Understanding the Potential of Propagators

Sascha Van Cauwelaert[1], Michele Lombardi[2], and Pierre Schaus[1]

[1] {sascha.vancauwelaert,pierre.schaus}@uclouvain.be, UCLouvain
[2] michele.lombardi2@unibo.it, University of Bologna

Abstract. Propagation is at the very core of Constraint Programming (CP): it can provide significant performance boosts as long as the search space reduction is not outweighed by the cost for running the propagators. A lot of research effort in the CP community is directed toward improving this trade-off, which for a given type of filtering amounts to reducing the computation cost. This is done chiefly by 1) devising more efficient algorithms or by 2) using on-line control policies to limit the propagator activations. In both cases, obtaining improvements is a long and demanding process with uncertain outcome. We propose a method to assess the potential gain of both approaches before actually starting the endeavor, providing the community with a tool to best direct the research efforts. Our approach is based on instrumenting the constraint solver to collect statistics, and we rely on *replaying* search trees to obtain more realistic assessments. The overall approach is easy to setup and is showcased on the Energetic Reasoning (ER) and the Revisited Cardinality Reasoning for BinPacking (RCRB) propagators.

Keywords: Constraint Programming, Propagator, Analysis, Energetic Reasoning, BinPacking.

1 Introduction

Propagation is undoubtedly one of the signature features of Constraint Programming (CP): it makes a constraint solver capable of skipping large portions of the search space, possibly achieving significant performance boosts. In practice, the effectiveness of the approach depends on the balance between the time saved by filtering values and the time spent in running the propagators. Improving this trade-off is the objective of huge research efforts in the CP community.

Here, *we consider the specific case where the goal is to optimize the performance of a given propagation technique, without changing its input-output behavior*. For example, we may be interested in finding a more efficient way to enforce Generalized Arc Consistency (GAC) for a specific constraint. In general, this goal can be achieved by either 1) devising more efficient algorithms that achieve the same filtering, or by 2) guarding the activation of the propagator with a necessary condition to reduce fruitless activations[3]. In both cases, obtaining improvements is a long and demanding process with uncertain outcome.

[3] A more general approach consists in trying to *predict* when the propagator should be triggered: we plan to develop tools to analyze this scenario as part of future research.

As an example, the SEQUENCE constraint was introduced in 1994 [5], but no poly-time GAC algorithm was available until 2006 [21]. Then, the original GAC run time of $O(n^3)$ was not low enough to consistently beat weaker (but cheaper) propagators. This motivated improvement efforts that are still ongoing [9, 11, 6]. The trade-off between computation time and pruning power is even more critical for NP-hard constraints. For example, Energetic Reasoning (ER) was proposed as a (powerful) filtering technique for CUMULATIVE in the nineties (see [14, 3]): however, the approach has never been widely employed due to its large run time. Improving the original $O(n^3)$ complexity took in this case around 20 years [12], while an approach to reduce the overhead by guarding the ER activation with a necessary condition was presented only in 2011 [7].

In general, *this line of research would greatly benefit from tools and methods to probe the potential of propagation techniques and to assess the likely impact of specific improvement measures.* Such tools would allow the researchers to focus their efforts in the most promising directions (notice that for preliminary analysis, profiling tools already allow to reason about potential linear speedups).

A typical approach for evaluating propagators consists in measuring time and fails w.r.t. a baseline propagator, on a set of benchmark instances that are solved to completeness. This allows to asses the propagator performance, but provides little or no information on how to improve it. It is also common to use static search strategies to make the evaluation fair and rigorous, with the risk to reduce the analysis significance, since dynamic strategies are often preferred in practice. Finally, the need to solve the problems to completeness may bias the analysis toward relatively small instances.

We propose to extend this basic evaluation approach by: 1) instrumenting the solver to collect information about the constraint; 2) storing and *replaying* search trees to enable fair comparisons with arbitrary search strategies and instance sizes. Our approach is simple and allows to assess the amount of improvement that could be obtained by reducing the propagator run-time or by controlling its activation. We use the Energetic Reasoning (ER) and the Revisited Cardinality Reasoning for BinPacking (RCRB) propagators as case studies.

2 The Proposed Approach

Formally, we consider the problem of evaluating a filtering function ϕ that maps a set of domains $D_0, \dots D_{n-1}$ to a second set of domains $D'_0, \dots D'_{n-1}$ such that $D'_i \subseteq D_i$. In practice, ϕ may represent a propagator for enforcing GAC or a domain-specific consistency level (e.g. Energetic Reasoning), or it can be some kind of meta-propagation scheme such as Singleton Arc Consistency [8].

We assume we are interested in reducing the time for computing ϕ, without changing the function definition. In particular, our goal is to assess the potential of two improvement directions: 1) increasing the efficiency of the current implementation and 2) guarding the activation of ϕ with a necessary condition.

Measuring the Performance: In order to make such an assessment, we must first be able to measure the performance of the current implementation of ϕ. Like many other approaches, we do this by comparing the time needed to solve a target CSP with and without ϕ. Let M and $M \cup \phi$ denote the two CSPs, with their variables, domains, and constraints. For the comparison to be meaningful, two well known conditions must be respected:

1. The two runs must explore the same search space;
2. All search nodes that are visited by both runs are visited in the same order.

The first requirement is always met as long as M and $M \cup \phi$ are semantically equivalent (i.e. they have the same solutions) and the problem is solved to completeness (feasibility or optimality).

Without the second requirement, one of the approaches could get an unfair advantage if the search strategy quickly hits a feasible solution (and stops, for feasibility problems), or a high-quality solution (and gets a good bound, for optimality problems). Moreover, if the second requirement is satisfied, then the nodes visited when solving $M \cup \phi$ will always be a subset of (or the same as) those visited when solving M. Typically, this is all guaranteed by using static search strategies. As an alternative, we propose an approach based on *replaying* search trees, which does not suffer from most of the drawbacks discussed in Section 1.

The Replay Technique: For the sake of precision, it is useful to introduce some notation at this point. As it is quite common in CP, we view tree search as a recursive process, where the search space is iteratively decomposed by opening choice points and posting constraints on each branch. Formally, we can define a search strategy as a function b that given the current state of the search and of a problem M returns a sequence of constraints $c_0, c_1, \ldots$ to be posted each on a different branch. By "search state" we refer to search parameters that are not part of the model (i.e. time markers for the $\mathit{SetTimes}$ strategy). The whole search process can be seen as the evaluation of a recursive function $\mathit{traverse}(b, M)$ having as parameters the search strategy b and the target problem M.

We guarantee the satisfaction of both requirements for measuring the performance by storing in a tree-like structure, during one run: 1) the branching constraints and 2) the search state. We then force the following run to post exactly the same constraints at the same search nodes. This is done by introducing two wrapper search strategies called $\mathit{store}(b)$ and $\mathit{replay}(b)$ that respectively memorize and re-post the constraints returned by the strategy b. Then, in order to evaluate a propagator ϕ, we simply run in sequence:

$$\mathit{traverse}(\mathit{store}(b), M) \tag{1}$$
$$\mathit{traverse}(\mathit{replay}(b), M) \tag{2}$$
$$\mathit{traverse}(\mathit{replay}(b), M \cup \phi) \tag{3}$$

and we compare the results of the two latter runs, which both use $\mathit{replay}(b)$ and hence incur the same search overhead. It is important that the first run is done with the baseline problem M, because, thanks to the additional propagation performed by ϕ, the run with $M \cup \phi$ may skip some parts of the stored tree. However, all of the runs will always explore the same space and visit the shared nodes in the same order.

This approach offers two significant advantages: 1) it allows to tackle arbitrarily large instances, since a time limit can be enforced on the first run and the second run will still be guaranteed to explore the same search space. 2) It allows to use any search strategy, including dynamic ones, making the evaluation more realistic. The comparison remains artificial to some degree, because an actual dynamic strategy may behave differently on the two runs. Still, the ability to make fair comparisons using an arbitrary strategy is a very valuable contribution. Our replay technique is easy to implement on mosts solvers that allow the user to write custom search strategies.

Assessing the Propagator Potential: In order to assess the potential of improving the efficiency of ϕ or controlling its activation, we instrument the solver to collect detailed information about the propagator. Specifically, we store the total time for running ϕ, making a distinction between activations that actually lead to some pruning and fruitless activations. The two time statistics are respectively referred to as t_ϕ^+ and t_ϕ^-. We collect the information by introducing a wrapper function $stats(\phi)$ that checks the domain sizes, then runs ϕ, and finally checks the domains again and stores the elapsed time. The overhead for the collection process is properly subtracted. Once again, this approach is easy to implement on most solvers that allow the user to write new propagators.

It is now easy to get a rough, but valuable, estimate of the impact of specific measures on the solution time. In particular, let $t(b, M)$ be the time required to solve the problem M with the strategy b (i.e. to run $traverse(b, M)$). Then we can estimate the impact of reducing the run time of ϕ by a factor $\mu \in [0, 1]$ by computing:

$$t(replay(b), M \cup stats(\phi)) - \mu \cdot (t_\phi^+ + t_\phi^-) \tag{4}$$

i.e. by subtracting a fraction of the total computation time of ϕ. Similarly, we can assess the impact of guarding ϕ with a necessary condition that stops a fraction $\mu \in [0, 1]$ of the fruitless propagator activations. This is done by computing:

$$t(replay(b), M \cup stats(\phi)) - \mu \cdot (t_\phi^-) \tag{5}$$

This simple, linear, approach allows us to compare *fictional* implementations of ϕ with real ones. By doing so, we get a chance to explore which values of μ would be necessary for beating the baseline, and we get a better understanding of the effort required to achieve such goal. In particular, we can approximately evaluate the impact of having an hypothetical time complexity for a fictional propagator. For instance, if the current implementation for ϕ is in $O(n^3)$ (where n is the number of variables), then we can estimate roughly what would be its cost for an $O(n^2)$ algorithm by choosing $\mu = (n - 1)/n$ in equation 4.

Deeper and more general insights can be obtained by comparing fictional and real propagators on full benchmarks. To this purpose, we rely on *performance profiles* [13]. A performance profile is a cumulative distribution function $F(\tau)$ of a given performance metric τ. In our case, the τ value is the ratio between the solution time of a target approach and that of the baseline. For the sake of clarity, if $F(2) = 0.75$ for an approach, it means that its performance is within a factor of 2 from the baseline in 75% of the benchmark problems. Assuming the benchmark is representative enough, the value of $F(\tau)$ can be interpreted as a probability.

Formally, let $\phi_0, \phi_1, \dots$ be the set of all considered implementations of ϕ (real and fictional alike), and let $\mathcal{M}$ be the set for all problems (instances) in the benchmark. Then the performance profile of ϕ_i is given by:

$$F_{\phi_i}(\tau) = \frac{1}{|\mathcal{M}|} \left| \left\{ M \in \mathcal{M} : \frac{t(replay(b), M \cup \phi_i)}{t(replay(b), M)} \leq \tau \right\} \right| \tag{6}$$

where $t(replay(b), M \cup \phi_i)$ for fictional implementations of ϕ is computed using Equation (4) or (5).

Reading of Performance Profiles: An important value of a given performance profile $F_{\phi_i}(\tau)$ is in $\tau = 1$. For a given ϕ_i, $F_{\phi_i}(\tau = 1)$ gives the percentage of instances that can be solved using $M \cup \phi_i$ in a time less (or the same) time as the baseline model M. Although F_M is not represented, it would actually be a step function $F_M(\tau < 1) = 0$ and $F_M(\tau \geq 1) = 100\%$. The space of τ is therefore divided in two important regions, $\tau < 1$ and $\tau \geq 1$. If $F_{\phi_i}(\tau) = 100\%$ for some $\tau < 1$, then using the model $M \cup \phi_i$ is always better than using the baseline, i.e. $M \cup \phi_i$ provides a speed-up for every instance. Unfortunately, this situation rarely happens in practice and it is thus interesting to read more carefully the performance profile. For a given pair ϕ_i, ϕ_j it is interesting to observe $F_{\phi_i}(\tau)$ - $F_{\phi_j}(\tau)$. It indicates the *gain* of ϕ_i over ϕ_j. That is, $F_{\phi_i}(\tau)$ - $F_{\phi_j}(\tau)$ reflects how many more (or less) instances can be solved by using $M \cup \phi_i$ instead of $M \cup \phi_j$ within a factor τ of the baseline time. Finally, the region above $F_\phi(\tau)$ for $\tau < 1$ is very informative, as it exhibits the gain of a given ϕ_i compared to the baseline M **and** to $M \cup \phi$, i.e., the two non-fictional models. Finally, instances with similar performance give rise to step-like changes in $F(\tau)$, while a linearly growing $F(\tau)$ is symptomatic of a diversified performance across the benchmark.

Limitation of the approach: A bottleneck of our approach is the need to store the search tree in memory. After an experimentation on toy problems with only a few constraints (such as the n-queens) we found it reasonable that no more than $\sim 5 \times 10^6$ nodes are created per minute on a standard laptop. Our data structure to store the branching decisions does not use more than 40 bytes per node. Hence, assuming that 16 GB of memory are available, we can record search attempts up to 40 minutes long. We believe this time limit should be large enough to collect valuable statistics in practice.

3 Experimentation

We applied our approach to two propagators, namely *Energetic Reasoning* (ER, see [14, 3]) and *Revisited Cardinality Reasoning for BinPacking* (RCRB, see [18]). Both the approaches provide powerful filtering, but are expensive to run, so that the design of more efficient implementations has a strong appeal. In order to assess the potential for improvements, we considered the following classes of fictional implementation:

- ϕ_{μ}^{cost}, i.e. an implementation for which the time is reduced by a factor μ.
- $\phi_{\mathcal{O}(f(n))}^{cost}$, i.e. an implementation for which the time complexity is $\mathcal{O}(f(n))$.
- ϕ_{p}^{oracle}, i.e. an implementation that guards ϕ with a necessary condition causing useless activations with a probability p.

The profile of ϕ_{0}^{oracle} (perfect necessary condition) bounds the gain that can be obtained by any necessary condition. The profile of $\phi_{1}^{cost}(\tau)$ (zero-cost implementation), or $\phi_{\mathcal{O}(1)}^{cost}(\tau)$, bounds the performance of any possible implementation. Against common intuition, ϕ_{1}^{cost} is not guaranteed to beat the baseline, since a weak filtering done by ϕ may trigger other (possibly expensive) propagators.

Experimental Set-up: We used the constraint solver *OscaR* [17] and ran instances on AMD Opteron processors (2.7 GHz). For each instance, we limited the run-time of $\mathit{traverse}(\mathit{store}(b), M)$ to 600 seconds and the run-time of $\mathit{traverse}(\mathit{replay}(b), M)$ and $\mathit{traverse}(\mathit{replay}(b), M \cup \phi)$ to 1200 seconds. Instances for which either $\mathit{traverse}(\mathit{replay}(b), M \cup \phi)$ timed out or $\mathit{traverse}(\mathit{replay}(b), M)$ took less than 1 second were filtered out. The target propagator ϕ was executed with low priority by the constraint scheduler.

Energetic Reasoning: We analyzed the *ER* propagator for the CUMULATIVE constraint[1, 2] on *Resource Constrained Project Scheduling Problems* (RCPSP). The baseline model M employs the Timetabling algorithm from [4] and the ER Checker [3], which both run in $\mathcal{O}(n^2)$ [3, 12]. We did not use the improvements proposed in [12]. We use a dynamic search strategy, i.e. the classical *SetTimes* approach from [16]. We consider two benchmarks: the BL instances [2] (20-25 activities) and the PSPLIB (j30 and j90, with 30 and 90 activities) [15]. We focus on investigating, for the chosen benchmarks: 1) the potential benefit of having an ER algorithm running in $O(n^2)$ rather than in $O(n^3)$; 2) the potential benefit of a perfect necessary condition (see [10] and [7] for related works).

Figure 1 and 2 report profiles respectively for the BL and j90 instances. The real ER propagator beats the baseline in $\sim 50\%$ of the cases for BL, but only in $\sim 10\%$ of the cases for j90. The larger problem size is a likely reason for the performance drop, so it is interesting to analyze the fictional, reduced-cost implementations (left-most figures). In the BL benchmark a cost reduction translates to roughly proportional benefits. On j90, an $O(n^2)$ ER would lead to dramatic performance improvement, but it would beat the baseline on only 40% of the cases. More interestingly, there is a 30% portion of instances where the baseline would win *no matter what the efficiency of ER is*, i.e. where the additional pruning of ER is sometimes detrimental rather than beneficial. On such instances, ER cannot lead to benefits unless we find a way to activate it only when it provides an actual advantage. As for using a necessary condition, a perfect approach would enable the same performance of a $O(n^2)$ ER, but even a small mistake probability would cancel most of the benefits.

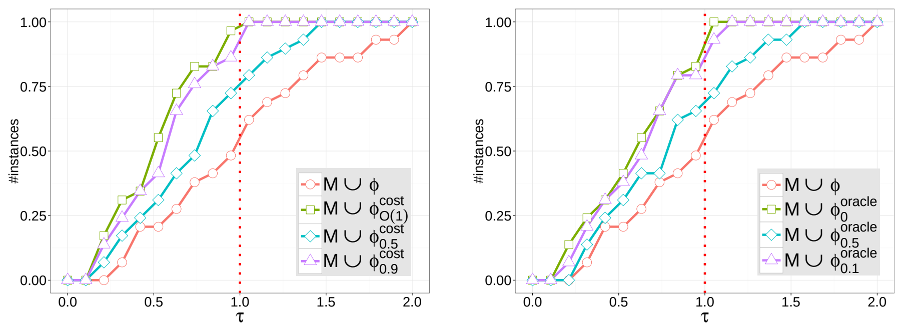

Fig. 1. Performance profiles for real and fictional ER propagators on the BL instances.

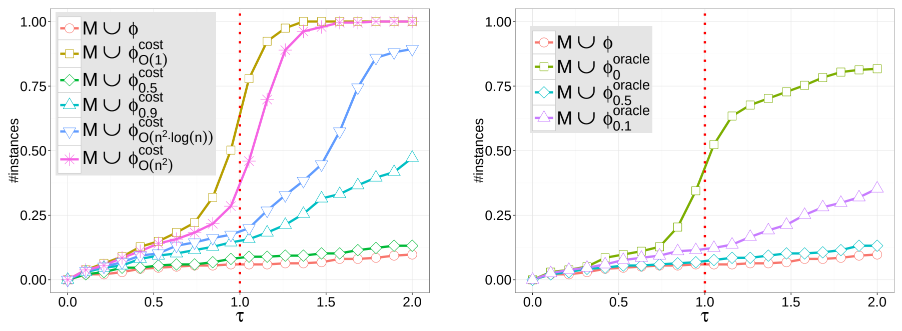

Fig. 2. Performance profiles for real and fictional ER propagators on the j90 instances.

Figure 3 compares profiles for different search strategies on j30 (*SetTimes* and a binary static approach): the potential gain of reducing the cost is very different for the two strategies, even if the performance of the real propagator is roughly identical. This points out the importance of having an approach for the rigorous comparison of propagators using practical search strategies.

Revisited Cardinality Reasoning for BinPacking: In our analysis of the RCRB propagator, we use as a benchmark the instances of the Balanced Academic Curriculum Problem (BACP) from [19, 18]. The baseline model M employs the BinPacking propagator from [20] and a GCC constraint (model **A** in [18]). The search heuristic is *binary first-fail*, i.e. we choose for branching the variable with the smallest domain and we assign on the left branch the minimum value.

Figure 4 (left) is very informative about the cost of RCRB. We can see that less than 25% of the instances are solved faster than the baseline model. Moreover, reducing its cost down to 0 provides a small gain before $\tau = 1.1$. From then, reducing the cost by a factor 0.9 is enough to solve a lot more of the instances. Hence, reducing the cost would improve considerably the RCRB, but not that much compared to the baseline model as the benefits come "too late" in terms of τ. A similar analysis can be done for figure 4 (right).

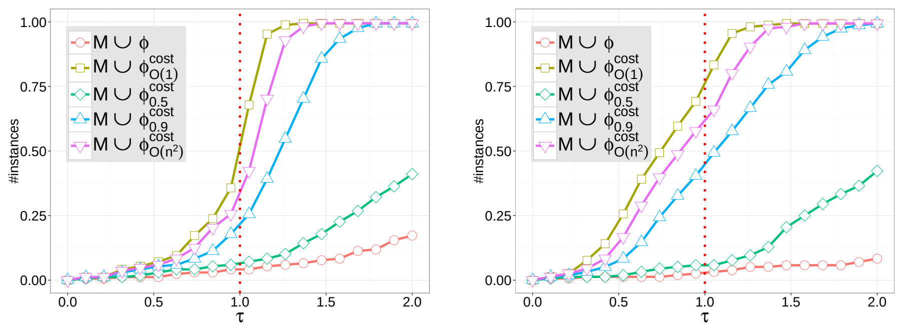

Fig. 3. Performance profiles for the SetTimes (left) and binary static (right) strategies.

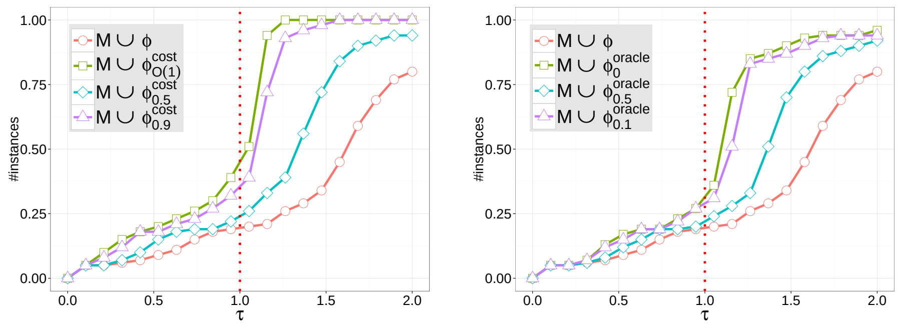

Fig. 4. Performance profiles with fictionally cost-reduced RCRB propagators.

4 Conclusion

Evaluating the potential advantages of reducing the cost of a given filtering procedure is of great importance to make our research efforts as fruitful as possible. In addition, being able to measure exactly the time gain provided by a filtering algorithm permits to reduce the bias in empirical evaluations. As a first step in this direction, we proposed a systematic methodology to simulate the performance of *fictional implementations of a propagator having reduced activation cost*. This is done *before* starting time-consuming research activities to actually reduce the cost. The approach was illustrated for Energetic Reasoning and Revisited Cardinality Reasoning for BinPacking over popular sets of instances. We found that reducing the propagator costs, *even to the point of making it negligible*, might actually be beneficial only on a small subset of a given instance set. Furthermore, this outcome can differ substantially depending on the considered benchmark and on the search strategy.

Acknowledgements We thank the anonymous reviewers for their valuable comments. This research is partially supported by the UCLouvain Action de Recherche Concertée ICTM22C1.

References

1. Abderrahmane Aggoun and Nicolas Beldiceanu. Extending chip in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 17(7):57–73, 1993.
2. Philippe Baptiste and Claude Le Pape. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Constraints*, 5(1-2):119–139, 2000.
3. Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-based scheduling: applying constraint programming to scheduling problems*, volume 39. Springer, 2001.
4. Nicolas Beldiceanu and Mats Carlsson. A new multi-resource cumulatives constraint with negative heights. In *Principles and Practice of Constraint Programming-CP 2002*, pages 63–79. Springer, 2002.
5. Nicolas Beldiceanu and Evelyne Contejean. Introducing global constraints in chip. *Mathematical and computer Modelling*, 20(12):97–123, 1994.
6. David Bergman, André A. Ciré, and Willem Jan van Hoeve. MDD propagation for sequence constraints. *J. Artif. Intell. Res. (JAIR)*, 50:697–722, 2014.
7. Timo Berthold, Stefan Heinz, and Jens Schulz. An approximative criterion for the potential of energetic reasoning. In *Theory and Practice of Algorithms in (Computer) Systems - First International ICST Conference, TAPAS 2011, Rome, Italy, April 18-20, 2011. Proceedings*, pages 229–239, 2011.
8. Christian Bessière and Romuald Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*, pages 54–59, 2005.
9. Sebastian Brand, Nina Narodytska, Claude-Guy Quimper, Peter J. Stuckey, and Toby Walsh. Encodings of the sequence constraint. In *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, pages 210–224, 2007.
10. Sascha Van Cauwelaert, Michele Lombardi, and Pierre Schaus. Supervised learning to control energetic reasoning : Feasibility study. In *Proceedings of the Doctoral Program CP2014*, 2014.
11. Kenil C. K. Cheng and Roland H. C. Yap. An mdd-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2):265–304, 2010.
12. Alban Derrien and Thierry Petit. A new characterization of relevant intervals for energetic reasoning. In *Principles and Practice of Constraint Programming*, pages 289–297. Springer, 2014.
13. Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.
14. Jacques Erschler and Pierre Lopez. Energy-based approach for task scheduling under time and resources constraints. In *2nd international workshop on project management and scheduling*, pages 115–121, 1990.
15. Rainer Kolisch, Christoph Schwindt, and Arno Sprecher. Benchmark instances for project scheduling problems. In *Project Scheduling*, pages 197–212. Springer, 1999.
16. Claude Le Pape, Philippe Couronné, Didier Vergamini, and Vincent Gosselin. Time-versus-capacity compromises in project scheduling. 1994.
17. OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.

18. François Pelsser, Pierre Schaus, and Jean-Charles Régin. Revisiting the cardinality reasoning for binpacking constraint. In *Principles and Practice of Constraint Programming*, pages 578–586. Springer, 2013.
19. Pierre Schaus et al. *Solving balancing and bin-packing problems with constraint programming*. PhD thesis, PhD thesis, Université catholique de Louvain, Louvain-la-Neuve, 2009.
20. Paul Shaw. A constraint for bin packing. In *Principles and Practice of Constraint Programming–CP 2004*, pages 648–662. Springer, 2004.
21. Willem Jan van Hoeve, Gilles Pesant, Louis-Martin Rousseau, and Ashish Sabharwal. Revisiting the sequence constraint. In *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, pages 620–634, 2006.