Exploiting Short Supports for Improved Encoding of Arbitrary Constraints into SAT

Özgür Akgün, Ian P. Gent, Christopher Jefferson,
Ian Miguel, Peter Nightingale

School of Computer Science, University of St Andrews, St Andrews, UK
{ozgur.akgun, ian.gent, caj21, ijm, pwn1}@st-andrews.ac.uk

Abstract. Encoding to SAT and applying a highly efficient modern SAT solver is an increasingly popular method of solving finite-domain constraint problems. In this paper we study encodings of arbitrary constraints where unit propagation on the encoding provides strong reasoning. Specifically, unit propagation on the encoding simulates generalised arc consistency on the original constraint. To create compact and efficient encodings we use the concept of short support. Short support has been successfully applied to create efficient propagation algorithms for arbitrary constraints. A short support of a constraint is similar to a satisfying tuple however a short support is not required to assign every variable in scope. Some variables are left free to take any value. In some cases a short support representation is smaller than the table of satisfying tuples by an exponential factor. We present two encodings based on short supports and evaluate them on a set of benchmark problems, demonstrating a substantial improvement over the state of the art.

1 Introduction

We address the problem of encoding constraint problems into SAT. This is an important step because it allows us to leverage the rich modelling languages available in constraints such as MiniZinc [26] and Essence Prime [20]. We have previously shown that the constraint modelling tool SAVILE ROW [17] can be used to translate constraint problems directly to SAT, exploiting automated modelling techniques such as common subexpression elimination [21]. We add to the important and growing literature on modelling of constraints in SAT [6]. Most study has been devoted to constraints such as linear constraints including the special case of cardinality constraints [1,2,8,24].

In this paper we show that we can improve SAT models of table constraints by exploiting short supports. Table constraints are vital in constraint modelling as they allow arbitrary constraints to be expressed. Table constraints can be expressed in SAT in such a way as to ensure that unit propagation in the SAT encoding performs reasoning equivalent to that done by generalised arc consistency (GAC) in the constraint problem [3]. A short support of a constraint is similar to a satisfying tuple, but a short support is not required to assign every variable: some variables are left free to take any value. Where it is possible, exploiting short supports has proved to improve efficiency of GAC propagation [12,19]. We show that Bacchus's encoding of table constraints into SAT can be adapted to exploit short supports. This can lead to much smaller encodings and

faster propagation, while still obtaining GAC. We present two encodings for table constraints with short supports into SAT. We get the advantages of modern SAT solvers automatically, such as generating explanations of failure and learning.

Short supports represent one method of compressing table constraints, and other methods have been proposed: MDDs [7], C-tuples [13] and their generalisation [22], and Smart Tables [14]. Uniquely, short supports allow us to directly improve the encoding of Bacchus without introducing any complications to it.

2 Preliminaries

The Propositional Satisfiability Problem (SAT) is to find an assignment to a set of Boolean variables so as to satisfy a given Boolean formula, typically expressed in conjunctive normal form [5]. SAT has many important applications, such as hardware design and verification, planning, and combinatorial design [15]. Powerful, robust solvers have been developed for SAT employing techniques such as conflict-driven learning, watched literals, restarts and dynamic heuristics for backtracking solvers [16], and sophisticated incomplete techniques such as stochastic local search [23].

A *constraint satisfaction problem* (CSP) is defined as a set of variables X, a function that maps each variable to its domain, $D : X \rightarrow 2^{\mathbb{Z}}$ where each domain is a finite set, and a set of constraints C. A constraint $c \in C$ is a relation over a subset of the variables X. The *scope* of a constraint c, named scope(c), is the set of variables that c constrains. During a systematic search for a solution to a CSP, values are progressively removed from the domains D. Therefore, we distinguish between the *initial* domains and the *current* domains. The function D refers to the current domains and D_s to the initial domains. A *literal* is a variable-value pair (written $x \mapsto v$). A literal $x \mapsto v$ is *valid* if $v \in D(x)$. The size of the largest initial domain is d. For a constraint c we use r for the size of scope(c). A constraint c is Generalised Arc Consistent (GAC) if and only if there exists a full-length support containing every valid literal of every variable in scope(c). GAC is established by identifying all literals $x \mapsto v$ for which no full-length support exists and removing v from the domain of x. We consider only algorithms for establishing GAC in this paper. A *full-length support* of constraint c is a set of literals containing exactly one literal for each variable in scope(c), such that c is satisfied by the assignment represented by these literals.

A *short support* is a support containing at most one literal for each variable in scope(c). As a motivating example for short supports, consider the lexicographic ordering constraint $\leq_{lex}$ on tuples. We can often know this is true just based on examining a small number of the variables in the constraint. Consider $\langle x_1, x_2, x_3\rangle \leq_{lex} \langle x_4, x_5, x_6\rangle$ where variables $x_1, \ldots x_6$ each have initial domain $D_s = \{1, 2, 3\}$. A full-length support is necessarily size 6: e.g. $\{x_1 \mapsto 2, x_2 \mapsto 2, x_2 \mapsto 2, x_4 \mapsto 2, x_5 \mapsto 2, x_6 \mapsto 2\}$ is a correct full-length support since $\langle 2, 2, 2\rangle \leq_{lex} \langle 2, 2, 2\rangle$. In contrast, the set $\{x_1 \mapsto 1, x_4 \mapsto 2\}$ is a correct short support even though it contains only two of the six variables: it is necessarily true that $\langle 1, *, *\rangle \leq_{lex} \langle 2, *, *\rangle$, whatever replaces the stars. Short supports can be of variable lengths as needed. For example, the short support $\{x_1 \mapsto 2, x_2 \mapsto 2, x_4 \mapsto 2, x_5 \mapsto 3\}$ is a correct short support of size 4, but no

literal can be removed from it without leaving at least one extension to a set of literals breaking the constraint. Following [19] we formally define short support as follows.

Definition 1 *[Short support] A **short support** S for constraint c and domains D_s is a set of literals $x \mapsto v$ such that $x \in \mathrm{scope}(c)$, $x \mapsto v$ is valid w.r.t D_s, x occurs only once in S, and every superset of S that contains one valid (w.r.t D_s) literal for each variable in $\mathrm{scope}(c)$ is a full-length support.*[1]

Note from the definition that any full-length support is also a short support. In the example the set $\{x_1 \mapsto 2, x_2 \mapsto 2, x_2 \mapsto 2, x_4 \mapsto 2, x_5 \mapsto 2, x_6 \mapsto 2\}$ is a short support and indeed no literal can be omitted to give another short support. In some cases even an empty set can be a short support. Suppose we change the motivating example so that $D_s(x_1) = \{0\}$ and other domains are unchanged. All valid assignments satisfy the lexicographic constraint since the only value of x_1 is 0 and $\langle 0, *, * \rangle \leq_{lex} \langle *, *, * \rangle$, so the empty set is a correct short support.

3 Encoding Table Constraints into SAT

Our encoding of constraint problems into SAT follows that which we have previously used and reported on [21]. When encoding a CSP variable, SAVILE ROW provides SAT literals for facts about the variable: $[x = a]$, $[x \neq a]$, $[x \leq a]$ and $[x > a]$ for a CSP variable x and value a. On all benchmarks used here, CSP variables are encoded in two ways. A variable with domain size 2 is represented with a single SAT variable. For variables with larger domains we have one SAT variable representing $[x = a]$ for each value $a \in D_s(x)$, and one SAT variable for each $[x \leq a]$ except $[x \leq \max(D_s(x))]$ that would always be true. Also, $[x = \max(D_s(x))] \leftrightarrow \neg[x \leq \max(D_s(x)) - 1]$ saving one more SAT variable. If we have a literal, e.g. $[x \leq a]$, where $a \notin D_s(x)$, then the literal is mapped as appropriate to True, False or an equivalent literal, e.g. $[x \leq b]$ for $b = \max(\{i \in D_s(x) | i < a\})$. The encoding has $2|D_s(x)| - 2$ SAT variables and consistency among them is maintained by the following clause set (sometimes called the ladder encoding [10]).

$$\forall a \in D_s(x). \quad [x = a] \leftrightarrow ([x \leq a] \wedge \neg[x \leq a + 1]) \quad \wedge \quad [x \leq a - 1] \rightarrow [x \leq a]$$

The only constraint we consider in this paper is the table constraint. This can be used to encode arbitrary constraints extensionally. The table constraint is very important in constraint programming, for constraints where no convenient expression in terms of simpler constraints is available. Suppose we have a constraint C on variables $x_1 \ldots x_r$ represented as a table of satisfying tuples, each of which is valid w.r.t. initial domains. Bacchus presented an encoding of table constraints [3]. Each satisfying tuple τ_i (where $i \in \{1 \ldots m\}$) is represented with an auxiliary SAT variable t_i. The first clause set

[1] The set of short supports depends on the domains D_s. We always use the initial domains. Elsewhere, short supports are generated using the current domains D but these sets are not necessarily short supports after backtracking [18,19]. A support of either type is valid iff all literals in it are valid.

ensures that each t_i becomes *false* when the tuple τ_i becomes invalid (i.e. a value in τ_i has been removed).

$$\forall i \in \{1 \dots m\}. \quad \forall j \in \{1 \dots r\}. \quad ([x_j = \tau_i[j]] \vee \neg t_i)$$

The second clause set states that each domain value of variables $x_1 \dots x_r$ must be supported by a valid tuple.

$$\forall i \in \{1 \dots r\}. \quad \forall a \in D(x_i). \quad ([x_i \neq a] \vee \bigvee t_j \quad \text{where} \quad \tau_j[i] = a)$$

Unit propagation (UP) applied to these clauses firstly removes from consideration any invalid tuple τ_i by setting t_i to *false*, then removes any domain value a of variable x_i (by setting $[x_i \neq a]$) where no remaining tuples support the value. Thus UP (re-)establishes GAC. Bacchus observed that the encoding has size $O(mr)$ which is linear in the size of the table representation of the constraint and applying UP has the same time complexity as a generic GAC propagator.

4 Short Support Encodings of Arbitrary Constraints

The idea of short support has already been successfully applied in constraint propagators [18,19,12]. Short support is defined above (Definition 1). Our contribution here is to exploit short supports in a new encoding that is smaller and more efficient than the Bacchus encoding while keeping the property that unit propagation establishes GAC. We assume that we already have a short support set for the constraint we wish to encode. It is often straightforward to construct a short support set for a constraint. Otherwise, an automated approach may be used such as the Greedy-Compress algorithm [12] that takes the table of full-length supporting tuples and compresses them to a short support set. It is possible that no short supports are available: in this case our encoding will provide neither harm nor benefit as it is equivalent to the Bacchus encoding.

The encoding for constraint C on variables $x_1 \dots x_r$ is as follows. Each short support σ_i ($i \in \{1 \dots m\}$) is represented with an auxiliary SAT variable s_i. The first clause set ensures that s_i is *false* when σ_i contains a literal that is invalid.

$$\forall i \in \{1 \dots m\}. \quad \forall (x_j \mapsto a) \in \sigma_i. \quad ([x_j = a] \vee \neg s_i)$$

The second clause set states that each literal $(x_i \mapsto a)$ of variables $x_1 \dots x_r$ must be supported by a valid short support, either *explicitly* (where the short support simply contains $(x_i \mapsto a)$) or *implicitly* (where the short support contains no literal of the variable x_i, meaning x_i may take any value).

$$\begin{aligned}&\forall i \in \{1 \dots r\}. \, \forall a \in D_s(x_i).\\ &\quad ([x_i \neq a] \vee \bigvee s_j \text{ where } (x_i \mapsto a) \in \sigma_j \text{ or } \forall b.(x_i \mapsto b) \notin \sigma_j)\end{aligned}$$

The two clause sets are sufficient for unit propagation to establish GAC on the constraint: the first clause set removes from consideration any short support that is invalid by setting the relevant s_i to *false*, and the second prunes any values that have no remaining short supports of either type (explicit or implicit).

This encoding has the property that the auxiliary variables s_i may not be uniquely determined when all SAT variables representing CSP variables have been assigned. This occurs when more than one short support is valid w.r.t. the CSP assignment. In this case at least one of the corresponding s_i must be true, but otherwise their values float freely. The free variables may cause additional search, and would cause a problem if we wished to count solutions. We obtain a second encoding without this issue by including the following additional clause set, which sets an s_i variable to true when all literals in σ_i are set true.

$$\forall i \in \{1 \dots m\}. \quad \left(\bigvee [x_j \neq a] \quad \text{where} \quad (x_j \mapsto a) \in \sigma_i\right) \vee s_i$$

Compared to the full-length table encoding, the short support encoding has fewer auxiliary variables, each representing a smaller conjunction of literals of the primary SAT variables. This is likely to be beneficial for conflict learning, facilitating more general and reusable explanations for conflicts.

We will refer to the short support encoding without the optional clause set as ShortTableSAT, and with the optional clause set as ShortTableSAT+.

5 Experimental Evaluation

To show the potential benefit of encoding using short supports, we evaluated our encodings of them on a number of problem classes. These are not intended to be an exhaustive or representative sample of possible problems, but a set of instances where short supports are available and thus show the potential benefit of our encodings. The instances we study are drawn from three general categories.

5.1 Case Study 1: Rectangle Packing

The rectangle packing problem [25] (with parameters n, *width* and *height*) consists of packing all squares from size 1×1 to $n \times n$ into the rectangle of size $width \times height$. This is modelled as follows: we have variables $x_1 \dots x_n$ and $y_1 \dots y_n$, where (x_i, y_i) represents the Cartesian coordinates of the lower-left corner of the $i \times i$ square. Domains of x_i variables are $\{0 \dots width - i\}$, and for y_i variables are $\{0 \dots height - i\}$. The only type of constraint is non-overlap of squares i and j: $(x_i + i \leq x_j) \vee (x_j + j \leq x_i) \vee (y_i + i \leq y_j) \vee (y_j + j \leq y_i)$. The domains of x_n and y_n are reduced to break flip symmetries [25]. The short supports of the non-overlap constraints are all of length two. Each short support satisfies one of the four disjuncts, thus satisfying the constraint. In a given instance, each constraint has a distinct short supports table because the constants i and j differ.

We compared the full length table encoding to both short table encodings on a set of instances taken from Jefferson and Nightingale [12] in addition to some generated ones. We generated two sets of instances: some small instances for all combinations of values for $n \in \{2 \dots 6\}$, $width \in \{10, 15, 20, 25\}$, and $height \in \{10, 15, 20, 25\}$; and some larger instances for all combinations of values for $n \in \{2, 4 \dots 30\}$, $width \in \{20, 25, 30, 35\}$, and $height \in \{20, 25, 30, 35\}$. For both of these sets we only kept

those instances where $width < height$ and also filtered out those which were trivially unsatisfiable due to area constraints.

5.2 Case Study 2: The Oscillating Life Problem and variants thereof

We consider the problem of maximum density oscillators (repeating patterns) in John Conway's Game of Life. We consider this and three variants. Immigration has two *alive* states. When a cell becomes alive, it takes the state of the majority of the 3 neighbouring live cells that caused it to become alive. Otherwise the rules of Immigration are the same as those of Life. Quadlife has four *alive* states. When a cell becomes alive, it takes the state of the majority of the 3 neighbouring live cells which caused it to become alive, unless all 3 neighbours have different colours in which case it takes the colour which none of its neighbours have. Apart from this the rules are the same as Life. Finally Brian's Brain has three states: *dead*, *alive* and *dying*. If a cell is *dead* and has exactly two *alive* (not dying) neighbours, it will become *alive*, otherwise it remains *dead*. If a cell is *alive*, it becomes *dying* after one time step. If a cell is *dying*, it becomes *dead* after one time step. We use an $n \times n$ grid with t time steps, for all pairs of values (n, t) where $n \in \{3 \dots 7\}$ and $t \in \{2 \dots 6\}$, giving 25 instances.

We use the problem and constraint model as described by Gent et al. [9]. For all four problems, we make one change: we minimise the occurrences of the value 0 (dead) in all layers. For Immigration, Quadlife and Brian's Brain we also add extra domain values for each additional state. For each cell and each time step, a single constraint links the cell and its eight neighbours to the same cell in the next time step. Therefore the constraints have arity 10. Short supports arise from sums in the rules, e.g. a live cell with more than three live neighbours will die: if the current cell is alive, any four neighbours are alive, and the next cell is dead then the constraint is satisfied and we have a short support of length 6.

5.3 Case Study 3: The Antichain Problem

The antichain problem is to find a set of multisets under some conditions [11]. Representing a multiset as a vector of integers (giving the cardinality of each possible value), we find a set of size n of vectors of length l, containing integers from the set $\{0 \dots d-1\}$. For each pair of vectors v_1, v_2, there must exist an index i where $v_1[i] < v_2[i]$ and a second index j where $v_1[j] > v_2[j]$. The problem is modelled with a two-dimensional matrix A with size n by l. Each pair of vectors is linked by a single constraint capturing both the $<$ and $>$ requirements, with scope size $2l$. The constraint linking any two vectors has short supports of length four. We compared the full length table encoding to both short table encodings on a set of 50 instances that includes all instances used by Jefferson et al [11].

5.4 Experimental Results

For a SAT solver we used the SAT'14 Competition version of Lingeling [4] (version ayv 86bf266b9332599f1b876e28a02fe8427aeaa2db). Each instance was solved 5

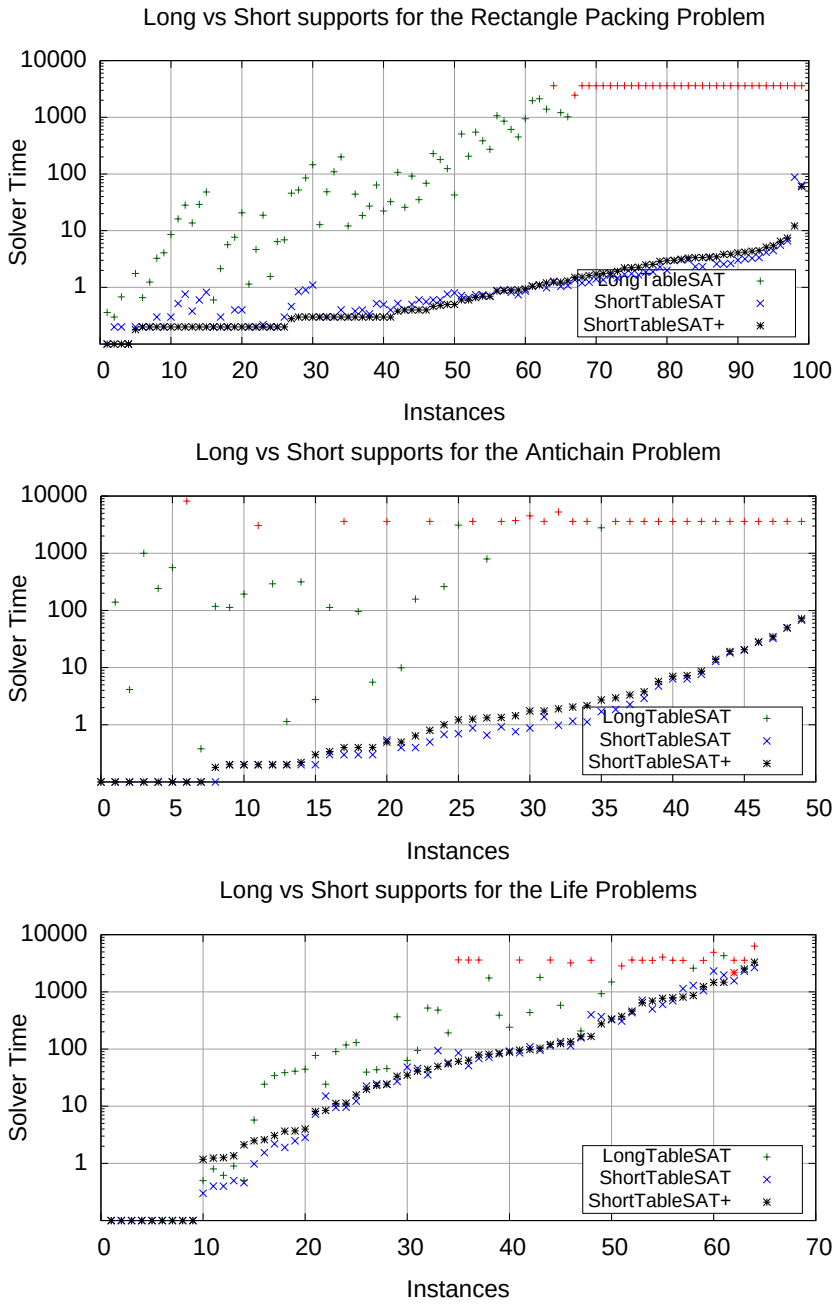

Fig. 1: Solver run times for the six problem classes

times, with random seeds changing from 1 to 5. We report the median of the five runtimes reported by Lingeling. Experiments were performed with 32 processes in parallel on a 32-core AMD Opteron 6272 at 2.1 GHz with 256 GB RAM. We set a limit of 1 hour for each Lingeling process. Results are in Figure 1. The x axis shows instances, ordered by increasing run time of ShortTableSAT+. Each position on the x axis represents the same instance within a plot. The y axis shows run time in seconds of Lingeling. Run time of the SAVILE ROW translation process is ignored, except that nothing is plotted if SAVILE ROW overran its time or space limit. We do plot (in red) the points where Lingeling reported that it reached its time limit. In some cases Lingeling reported reaching the time limit but also reported a time substantially less or greater than 1 hour. We simply plotted the time Lingeling reported using a red point.

For the packing problem (Figure 1, top), we see that we benefit greatly from use of short supports. There are many instances where LongTableSAT is unable to solve the instance, but both short table encodings are. On most other instances both short support methods are at least one and often several orders of magnitude faster. There is no clear preference between the two short encodings, with both methods faster on some instances, although ShortTableSAT+ is typically faster on the instances which can be solved fastest. We see in the antichain problem (Figure 1, middle) that again short supports provide improved search performance compared to LongTableSAT, by orders of magnitude. In this case it seems that ShortTableSAT is the better of the two short table encodings. For the Life, Immigration, Brian's Brain and Quadlife problem classes (Figure 1, bottom), we see that short supports do improve search but to a much lesser degree than in the previous cases. There are a small number of cases where LongTableSAT beats one of the short methods. However, using short tables is still much faster in most cases.

Our results show that the use of short supports in a SAT encoding can greatly improve solving performance over the use of full length table constraints.

6 Conclusions

Encoding to SAT and solving with a modern CDCL SAT solver is a very effective way to solve difficult finite-domain constraint problems. We have studied the encoding of table constraints, and proposed two new encodings based on the idea of *short supports*. These improve upon an existing encoding in both size and solving efficiency. In our experiments, the new encoding is consistently faster, frequently by over 10 times and in some cases by over 1000 times.

Acknowledgements We would like to thank the EPSRC for funding this work through grants EP/H004092/1, EP/K015745/1, and EP/M003728/1. In addition, Dr Jefferson is funded by a Royal Society University Research Fellowship.

References

1. Abío, I., Mayer-Eichberger, V., Stuckey, P.J.: Principles and Practice of Constraint Programming: 21st International Conference, CP 2015, Cork, Ireland, August 31 – September 4,

2015, Proceedings, chap. Encoding Linear Constraints with Implication Chains to CNF, pp. 3–11. Springer International Publishing, Cham (2015), http://dx.doi.org/10.1007/978-3-319-23219-5_1
2. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality networks: a theoretical and empirical study. Constraints 16(2), 195–221 (2011), http://dx.doi.org/10.1007/s10601-010-9105-0
3. Bacchus, F.: GAC via unit propagation. In: Principles and Practice of Constraint Programming (CP 2007). pp. 133–147 (2007)
4. Biere, A.: Lingeling, plingeling and treengeling entering the sat competition 2013. Proceedings of SAT Competition pp. 51–52 (2013)
5. Biere, A., Heule, M., van Maaren, H.: Handbook of Satisfiability, vol. 185. IOS Press (2009)
6. Brain, M., Hadarean, L., Kroening, D., Martins, R.: Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings, chap. Automatic Generation of Propagation Complete SAT Encodings, pp. 536–556. Springer Berlin Heidelberg, Berlin, Heidelberg (2016), http://dx.doi.org/10.1007/978-3-662-49122-5_26
7. Cheng, K.C.K., Yap, R.H.C.: An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. Constraints 15(2), 265–304 (2010)
8. Eén, N., Sörensson, N.: Translating pseudo-boolean constraints into SAT. JSAT 2(1-4), 1–26 (2006), http://jsat.ewi.tudelft.nl/content/volume2/JSAT2_1_Een.pdf
9. Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Generating special-purpose stateless propagators for arbitrary constraints. In: Principles and Practice of Constraint Programming (CP 2010). pp. 206–220 (2010)
10. Gent, I.P., Nightingale, P.: A new encoding of alldifferent into sat. In: Proc. 3rd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems (ModRef 2004). pp. 95–110 (2004)
11. Jefferson, C., Moore, N., Nightingale, P., Petrie, K.E.: Implementing logical connectives in constraint programming. Artificial Intelligence 174, 1407–1429 (2010)
12. Jefferson, C., Nightingale, P.: Extending simple tabular reduction with short supports. In: Proceedings of 23nd International Joint Conference on Artificial Intelligence (IJCAI). pp. 573–579 (2013)
13. Katsirelos, G., Walsh, T.: A compression algorithm for large arity extensional constraints. In: Proceedings CP 2007. pp. 379–393 (2007)
14. Mairy, J.B., Deville, Y., Lecoutre, C.: The smart table constraint. In: Proceedings of Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2015). pp. 271–287 (2015)
15. Marques-Silva, J.: Practical applications of boolean satisfiability. In: 9th International Workshop on Discrete Event Systems (WODES 2008). pp. 74–80 (2008)
16. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th annual Design Automation Conference. pp. 530–535. ACM (2001)
17. Nightingale, P., Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I.: Automatically improving constraint models in Savile Row through associative-commutative common subexpression elimination. In: 20th International Conference on Principles and Practice of Constraint Programming (CP 2014). pp. 590–605. Springer (2014)
18. Nightingale, P., Gent, I.P., Jefferson, C., Miguel, I.: Exploiting short supports for generalised arc consistency for arbitrary constraints. In: Proceedings IJCAI 2011. pp. 623–628 (2011)
19. Nightingale, P., Gent, I.P., Jefferson, C., Miguel, I.: Short and long supports for constraint propagation. Journal of Artificial Intelligence Research 46, 1–45 (2013)

20. Nightingale, P., Rendl, A.: Essence' description (2016), arXiv:1601.02865 [cs.AI]
21. Nightingale, P., Spracklen, P., Miguel, I.: Automatically improving SAT encoding of constraint problems through common subexpression elimination in Savile Row. In: Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP 2015). pp. 330–340 (2015)
22. Régin, J.: Improving the expressiveness of table constraints. In: The 10th International Workshop on Constraint Modelling and Reformulation (ModRef 2011) (2011)
23. Shang, Y., Wah, B.W.: A discrete lagrangian-based global-search method for solving satisfiability problems. Journal of global optimization 12(1), 61–99 (1998)
24. Silva, J.P.M., Lynce, I.: Towards robust CNF encodings of cardinality constraints. In: Bessiere, C. (ed.) Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4741, pp. 483–497. Springer (2007), http://dx.doi.org/10.1007/978-3-540-74970-7_35
25. Simonis, H., O'Sullivan, B.: Search strategies for rectangle packing. In: Proceedings CP 2008. pp. 52–66 (2008)
26. Stuckey, P.J., Tack, G.: Minizinc with functions. In: Gomes, C.P., Sellmann, M. (eds.) Proceedings of 10th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2013). Lecture Notes in Computer Science, vol. 7874, pp. 268–283. Springer (2013)