Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning

Search Strategy Simulation in Constraint Booleanization

Jinbo Huang
Optimisation Research Group, NICTA and
Research School of Computer Science, Australian National University

Abstract

Within the recently proposed *Universal Booleanization* framework, we consider the CUMULATIVE constraint, for which the original Boolean encoding proves ineffective, and present a new Boolean encoding that causes the SAT solver to simulate, largely, the search strategy used by some of the best-performing native methods. Apart from providing motivation for future research in a similar direction, we obtain a significantly enhanced version of Universal Booleanization for problems containing CUMULATIVE constraints.

Introduction

The recently proposed MiniZinc language (Nethercote et al. 2007) is an example attempt toward a common modeling language for constraint programming (CP), and there now exist its implementations based on finite domain (FD) (G12 2009), linear programming (LP) (Brand et al. 2008), satisfiability (SAT) (Huang 2008), combination of FD and SAT in the form of *lazy clause generation* (Ohrimenko, Stuckey, and Codish 2007), as well as interfaces to such popular CP systems as ECLiPSe (ECLiPSe 2009) and Gecode (Schulte, Lagerkvist, and Tack 2009).

In this work we consider the SAT-based approach proposed by Huang (2008), which implements MiniZinc by translating it to Boolean formulas such that any constraint model written in MiniZinc (not involving floating point numbers) can be solved by one or more calls to a SAT solver. A major limitation of the Booleanization of Huang (2008) is the lack of direct support for global constraints. Instead these are supported by expanding each global constraint in the model into a set of more primitive constraints using a standard definition.

For any particular type of constraint X, it is quite possible that Booleanization does not lead to the most efficient solver. However, given the suitability of Booleanization for many types of problems (Huang 2008), there will be situations where a constraint model contains some constraints of type X, but is otherwise best solved by Booleanization. In these situations, it can well be that Booleanizing the whole model will lead to the most efficient solution, even if the constraints of type X, on their own, would have admitted better solution methods. In other words, the goal in this case is not necessarily to devise a Booleanization that will outperform native methods for constraints of type X, but rather one whose effectiveness is maximized, particularly by capitalizing on successful techniques used in native methods.

It is in this spirit that we propose, in this work, a new, substantially improved Boolean encoding of CUMULATIVE, one of the the most widely used global constraints (Rossi, van Beek, and Walsh 2006). We first present an encoding that utilizes recent advances in solving pseudo-Boolean (PB) constraints (Eén and Sörensson 2006), and then show how we can augment the encoding to effectively simulate domain splitting, a search strategy known to be beneficial for CUMULATIVE constraints in native search algorithms (Simonis and O'Sullivan 2008; Huang and Korf 2009).

Empirical results indicate that our new encoding leads to significant improvements over the original Booleanization, while we observe, on the other hand, the efficiency of native solvers over our Booleanization on some of the benchmarks. In concluding the paper, we discuss analytically some strength and weaknesses of our work.

The CUMULATIVE Constraint

Given a set of tasks each of duration d_i and continuously requiring r_i units of a resource, the CUMULATIVE constraint asks that they be scheduled to start respectively at time s_i (which has a finite domain) so that at no point in time does the total resource used by all tasks exceed a given bound b. More formally,

$$\text{CUMULATIVE}(s,d,r,b) \stackrel{def}{=} \bigwedge_{t:\ \min s_i \le t < \max(s_i+d_i)} \Big(\sum_{i:\ s_i \le t < s_i+d_i} r_i \Big) \le b, \qquad (1)$$

where s is an array of bounded integer variables, d and r are arrays of integer constants, and b is an integer constant.

Figure 1 depicts an example CUMULATIVE constraint involving three tasks, along with one of its solutions that minimizes the total time span of the tasks.

In addition to its natural use in scheduling, CUMULATIVE has recently been used as a relaxation of rectangle packing to provide effective pruning for the latter problem (Simonis and O'Sullivan 2008; Huang and Korf 2009). Rectangle packing is deciding whether a given set of rectangles (with fixed

$0 \leq s_1, s_2, s_3 \leq 2$
$d_1 = 2, r_1 = 2$
$d_2 = 2, r_2 = 1$
$d_3 = 1, r_3 = 3$
$b = 3$

CUMULATIVE(s, d, r, b)

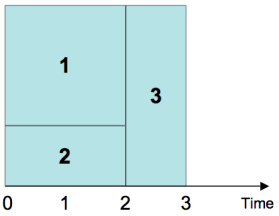

Figure 1: A CUMULATIVE constraint and a solution.

orientations) can be placed into a larger rectangle with no overlap, which has many applications in its own right (Korf 2003). The CUMULATIVE relaxation of a rectangle packing instance asks whether the rectangles can be placed within the two vertical walls of the enclosing rectangle so that at no point along the horizontal line does their total height exceed the height of the enclosing rectangle. This is equivalent to allowing each rectangle to be sliced into vertical strips of unit width to be placed contiguously along the horizontal dimension but independently along the vertical dimension. Note that the CUMULATIVE solution depicted in Figure 1 happens to correspond to a rectangle packing solution with enclosing width equal to the total time span of the tasks and enclosing height equal to the resource bound. In general, this is not always possible.

The CUMULATIVE relaxation allows one to place rectangles in one dimension first, and consider the second dimension only if the one-dimensional placement is not pruned by the CUMULATIVE constraint, which leads to significantly faster solutions than placing rectangles directly in the original two-dimensional space (Simonis and O'Sullivan 2008; Huang and Korf 2009). This relation between the two problems provides additional motivation for the study of the CUMULATIVE constraint, and in fact we shall later adapt a standard rectangle packing benchmark into one for CUMULATIVE for use in our experiments.

Booleanization of CUMULATIVE

Our encoding of CUMULATIVE follows Equation (1), in such a way that each summation in the equation is treated as a PB constraint. We then enlist recent advances in PB constraint solving to further convert them into pure Boolean formulas. On top of this baseline encoding, we then introduce additional Boolean variables and clauses to effectively implement *domain splitting*, a search strategy recently identified to be beneficial for CUMULATIVE constraints.

Baseline Encoding

Let us describe our encoding using the example in Figure 1. We start by creating one Boolean variable for each value of s_1's domain, for a total of three variables s_{10}, s_{11}, s_{12}, and posting the following three constraints:

$$s_{10} \leftrightarrow (s_1 = 0), s_{11} \leftrightarrow (s_1 = 1), s_{12} \leftrightarrow (s_1 = 2). \quad (2)$$

The same is done for s_2 and s_3. These fall under the category of reified integer comparisons discussed in Huang (2008), and can be Booleanized by the existing method, providing the necessary "channeling" between what we are going to do next and the existing encoding of any other constraints in the model.

In the second step, we create $m = 4$ "occupancy" variables for Task 1, $o_{10}, o_{11}, o_{12}, o_{13}$, where m is the total possible time span of all tasks: $m = \max(s_i + d_i) - \min(s_i)$. Each o_{1i} will encode that Task 1 is in process between time points i and $i + 1$. This is achieved by a set of formulas linking the s_{1i} with the o_{1i} variables taking into account the duration $d_1 = 2$ of the task:

$$\begin{aligned} &s_{10} \rightarrow o_{10},\ s_{10} \rightarrow o_{11}; \\ &s_{11} \rightarrow o_{11},\ s_{11} \rightarrow o_{12}; \\ &s_{12} \rightarrow o_{12},\ s_{12} \rightarrow o_{13}. \end{aligned} \quad (3)$$

Again the same is done for every other task. Note that it's not necessary to encode the "unoccupied" time slots (for example, $s_{10} \rightarrow \neg o_{12},\ s_{10} \rightarrow \neg o_{13}$), because if the CUMULATIVE constraint can be satisfied by allowing tasks to run "overtime," it surely can be satisfied the normal way by the same schedule.

In the third step, we write m PB constraints, one for each unit time slot, to encode that the resource bound not be exceeded at any point in time:

$$\begin{aligned} &r_1 \cdot o_{10} + r_2 \cdot o_{20} + r_3 \cdot o_{30} \leq b, \\ &r_1 \cdot o_{11} + r_2 \cdot o_{21} + r_3 \cdot o_{31} \leq b, \\ &r_1 \cdot o_{12} + r_2 \cdot o_{22} + r_3 \cdot o_{32} \leq b. \end{aligned}$$

It now remains to further convert these into pure Boolean formulas. For that we enlist the recent work of Eén and Sörensson (2006), where three methods were proposed for Booleanizing PB constraints, based on binary decision diagrams (BDDs), adder networks, and sorting networks. The BDD-based method is the only one where unit propagation on the Boolean formula enforces arc consistency for the original PB constraint. Although the BDD-based encoding can be larger than those produced by the other two methods, in our experiments it proves the best choice overall for the types of problems we are solving.

This method works by constructing a BDD for the PB constraint, and then converting the BDD into clauses using a standard procedure involving the introduction of auxiliary variables (Eén and Sörensson 2006). For example, the first PB constraint shown above (with $r_1 = 2, r_2 = 1, r_3 = 3, b = 3$ plugged in) is translated into the following three clauses:

$$\neg o_{10} \vee \neg x, \neg o_{20} \vee \neg x, \neg o_{30} \vee x,$$

where x is an auxiliary variable introduced during the translation. One can check that these are indeed equivalent to the PB constraint, and that unit propagation will enforce arc consistency.

In summary, for a CUMULATIVE constraint over n tasks we create $n \cdot |\text{domain}(s)|$ domain variables and $n \cdot m$ occupancy variables (MiniZinc syntax requires the task start times s to all have the same domain), and a proportional number of new constraints. Further Booleanization of each reified integer comparison in the form of Formulas (2) adds a constant number of variables and clauses (Huang 2008).

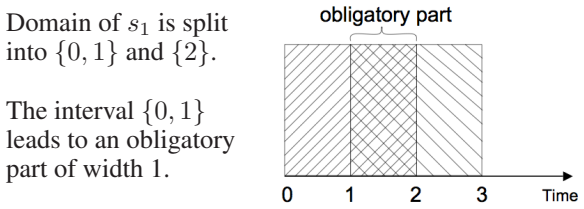

Figure 2: Obligatory part for an interval.

Booleanization of the PB constraints adds additional variables and clauses whose number depends on the external tool that implements the Booleanization and can be exponential in n in the worst case.

Domain Splitting

It is well known that the search strategy is critical for constraint solvers based on backtracking search. By contrast, when a constraint model is Booleanized to be solved by a SAT solver, we lose direct control over the search and have to rely on the generic strategies of whatever SAT solver we choose to use. Although good strategies used by the SAT solver may be sufficiently adaptive so that a good search behavior is achieved despite the solver's blindness to the original problem structure, we show here that in our present case we can do even better by going an extra mile in our encoding to simulate a search strategy known to be beneficial for the original search space.

Specifically, it has been shown that splitting the domains of the s_i variables (start times of the tasks) into intervals and branching on the intervals instead of individual values can significantly improve the efficiency of solving CUMULATIVE constraints. Evidence to this effect has been reported both for the case where an existing CP system is used (Simonis and O'Sullivan 2008), and for the case where the search is implemented from scratch (Huang and Korf 2009).

The effect of branching on an interval instead of an individual value is twofold: It reduces the branching factor, but weakens propagation. Consider the example in Figure 2 where the domain of s_1 (the start time of Task 1) has been split into two smaller intervals $\{0,1\}$ and $\{2\}$. This reduces the branching factor from 3 to 2. If we take the second branch, usual propagation follows based on the fact that Task 1 is now set to occupy time slots 2 and 3 (since $d_2 = 2$). If we take the first branch, however, the exact position of Task 1 in the schedule remains unfixed, preventing the same kind of propagation from taking place.

An important observation here is that we can still utilize partial propagation in these situations. In this example, whether $s_1 = 0$ or 1, we know that time slot 1 must be occupied. This is known as an *obligatory part* (see Figure 2). In general, branching on an interval is now followed by propagation based on the obligatory part for that interval.

While this form of propagation may help prune many branches of the search tree, at some point we must still decide on the values of the variables in order to get a concrete solution, or to prune those branches that remain open after the propagations based on obligatory parts. When to do these decisions thus becomes part of the variable ordering strategy of the search. Examples of such strategies can be found in Simonis and O'Sullivan (2008) and Huang and Korf (2009).

Now to simulate this form of search on our Boolean encoding of the problem, we begin by also splitting the domains of the variables, and creating new Boolean variables to represent the resulting intervals. For the example of Figure 2, we create two new variables $intv_{10}, intv_{11}$ to encode the intervals, and add the following formulas:

$$intv_{10} \vee intv_{11}, intv_{10} \rightarrow (s_{10} \vee s_{11}), intv_{11} \rightarrow s_{12}. \quad (4)$$

The first formula says that at least one interval must be chosen; the other two define the intervals (in practice the second interval, being of size one, does not need additional encoding; we retain it here for the purpose of illustrating the general method). Note that it's not necessary to explicitly require that "at most one interval be chosen" because that is implied by the encoding: Choosing more than one interval would ultimately imply setting the start time to more than one value, which would falsify the encoding. By the same token the use of "$\rightarrow$" instead of "$\leftrightarrow$" in these formulas is sufficient.

Next is to encode the obligatory parts. Continuing with our example, the formula below encodes the obligatory part for the first interval, saying that time slot 1 must be occupied as long as that interval is chosen:

$$intv_{10} \rightarrow o_{11}. \quad (5)$$

We now come to an issue critical to the effectiveness of our encoding. In Formulas (3), we already have

$$s_{10} \rightarrow o_{11}, s_{11} \rightarrow o_{11}, \quad (6)$$

which means if the SAT solver derives $o_{11} = true$ at some point during its search and that eventually contributes to a conflict, it's quite possible that $intv_{10} = true$ and one of $s_{10} = true$ and $s_{11} = true$ have both been set, and hence both have the potential to be identified as part of the cause of the conflict (we assume the use of a clause learning SAT solver), even though $intv_{10} = true$ should be the preferred choice, as that would lead to potentially pruning an interval as opposed to a single value once the conflict clause is learned. In other words, these multiple implications of the same literal in a way confuse the SAT solver, diminishing the effectiveness of the reduced branching factor. The solution is quite simple once the problem is realized: We drop Formulas (6) from the encoding. The SAT solver would now be forced to identify $intv_{10} = true$ as contributing to the conflict, pruning a larger portion of the search space than if a single value in the interval is identified as responsible.

The final issue to discuss is the sizes of the intervals. Larger intervals lead to a smaller branching factor, but narrower obligatory parts and hence weaker propagation. The best compromise has been identified by empirical means in previous work, and can vary across different types of problems (Simonis and O'Sullivan 2008; Huang and Korf 2009). In our case we have found $p \approx 0.9$ to work best, where p is the ratio of interval size to task duration; our experimental results will be reported based on that setting.

In summary, to implement domain splitting we add as many new variables as the number of intervals created by splitting the variable domains, and a roughly proportional number of associated clauses in the form of Formulas (4). The encoding of obligatory parts actually reduces the number of clauses as each clause in the form of Formula (5) replaces several clauses in the form of Formulas (6). In all cases in our experiments the overall encoding size is reduced compared with the baseline encoding.

Experimental Results

We implemented our new encoding on top of FZNTINI (Huang 2008), and will refer to our new program as FZNTINI+. For comparison we use the original FZNTINI, Gecode/FlatZinc (Gecode version 3.0.2 with FlatZinc interpreter version 1.5), and G12/FD (from MiniZinc version 0.9), all of which use the default expansion of CUMULATIVE that comes with MiniZinc. To assess the relative contributions of the baseline coding and domain splitting, we also ran FZNTINI+ with domain splitting disabled on all benchmarks, and will refer to that program as FZNTINI+−. Our hardware is a cluster of CPUs running Linux at 2.4 GHz with 4 GB of RAM. A time limit of 4 hours applied to each run of a solver on each instance.

Our first group of benchmarks are an adaptation of the *consecutive square packing* benchmarks (Korf 2003), and we will refer to them as *consecutive square scheduling*. Specifically, each benchmark is a satisfaction problem specified by a triple (n, w, h), asking for a set of n "square" tasks with $d_i = r_i = i$ for $i = 1, \ldots, n$ to be scheduled with h as the the resource bound and the total time span $\leq w$. For each n from 11 to 20, we create a sequence of pairs (w, h) in nondecreasing order of $w \times h$ using the method of Simonis and O'Sullivan (2008) so that the first satisfiable instance in the sequence gives an optimal schedule with respect to minimizing $w \times h$; the sequence then terminates. This gives us a total of 80 instances, 10 of them satisfiable (one for each n) and the rest unsatisfiable. As each of these consists primarily of a single CUMULATIVE constraint, they provide a good basis for evaluating the effectiveness of our new CUMULATIVE encoding in isolation.

In the MiniZinc model we specify `(first_fail, indomain_split)` as the search strategy, meaning that variables are assigned in increasing order of their domain size, and values are assigned by successively splitting the domain in half, which appears to work best for Gecode/FlatZinc and G12/FD.

Table 1 (top) summarizes the results on these benchmarks. We observe that our baseline encoding (FZNTINI+−) is immediately a step up from the original FZNTINI, solving all instances while the latter only solved the 10 satisfiable ones. With domain splitting enabled (FZNTINI+), the running time further reduces by a factor of 4. Gecode/FlatZinc and G12/FD exhibited similar performance; both of them were over two orders of magnitude slower than FZNTINI+.

Our second group of benchmarks come from the MiniZinc distribution (G12 2009). They are ten instances of a *resource-constrained project scheduling problem* (RCPSP),

n	Instances	**A**	**B**	**C**	**D**	**E**
11	3	10 (1)	0.07	0.06	0.12	0.33
12	6	18 (1)	0.26	0.17	0.29	2
13	5	24 (1)	0.28	0.18	0.77	4
14	8	41 (1)	0.5	0.39	2	8
15	8	99 (1)	2	0.66	15	78
16	8	127 (1)	2	0.94	59	212
17	5	297 (1)	3	2	89	449
18	13	641 (1)	136	16	3552	10632
19	10	2294 (1)	63	15	5783	16034
20	14	497 (1)	44	30	34027	38909 (12)
Σ	80	4044 (10)	247	64	43527	66324 (78)

ID	**A**	**B**	**C**	**D**	**E**
0	–	38	50	–	–
1	–	493	249	–	–
2	–	103	90	2	9
3	–	107	143	3	14
4	–	6282	7675	–	–
5	–	1304	1328	–	–
6	–	970	1050	5	19
7	–	–	1169	8	29
8	–	–	–	–	–
9	–	2616	2898	14	86
Σ	–	11908 (8)	14649 (9)	30 (5)	155 (5)

Table 1: Performance of solvers (**A**: FZNTINI, **B**: FZNTINI+−, **C**: FZNTINI+, **D**: Gecode/FlatZinc, **E**: G12/FD) on consecutive square scheduling (top) and RCPSP (bottom). Shows total time in seconds on solved instances. Number of solved instances in parentheses if not all were solved. For RCPSP, "−" denotes an unsolved instance.

each involving four different resources (hence four CUMULATIVE constraints), between 30 and 120 tasks, and a set of precedence relations between tasks, where the total time span of tasks is to be minimized.

The search strategy that comes with the model is `(smallest, indomain_min)`, meaning that variables are assigned in increasing order of their smallest domain value, and values are assigned in ascending order.

Table 1 (bottom) summarizes the results on these benchmarks (the first column shows the identifier for each instance). Again our baseline encoding (FZNTINI+−) greatly improves on the original FZNTINI, which could not solve any instance. Domain splitting (FZNTINI+) allowed an additional instance to be solved, but did not significantly affect the performance on other instances. We believe this is due to the relatively short task durations (all ≤ 10) in these benchmarks, which lead to small intervals and obligatory parts, diminishing the benefit of domain splitting.

Both Gecode/FlatZinc and G12/FD solved (the same) half of the instances very quickly (their inability to solve the other half is likely due to the unsuitability of the search strategy for those particular instances). We believe one reason why Booleanization was much slower on those instances was that the precedence relations were handled less efficiently. In an FD solver, those relations can be efficiently enforced by simply pruning the domains of variables "on demand" during search. In Booleanization, they are treated

as generic inequalities on integer expressions, which when Booleanized using binary encodings of integers lose much of their propagation power.

Hence, while Table 1 (bottom) demonstrates the effectiveness of our new encoding over the previous one, it gives an example class of instances for which our Booleanization approach does not offer the most efficient solutions in the first place; the interested reader may find additional results to this effect that have been obtained using a specialized solver for RCPSP based on lazy clause generation (Schutt et al. 2009).

Discussion

We have shown that it is possible to simulate a nontrivial search strategy used by native constraint solvers in a static SAT encoding, without having to modify the behavior of the SAT solver. For the CUMULATIVE constraint in particular, we have demonstrated that our new encoding greatly enhances the efficiency of generic SAT-based constraint solving where these constraints are involved.

A major strength of the Universal Booleanization framework lies in the relative power of two different classes of reasoning algorithms. Specifically, clause learning used in modern SAT solvers is understood to be exponentially more powerful, in principle, than simple backtracking search (Pipatsrisawat and Darwiche 2009); a separation remains even when the latter is augmented with (the traditional form of) nogood learning (Katsirelos and Bacchus 2005), as traditional nogoods involve decision variables only and are hence less general than clauses learned in SAT solvers.

On the other hand, Universal Booleanization currently suffers from weak propagation in binary integer arithmetic, as discussed earlier, and our approach to the Booleanization of CUMULATIVE also has the weakness that pruning techniques useful in the original search space may not be easy to incorporate. An example is the pruning method from Huang and Korf (2009), based on the observation that the resource left by already scheduled tasks may not all be usable by the remaining tasks. For example, a single unit of resource left over in some time slot can never be utilized by tasks requiring two units of resource or more. Pruning is effected whenever an insufficient amount of resource is identified for a group of unscheduled tasks. This type of technique requires keeping track of certain measures *during search*; hence it's not at all obvious how it can be incorporated into a *static* encoding of the problem.

Hence there are many avenues that remain to be explored, as to how one may best combine the power of domain-specific techniques with that of SAT, how different types of encodings may be utilized to enhance propagation, and on a more general level, how different techniques may be best hybridized, either automatically or with user assistance, to handle different types of problems.

Acknowledgements

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

Brand, S.; Duck, G. J.; Puchinger, J.; and Stuckey, P. J. 2008. Flexible, rule-based constraint model linearisation. In *10th International Symposium on Practical Aspects of Declarative Languages (PADL)*, 68–83.

ECLiPSe. 2009. The ECLiPSe constraint programming system. http://87.230.22.228/.

Eén, N., and Sörensson, N. 2006. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation* 2:1–26.

G12. 2009. MiniZinc Distribution. www.g12.cs.mu.oz.au/minizinc/.

Huang, E., and Korf, R. E. 2009. New improvements in optimal rectangle packing. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*.

Huang, J. 2008. Universal Booleanization of constraint models. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming (CP)*, 144–158.

Katsirelos, G., and Bacchus, F. 2005. Generalized nogoods in CSPs. In Veloso, M. M., and Kambhampati, S., eds., *Proceedings of the 20th AAAI Conference on Artificial Intelligence (AAAI)*, 390–396. AAAI Press / The MIT Press.

Korf, R. E. 2003. Optimal rectangle packing: Initial results. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS)*, 287–295.

Nethercote, N.; Stuckey, P. J.; Becket, R.; Brand, S.; Duck, G. J.; and Tack, G. 2007. Minizinc: Towards a standard CP modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP)*, 529–543.

Ohrimenko, O.; Stuckey, P. J.; and Codish, M. 2007. Propagation = lazy clause generation. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP)*, 544–558.

Pipatsrisawat, K., and Darwiche, A. 2009. On the power of clause-learning SAT solvers with restarts. In Gent, I. P., ed., *CP*, volume 5732 of *Lecture Notes in Computer Science*, 654–668. Springer.

Rossi, F.; van Beek, P.; and Walsh, T., eds. 2006. *Handbook of Constraint Programming*. Elsevier.

Schulte, C.; Lagerkvist, M.; and Tack, G. 2009. Gecode. http://www.gecode.org/.

Schutt, A.; Feydy, T.; Stuckey, P. J.; and Wallace, M. 2009. Why cumulative decomposition is not as bad as it sounds. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP)*, 746–761.

Simonis, H., and O'Sullivan, B. 2008. Search strategies for rectangle packing. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming (CP)*, 52–66.