible or sub-optimal decisions, (*iii*) branching strategies guided by the relaxed solutions and impacting on them from one node to its child, and (*iv*) heuristics computing upper bounds and feasible solutions, at the root node only or during the search. These features are not standard in CP-based B&B which are thus recognized to be weak for solving optimization problems in general. The concept of using relaxations for integrating features (*i*) to (*iii*) has been examined by Focacci et al. [5]. They show how to exploit relaxed optima in CP-B&B for node pruning, domain filtering (also called back-propagation), and search guiding. This approach has been fully applied to only few applications in the current literature but since, back-propagation algorithms have demonstrated their benefit when a convenient relaxation of a problem exists. In the context of scheduling, several back-propagation algorithms have been applied to specific problems with different optimality criteria, either for *sum objectives*, e.g. weighted number of late jobs [6], total set-up times [5], total tardiness [7], total earliness/tardiness [8, 9], total weighted completion time [10, 11], or for *bottleneck objectives*, e.g. maximum lateness [12].

The unifying framework we propose aims at taking advantage of the genericity of the full approach introduced by [5]. It is especially applicable to scheduling problems for several reasons. From the user perspective, there exist libraries of relaxations and algorithms (e.g. [3, 13, 14]) which could be queried either manually or automatically to be integrated in the framework. Furthermore, each relaxation remains valid for various problems with different resource and job characteristics. Finally, from the solver perspective, most of these algorithms, especially priority list algorithms, offer all the features required (e.g. low complexity and incrementality) to make the approach fully effective. To our knowledge, no generic implementation of [5] has been proposed so far. The Aeon system [15] provides a high-level modeling library and solvers for scheduling problems, but it does not integrate this approach in a systematic way. Our framework can be seen as a possible extension to this generic-purpose system.

Our first contribution, described in Section 1 of this paper, is a customizable, modular and compositional template offering optimality reasoning functions to any CP-B&B: the user specifies an algorithm for solving a relaxation of his core problem, then the system automatically derives altogether basic bounding, filtering (by probing) and search guiding (e.g. with regrets) techniques. The user can also easily adapt these default functions to make them more effective with his specific relaxation. To validate the framework, we present in Section 2 a first instantiation on the single machine problem with release dates and lateness min-

imization $(1|r_j|L_{max})$ (see e.g. [16]) by implementing two standard relaxations. This application confirms the simplicity of implementation offered by the framework, while the experimental results given in Section 3 show its effectiveness. In the future, we envisage to test models of other scheduling problems either based on the same relaxations or requiring to implement new ones for different objectives. Compared to ad-hoc solution methods for pure scheduling models, which are mostly based on powerful meta-heuristics, we will also conduct a study about the integration of heuristics in our generic framework.

1 The Optimization-oriented Framework

Relaxations in CP-based B&B. Pruning mostly derives from feasibility reasoning. When coping with an optimization problems with a B&B algorithm, pruning must also proceed from optimality reasoning: *Pruning* a node means to discard from the search a subset that contains no solution whose cost is better than the current incumbent, *i.e.* the value of the best solution found so far. This can be detected by solving a relaxation of the subproblem: the corresponding node is pruned if the optimum of the relaxation equals or exceeds the incumbent. *Cost-based filtering* allows to prevent such a thrash early, by removing variable-value assignments which do not lead to solutions better than the incumbent. A sub-optimal assignment is detected by estimating its extra-cost to the optimum when added to the relaxation and, again, by comparing it to the incumbent. *Cost-guided branching* aims at minimizing the size of the tree with strategies that select the next branching according to the relaxed optima and solutions. These are state-of-the-art techniques of integer linear programming solvers. In contrast, they have received few attention in CP, one notable exception being regret-based strategies. *Relaxation-based heuristics* can be run at each node of the search to derive a feasible solution from a relaxed solution. When an improving solution is found, the incumbent is updated.

To be effective, the features require certains informations that are not provided by all relaxations. Concerning heuristics, for instance, the solution of a linear relaxation can rarely be made feasible, *i.e.* integer, while it is often possible to recover feasibility from a lagrangian or a combinatorial relaxation, depending on the semantic of the relaxed constraints. A dual relaxation does not even return a relaxed solution, but just a bound. Concerning cost-based filtering, the default probing procedure, which consists in re-solving the relaxation after adding each assignment decision independently, can be employed with any relaxation. However, the procedure can be greatly improved when the relaxation provides the bound estimates for all the assignments at once, as a gradient function. This is the case of linear and lagrangian relaxations with reduced costs. In the context of CP, Focacci et al. [5] proposed to embed in a global constraint an optimization component representing a suitable relaxation of the constraint itself. The framework below is a customizable implementation of this approach.

Specifications of the Framework. The framework attends to solve any constraint optimization problem $\mathcal{P}$ for which at least one convenient relaxation $\mathcal{R}$ exists. W.l.o.g assume that $\mathcal{P}$ is a minimization problem $\min\{Z|Z = cost(X), X \in \mathcal{C}\}$ with $\mathcal{C}$ the constraints, X the decision variables, *cost* the objective function and Z a range variable representing the solution value. $\mathcal{R}$ can be stated as $\min_{(X\in\mathcal{C}')} cost(X)$ with $\mathcal{C}' \subseteq \mathcal{C}$. At each point of the resolution, the current domains $\mathcal{D}$ of variables X define an (optimization) instance of $\mathcal{R}$ denoted $\mathcal{R}(\mathcal{D})$: $\min_{(X\in\mathcal{D}\cap\mathcal{C}')} cost(X)$. With the upper bound $Z_{\max}$ of variable Z, they define a satisfaction instance of $\mathcal{P}$ denoted $\mathcal{P}(\mathcal{D}, Z_{\max})$: $s \in \mathcal{D}$ is a feasible solution of $\mathcal{P}(\mathcal{D}, Z_{\max})$ if and only if $s \in \mathcal{C}$ and $cost(s) \leq Z_{max}$. As in a standard B&B, each time a feasible solution of $\mathcal{P}(\mathcal{D}, Z_{\max})$ is reached, *i.e.* when $\mathcal{D}$ becomes a singleton $\{s\}$, it is stored as the new incumbent and the search goes on after the maximum value $Z_{\max}$ has been restricted to $cost(s) - 1$.

The framework is built upon a standard CP engine and made of pre-implemented generic components and of required and optional components the user has to provide to solve its own problem $\mathcal{P}$. First, the user adds to the model of $\mathcal{P}$ a constraint `relaxation`(Z, X) whose solutions are $(cost(s), s)$ for all feasible solution s of $\mathcal{P}(\mathcal{D}, Z_{\max})$. To ensure this, a propagator must be implemented upon a given abstract template. The minimum implementation requires the code of two primitives: `buildRelaxation`$(\mathcal{D})$ that builds the instance $\mathcal{R}(\mathcal{D})$ from any current domain $\mathcal{D}$ of X, and `bound`$(\mathcal{D})$ that returns the optimal value of this instance. The template is modular in order to be easily customized. The list of primitives, their specifications and default implementations follow:

`decisionList`$(\mathcal{D})$ returns a finite ordered list $(D_1, \ldots, D_p) \subseteq \mathcal{D}^p$ of decisions to probe. The default is the list of assignments from variables X to values in $\mathcal{D}$ in the lexicographic order.

`probeDecision`$(D_k, Z_{\max})$ builds instance $\mathcal{R}(D_k)$ then returns (`bound`$(D_k) > Z_{\max}$) as a boolean. By default, $\mathcal{R}(D_k)$ is built from scratch using `buildRelaxation`.

`pruneDecision`$(\mathcal{D}, D_k)$ returns a restricted domain $\mathcal{D}'$ with $\mathcal{D} \setminus D_k \subseteq \mathcal{D}' \subseteq \mathcal{D}$.

`filter`$(\mathcal{D}, Z_{\max})$ returns the sublist of `decisionList`$(\mathcal{D})$ of decisions D_k such that `probeDecision`$(D_k, Z_{\max})$ is true.

`propagate`$(\mathcal{D}, Z_{\max})$ notifies the propagation engine that: (i) $Z_{\min} := \max(Z_{\min}$,`bound`$(\mathcal{D}))$ then (ii), if no failure occurs (*i.e.* if `bound`$(\mathcal{D}) \leq Z_{\max}$), $\mathcal{D} :=$`pruneDecision`$(\mathcal{D}, D_k)$ for each decision $D_k \in \mathcal{D}$ returned by `filter`$(\mathcal{D}, Z_{\max})$.

Hence, in the default implementation of the propagator, all variable-value assignments are probed in turn by, each time, building and solving the relaxation from scratch. Depending on its knowledge of the relaxation, the user can greatly improve this algorithm by overloading one or several primitives independently. For instance, he may redefine `decisionList` in order to discard from probing some assignments that are known for not changing the relaxed optimum value. The way the decisions are probed in sequence is also crucial in terms of computational complexity. The relaxations can be built and solved, as follows, from the less to the more incremental fashion: (*i*) *by default*, for each decision D_k, build $\mathcal{R}(D_k)$ then solve it to get the optimum ; (*ii*) *by overloading* `probeDecision`, build $\mathcal{R}(\mathcal{D})$ then, for each decision D_k, update $\mathcal{R}(D_k)$ then solve it to get the

optimum ; (*iii*) *by overloading* `filter`, build $\mathcal{R}(\mathcal{D})$ then solve it to get the optimum and a gradient function then, for each decision D_k, get $opt + gradient(D_k)$ Furthermore, the optima of the probed relaxations can be exploited within the branching strategy for guiding the search. Hence, these values and the associated decisions, are stored during the computation of `filter`, to be queried by the search engine when choosing the next decision to branch, through different primitives such as `getBestCostDecision()` or `getBestRegretDecision()`.

Finally, most of the time, solving a relaxation $\mathcal{R}(\mathcal{D})$ returns the optimal value with an optimal solution. This solution can also be exploited for guiding the search (by assigning a variable to its value in the solution) or directly for pruning the search if the solution is identified as feasible for $\mathcal{P}$. Hence, when possible the user will be interested to define these additional functions: `relax`$(\mathcal{D})$ returns an optimal solution of $\mathcal{R}(\mathcal{D})$ as an instantiation $s \in \mathcal{D}$ of X if exists, and the empty set, otherwise (or by default); `isFeasible`$(\mathcal{D})$ returns true iff `relax`$(\mathcal{D})$ is a feasible solution of $\mathcal{P}$. The default just calls the solution checker of the propagation engine; `updateIncumbent`$(\mathcal{D}, Z_{\max})$ notifies the search engine of the new solution `relax`$(\mathcal{D})$ if `isFeasible`$(\mathcal{D})$ is true and its cost is lower than $Z_{\max}$ (if the cost is $Z_{\min}$ then a backtrack occurs).

2 Application to $1|r_j|L_{max}$

The scheduling model that we analyze is as follows. There are n independent jobs $J_1, \ldots, J_n$ that have to be scheduled without overlapping on a single machine. Each job J_j requires processing during a given uninterrupted time $p_j \geq 0$, not starting before its release date $r_j \geq 0$, and being ideally completed before a due date $d_j \geq 0$. Let C_j denote the completion time of a job J_j, the objective function to minimize is the maximum lateness defined as $L_{max} = \max_{1 \leq j \leq n}(C_j - d_j)$.

Basic Constraint Model. CP models of scheduling problems usually represent a non-preemptive job J_j (also called task) by a triplet of non-negative integer variables (s_j, p_j, e_j) denoting the start, processing time and end of the task such that $s_j + p_j = e_j$. For every ordered pair of tasks J_i and J_j, $i < j$, we introduce a boolean variable b_{ij} standing for the ordering between J_i and J_j . The value of b_{ij} is equal to 1 if J_i precedes J_j and to 0 otherwise, *i.e.* if J_j precedes J_i. The variables s_i, e_i, s_j, e_j, and b_{ij} are linked by a disjunction constraint, on which bounds consistency is maintained, and defined by:

$$\texttt{disjunction}(J_i, J_j, b_{ij}) \;\Leftrightarrow\; (e_i \leq s_j \wedge b_{ij} = 1) \vee (e_j \leq s_i \wedge b_{ij} = 0).$$

For n jobs, we must state a quadratic number $(n(n-1)/2)$ of disjunction constraints. The next constraint imposes that the objective variable Z is equal to the maximal lateness of the tasks and the second next enforces that the makespan T of the schedule is the latest completion time of its tasks. We also state a redundant constraint on the objective variable which imposes that the maximal lateness is greater than the makespan minus the greatest due date.

$$Z = \max_{1 \leq j \leq n}(e_j - d_j), \qquad T = \max_{1 \leq j \leq n}(e_j), \qquad T - \max_{1 \leq j \leq n}(d_j) \leq Z.$$

Model Reinforcement. The model above is enough, but we can get stronger inference by adding a disjunctive global constraint:

$$\texttt{disjunctive}(\langle J_1, \ldots, J_n \rangle, T) \qquad (\texttt{heavy})$$

We can also state a cubic number of boolean clauses which enforce the transitivity of the precedence relations (assuming that b_{vu} denotes $1 - b_{uv}$ if $u > v$):

$$b_{ik} = v \wedge b_{kj} = v \Rightarrow b_{ij} = v, \quad 1 \leq i < j \leq n,\ 1 \leq k \leq n,\ v \in \{0,1\} \ \ (\texttt{clauses})$$

Next constraints break symmetries by ordering pairs of jobs with equal durations. Their correctness can be proven by using simple interchange arguments:

$$(r_i \leq r_j) \wedge (p_i = p_j) \wedge (d_i \leq d_j) \implies b_{ij} = 1 \qquad 1 \leq i \leq n,\ 1 \leq j \leq n \ \ (\texttt{order})$$

Cost-based Domain Filtering. We have tested our framework with two well-known relaxations: $1|r_j; prec; pmtn|L_{max}$ allowing preemption and that can be solved in quadratic time by the *modified due date algorithm*, and $1|prec|L_{max}$ ignoring the release dates and that can be solved in quadratic time by the *Lawler algorithm* (see e.g. [3]). Thus, we provide two instances of the template constraint: `relaxation[pmtn]`(Z, b, J) and `relaxation[prec]`(Z, b, J). For both relaxations, we implemented the following primitives: `buildRelaxation` initializes in $O(n^2)$ the precedences (given b_{ij}), the release and due dates (given s_j and d_j); `relax` executes in $O(n^2)$ the corresponding algorithm; `probeDecision` calls a ($\leq O(n^2)$) `updateRelaxation` method instead of `buildRelaxation`; `isFeasible` checks in $O(n)$ the feasibility of the solution returned by `relax`, *i.e.* either the absence of preemption in `[pmtn]` or the respect of the release dates in `[prec]`.
We also overloaded the default `decisionList` primitive in order to only test a subset of disjunctions which may actually change the current relaxed solution. We propose two alternatives illustrated in Figure 1: rule `swap` tries in $O(n)$ to swap only the pairs of jobs with consecutive starting times in the relaxed solution; rule `sweep` additionally tries in $O(n^2)$, for `[pmtn]` only, to fix disjunctions where a job is preempted by another. Note the modularity of the components since we can state either zero or one or both relaxations in the model.

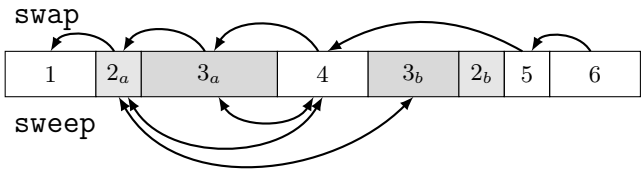


Fig. 1: Overloading `decisionList`$(\mathcal{D})$ for the relaxation $1|r_j; prec; pmtn|L_{max}$.

3 Experimental Results

This section presents computational experiments conducted to evaluate our framework. The implementation is based on the java library Choco [17] and tested on randomly generated instances proposed by Jouglet [11] ranging from 20 to 400 jobs (1440 instances: 240 for each $n \in \{20, 50, 100, 150, 200\}$ and 120 for each $n \in \{300, 400\}$). The model is solved by a B&B algorithm guided by branching on the boolean variables b_{ij} taken in lexicographic order. This realizes the standard search strategy aiming at fixing the disjunctions between tasks. We will evaluate the efficiency of the framework under two optimization procedures: *top-down* and *bottom-up*. The *top-down* procedure starts with an upper bound *ub* and tries to improve it. The *bottom-up* procedure starts with a lower bound *lb* as target upper bound which is incremented by one unit until the problem becomes feasible. The time limit has been fixed to 100 seconds for solving one instance. In the result tables, opt denotes the number of instances solved optimally by the CP-based B&B, and $\tilde{t}$, $\tilde{n}$ and $\tilde{b}$ denote the average time (in seconds), the average numbers of nodes and backtracks for this subset of instances.

Domain Initialization is an important preprocessing step to discard easy instances and deductions. It consists in a) solving both relaxations to compute a lower bound, b) if the relaxed solution is not feasible, computing an upper bound using a simple randomized list heuristics over 100 iterations, c) if the upper bound is not optimal, adjusting the domains of the variables accordingly. To see the impact of this procedure on the basic model without any redundant nor optimization constraint, we examined the 480 instances with less than 50 jobs. Results show that domain initialization is worthwhile because it solves, alone, 189 instances optimally. It then allows the CP engine to solve more instances, in less time, less nodes and less backtracks: for instance, the *top-down* procedure solves 86 additional instances within 8 seconds in average whereas, without domain initialization, it solves only a total of 21 instances within 20 seconds in average. Finally, the *bottom-up* procedure gives the best results since it solves, after initialization, 110 additional instances within 5 seconds in average. This indicates that the simple relaxations we considered give quite tight lower bounds for the problem. In the following experiments, we only consider those "non-trivial" instances that are not solved by the initialization procedure alone.

Performance of the Relaxation-based Constraints. We evaluate the impact of our optimization constraints `relaxation[pmtn]` and `relaxation[prec]` upon the basic model. Table 1a gives results of the *bottom-up* and *top-down* procedures on the 818 non-trivial instances with less than 200 jobs: `basic` is the basic model, `pmtn`, `prec` and `pmtn+prec` invoke pruning without filtering for each or both relaxations, `*-swap` and `*-sweep` invoke pruning and cost-based filtering. The results clearly show the necessity of considering lower bounds during the resolution: the preemptive relaxation allows to solve at least 6 times more instances at optimality. With the *bottom-up* procedure, cost-based filtering helps

	bottom-up				*top-down*			
	opt	$\tilde{t}$	$\tilde{n}$	$\tilde{b}$	opt	$\tilde{t}$	$\tilde{n}$	$\tilde{b}$
`basic`	117	6.4	118k	181k	86	8.4	184k	278k
`pmtn`	743	5.4	683	313	**587**	10.1	6.4k	7.3k
`pmtn_swap`	**766**	7.0	334	93	565	15.0	2.5k	2.7k
`pmtn_sweep`	764	7.6	261	46	552	14.2	1.9k	1.9k
`prec`	149	4.6	47k	71k	109	9.9	117k	173k
`prec_swap`	**158**	5.1	41k	61k	**114**	7.5	74k	111k
`pmtn_prec`	743	5.6	690	332	584	10.6	6.2k	7.0k

(a) Cost-based filtering.

	opt	$\tilde{t}$	$\tilde{n}$	$\tilde{b}$
`pmtn_sweep`	764	7.6	261	46
`order`	785	7.7	296	68
`heavy`	782	7.0	213	15
`order+heavy`	**807**	5.9	190	9
`clauses`	456	64.5	89	13

(b) Constraint models.

Table 1: Results on 818 non-trivial instances with less than 200 jobs.

then to solve more than 20 instances out the 75 remaining instances while greatly reducing the tree size. With *top-down* instead, cost-based filtering occurs less often as the upper bound is not tight anymore. The second relaxation (`prec`) is much less effective, because it ignores the release dates which are an important information, especially at the beginning of the search when only a few disjunctions are fixed. It even hinders the preemptive relaxation when both relaxations (`pmtn+prec`) are used. However, as the lower bounds are less tight, cost-based filtering appears as an improvement even with the *top-down* procedure.

Model Reinforcement. From the results above, we retain now the *bottom-up* procedure on the `pmtn_sweep` model. Table 1b shows how the model is reinforced by each type of redundant structural constraints: the `clauses` are both time and memory consuming, while `heavy` and `order` allow to solve 18 and 21 additional instances. Actually, they are even complementary since only 11 instances remain unsolved by stating both `heavy` and `order`.

Comparisons of Branching Strategies and to a "Natural" Approach. Branching strategies in scheduling can be divided into two main families: assigning starting times (e.g. `setTimes` [18]) or fixing precedences (e.g. `profile` [19] and `slack/wdeg` [20]). The top part of Table 2 show the behavior of our best model (`pmtn_sweep`+`heavy`+`order`) in function of the different strategies within a *bottom-up* procedure. They are compared on the whole set of 995 non-trivial instances with up to 400 jobs.

In the context of lateness minimization, `profile` and `slack/wdeg` do not help when compared to the static `lex` strategy. In fact, the increased depth of the search tree prevents to prove optimality when the algorithm starts to backtrack. `setTimes` is not compatible at all with our model as many relaxed solutions become infeasible because of fixed starting times. Actually, this "natural" strategy is not more effective with the "natural" model (*i.e.* the `heavy` model without relaxation), as shown by the results in the bottom part of Table 2. These last results confirm the predominance of relaxation-based reasoning over redundant structural constraints or branching strategies, when optimizing other objectives than the makespan in a scheduling problem of that kind.

	opt	$\tilde{t}$	$\tilde{n}$	$\tilde{b}$
`lex`	**899**	9.9	246	9
`profile`	867	11.9	505	1
`slack/wdeg`	805	13.7	556	1
`setTimes`	490	3.5	77	95
`natural_bu`	**390**	2.7	1.9k	9.1k
`natural_td`	311	3.4	3.6k	15.1k

Table 2: Main Results.

Bibliography

[1] Rina Dechter. Temporal constraint networks. In *Constraint Processing*, pages 333–362. Morgan Kaufmann, San Francisco, 2003.

[2] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-based scheduling: applying constraint programming to scheduling problems*, volume 39 of *International Series in Operations Research and Management Science*. Springer, 2001.

[3] Peter Brucker. *Scheduling algorithms*. Springer Verlag, 2007.

[4] Claude Le Pape. Constraint-based scheduling: A tutorial. In *First International Summer School on Constraint Programming*, 2005.

[5] Filippo Focacci, Andrea Lodi, and Michela Milano. Cost-based domain filtering. In Joxan Jaffar, editor, *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP 1999)*, volume 1713 of *Lecture Notes in Computer Science*, pages 189–203. Springer, 1999.

[6] P. Baptiste, C. Le Pape, and L. Péridy. Global constraints for partial csps: A case-study of resource and due date constraints. *Principles and Practice of Constraint Programming*, pages 87–101, 1998.

[7] Philippe Baptiste, Jacques Carlier, and Antoine Jouglet. A branch-and-bound procedure to minimize total tardiness on one machine with arbitrary release dates. *European Journal of Operational Research*, 158(3):595–608, 2004.

[8] A. Kéri and T. Kis. Primal-dual combined with constraint propagation for solving rcpspwet. *Operations Research Proceedings 2005*, pages 685–690, 2006.

[9] Jean-Noël Monette, Yves Deville, and Pascal Van Hentenryck. Justin-time scheduling with constraint programming. In *International Conference on Automated Planning and Scheduling (ICAPS'09)*, pages 241–248, 2009.

[10] A. Kovács and J.C. Beck. A global constraint for total weighted completion time for unary resources. *Constraints*, 16(1):100–123, 2011.

[11] Antoine Jouglet. Single-machine scheduling with no idle time and release dates to minimize a regular criterion. *Journal of Scheduling*, to appear. URL http://dx.doi.org/10.1007/s10951-010-0185-x.

[12] Arnaud Malapert, Christelle Guéret, and Louis-Martin Rousseau. A constraint programming approach for a batch processing problem with non-identical job sizes. *European Journal of Operational Research*, 2011.

[13] Peter Brucker, Sigrist Knust, and Christoph Dürr. The scheduling zoo. http://www.lix.polytechnique.fr/d̃urr/query/.

[14] H. Bräsel, L. Dornheim, S. Kutz, M. Mörig, and I. Rössling. Lisa-a library of scheduling algorithms. 2006.

[15] Jean-Noël Monette, Yves Deville, and Pascal Van Hentenryck. Aeon: Synthesizing scheduling algorithms from high-level models. *Operations Research and Cyber-Infrastructure*, pages 43–59, 2009.

[16] Z. Liu. Single machine scheduling to minimize maximum lateness subject to release dates and precedence constraints. *Computers & Operations Research*, 37(9):1537–1543, 2010.

[17] Choco. http://choco.mines-nantes.fr.

[18] Claude Le Pape, Philippe Couronne, Didier Vergamini, and Vincent Gosselin. Time-versus-capacity compromises in project scheduling. In *Proceedings of the Thirteenth Workshop of the U.K. Planning Special Interest Group*, 1994.

[19] J. Christopher Beck, Andrew J. Davenport, Edward M. Sitarski, and Mark S. Fox. Texture-based heuristics for scheduling revisited. In *AAAI/IAAI*, pages 241–248, 1997.

[20] Diarmuid Grimes, Emmanuel Hebrard, and Arnaud Malapert. Closing the open shop: Contradicting conventional wisdom. In Ian P. Gent, editor, *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP 2009)*, volume 5732 of *Lecture Notes in Computer Science*, pages 400–408. Springer, 2009.