Youssef Hamadi and Lucas Bordeaux (Eds.)

Integration of SAT and CP Techniques

First International Workshop
Nantes, France, September 25, 2006

Held in Conjunction with the 12th International Conference on
Principles and Practice of Constraint Programming (CP 2006)

Contents

Foreword

SAT and CP techniques are two problem solving technologies which share many similarities, and there is considerable interest in cross-fertilising these two areas. The techniques used in SAT (propagation, activity-based heuristics, conflict analysis, restarts, etc.) constitute a very successful combination which makes modern DPLL solvers robust enough to solve large real-life instances without the heavy tuning usually required by CP tools. Whether such techniques can help the CP community develop more robust and easier-to-use tools is an exciting question. One limitation of SAT, on the other hand, is that not all problems are effectively expressed in a Boolean format. This makes CP an appealing candidate for many applications, like software verification, where SAT is traditionally used but more expressive types of constraints would be more natural.

The goal of this first workshop on SAT and CP integration is to boost the discussions between the SAT and CP communities by encouraging submissions at the border of these two areas. We have selected eight high quality papers which we believe are addressing this goal.

We wish to thank all the authors who submitted papers to this workshop and the members of the program committee for their work in the organization of this event.

September 2006
Youssef Hamadi and Lucas Bordeaux

Organizing Committee

Youssef Hamadi and Lucas Bordeaux Microsoft Research, Cambridge, UK

Program Committee

Christian Bessiere, LIRMM, Montpellier, France
Lucas Bordeaux, Microsoft Research, UK
Ian P. Gent, University of St Andrews, UK
Youssef Hamadi, Microsoft Research, UK
Joao Marques-Silva, University of Southampton, UK
Madan Musuvathi , Microsoft Research, Redmond, USA
Robert Nieuwenhuis, UPC, Barcelona, Spain
Andreas Podelski, Max-Planck Institut, Germany
Lakdhar Saïs, CRIL, Lens, France
Karem A. Sakallah, University of Michigan, USA
Sathiamoorthy Subbarayan, ITU, Copenhagen, Denmark
Lintao Zhang, Microsoft Research, Silicon Valley, USA

Constraint-Based Subsearch in Dynamic Local Search for Lifted SAT Problems

Colin Quirke[1] and Steve Prestwich[2]

[1] Boole Centre for Research in Informatics,
University College, Cork, Ireland
c.quirke@4c.ucc.ie
[2] Cork Constraint Computation Centre
Department of Computer Science, University College, Cork, Ireland
s.prestwich@cs.ucc.ie

Abstract. Many very large SAT problems can be more naturally expressed by quantification over variables, or "lifting". We explore implementation, heuristic and modelling issues in the use of local search on lifted SAT models. Firstly, adapting existing local search algorithms to lifted models creates overheads that limit the practicality of lifting, and we design a new form of dynamic local search for lifted models. Secondly, finding a violated clause in a lifted model is an NP-complete problem called "subsearch", and we show that subsearch benefits from advanced constraint techniques. Thirdly, lifting provides the opportunity for using SAT models that would normally be ignored because of their poor space complexities, and we use alternative SAT-encodings of a constraint problem to show that such lifted models can give superior results.

1 Introduction

In this paper we address the problem of solving very large Boolean Satisfiability (SAT) problems. We show two links between Constraint Satisfaction Problems (CSPs) and SAT: (i) large CSPs can be modelled as large SAT problems, and solved using a SAT algorithm capable of handling such problems; (ii) Constraint Programming (CP) techniques can boost the performance of a SAT solver aimed at large problems.

An interesting technique for handling large SAT problems is *lifting* [2, 7]: replacing a large set of related clauses by a single formula quantified over variables, then redesigning a search algorithm to operate on these formulae. This is impractical for random problems which (by definition) cannot be lifted, but structured problems may produce exponentially smaller formulae. Both complete and incomplete search algorithms may be adapted to lifted SAT models.

Often, when modelling a structured problem in the SAT domain, we develop lifted formulae such as

$$\forall a, b, c. \quad \neg X(a, b) \vee Y(b, c)$$

where a, b and c are finite domain, integer variables and $X()$ and $Y()$ are boolean predicates. The formulae are then easily *grounded* into propositional logic by enumerating the quantification and replacing each predicate with a boolean variable. These ground clauses can then be solved by existing SAT solvers.

Lifted formulae can produce very large sets of ground clauses, so it is desirable to work with the model in its lifted form. To achieve this we delay the grounding of the lifted model until it is specifically required by a search algorithm. In this way we avoid the need to store the complete ground model at any one time, and only concern ourselves with the portion which is currently interesting to the search. This procedure requires us to use advanced search techniques that are separate to the search for a solution, and so is labelled *subsearch*. This subsearch can be naturally expressed as a constraint satisfaction problem and solved using constraint techniques. Lifting, therefore, is an application of constraint techniques to solve large SAT problems.

Here we consider only local search algorithms, as these are often superior on large problems. We shall explore several issues:

- **Search heuristics.** Previous lifted SAT algorithms [2, 7] have been direct adaptations of existing search algorithms, which is a natural approach to take. However, we shall show that lifting current local search algorithms has several disadvantages: they create large runtime overheads, they may consume a great deal of memory, and some of the best heuristics cannot be applied to lifted models.
- **Implementation techniques.** The problem of locating clauses with certain properties (for example the property of violation), which must be solved by any search algorithm, is an NP-complete problem [2]. This *subsearch* problem is therefore a candidate for solution by constraint-based methods. We shall investigate the use of forward checking vs generalised arc consistency, and conflict-based backjumping vs chronological backtracking.
- **Modelling.** What impact does lifting have on our choice of SAT model for a problem? When faced with a new problem to solve by SAT methods, we often choose a model with good space complexity. But given the option of lifting, we are free to consider SAT-encodings that would otherwise be impractical. We use a constraint problem to show that such models may in fact give superior results.

This paper extends the work of [11] in several directions.

2 Dynamic local search for lifted models

Some local search algorithms fall into the category of random walks, which are a significant advance over earlier successful algorithms such as GSAT [15]. The best-known such algorithm is Walksat [5, 14] which has a number of variants. Walksat/G randomly selects a violated clause then flips the variable (reassigns it from true to false or vice-versa) that minimizes the total number of violations. Walksat/B selects flips that incur the fewest breaks (non-violated clauses that

would be violated by the flip). Both select a random variable in the clause (a random walk move) with probability p (the noise parameter). Walksat/SKC (Selman-Kautz-Cohen) is a version of B that allows freebies to override the random walk heuristic. Freebies are flips that incur no breaks, and if at least one freebie is possible from a violated clause then it is always selected. For every candidate variable in a violated clause the *break count* is the number of clauses that would be newly violated by a flip, while the *make count* is the number of clauses that wold be newly satisfied. Both SKC and B use a break count for each candidate variable, and G uses a combination of make count and break count to determine the flip with minimal violation.

Some problems defeat random walk algorithms, but are solved quite easily by an alternative form of local search based on clause weighting. These algorithms modify the objective function during search. They attach a weight to each clause and minimize the sum of the weights of the violations. The weights are varied dynamically, making it unlikely for the search to be trapped in local minima. Clauses that are frequently violated tend to be assigned higher weights. An early SAT algorithm of this form was Breakout [6] which increments violated clause weights at local minima. The Discrete Lagrangian Method (DLM) [16] periodically smooths the weights to reduce the effects of out-of-date local minima, and is based on the Operations Research technique of Lagrangian relaxation.

We would like to adapt the best current local search algorithms to lifted models. Previous lifted SAT algorithms [2, 7] have been direct adaptations of existing search algorithms, which is a natural approach. However, we claim that most of the best known local search algorithms are unsuitable for lifting, for several reasons:

- Current algorithms typically maintain a set of currently violated clauses, which is used to guide local moves. But for very large problems the set of violated clauses may also be very large. Though in practice it often remains manageably small it is not guaranteed to do so for all problems, especially in early phases of the search when it is far from any solution. In fact it is reported in [2] that a lifted solver ran out of memory on a lifted model corresponding to billions of clauses.
- Most SAT local search algorithms try to minimise the number of violated clauses, or in some cases to minimise break counts. The former type of algorithm also implicitly maintain break counts, along with make counts. We show that algorithms maintaining break counts incur large overheads on lifted problems. We propose instead to develop an algorithm, with consideration for the underlying constraint technology.
- Dynamic local search algorithms currently give the best local search results on many benchmarks. However, most such algorithms are based on clause weighting, which is impractical in a lifted context because individual clauses are not explicitly represented.

These drawbacks may partially explain why lifting has not taken off in the SAT community. We shall develop a local search algorithm that does not maintain a

set of violated clauses, is not based on the use of break counts, and dynamically updates its search heuristics without weighting clauses.

2.1 Walksat with make counts

There is an asymmetry in make and break counts: though the latter are expensive to compute by subsearch, the former are not. Assuming that the counts are known for a given problem state, consider updating the make and break counts having chosen a variable to flip. To update the make count we need to check each currently violated clause and decrement the make count for the other variables in these clauses. We may then flip the variable and again check the newly violated clauses and increment the make count for each variable they contain. To update the break count we need to check each satisfied clause and increment the break count of any variable which will become the only satisfying variable in that clause. We may then flip the variable and check each satisfied clause. If any clause is newly satisfied we increment the flipped variable's break count. If any clause is now satisfied by our flipped variable and one other variable we decrement the latter variable's break count. As the set of violated clauses is typically much smaller than the set of satisfied clauses, updating make counts is a much cheaper procedure.

Break counts are implicitly used in Walksat/G and other algorithms, besides the obvious variants such as B and SKC. Can we base a Walksat-like algorithm on make counts alone? As far as we know, no such algorithm has been reported in the literature. We propose an algorithm called Walksat/M that uses make counts to guide search (Algorithm 2.1). The subsearch in this algorithm is required at lines 7 and 9. The subsearch at line 7 for a violated clause is required by all of the aforementioned, random walk algorithms. The subsearch for make counts at Line 9 is where SKC and B use subsearch for break counts and G requires subsearch for both make and break counts. Details on constraint-based subsearch appear in Section 3.

Before lifting this algorithm we evaluate its potential using a ground version. We compare the performance of Walksat/M with Walksat/SKC on randomly generated problems from the phase transition with 50 variables (uf_50), 100 variables (uf_100) and 200 variables (uf_200). The sets uf_50 and uf_100 both contain 1000 instances, while set uf_200 contains 100 instances. Each instance is solved 1000 times by both algorithms with approximately optimal noise settings and the median steps to solution for each set is presented in Figure 1. Walksat/SKC is clearly superior on these problems, but Walksat/M gives quite good results. Scatter plots comparing the algorithms on each individual instance of uf_100 and uf_200 are presented in Figure 2. These plots record the median search cost of both algorithms on each instance.

We also compare the algorithms on structured problems. We use Blocks World (bw) planning and the All Interval Series (ais) from SATLib[1]. Each

[1] `http://www.cs.ubc.ca/~hoos/SATLIB`

Algorithm 2.1 Walksat/M

```
1: for i := 1 to MAX-TRIES do
2:    P := a randomly generated truth assignment
3:    for j := 1 to MAX-FLIPS do
4:       if P is a solution then
5:          return P
6:       else
7:          c := a randomly selected violated clause
8:          for all literal l ∈ c do
9:             M(l) = number of clauses that become satisfied if l flipped
10:         m = maximum value of M(l)
11:         L_m = {l|M(l) = m}
12:         if with probability p then
13:            flip a random literal in c
14:         else
15:            flip a random literal in L_m
16: return failure
```

instance is solved 100 times at approximately optimal noise settings and median search steps for each instance are presented in Figure 1. Walksat/SKC, again, outperforms Walksat/M, which fails to solve larger instances in a reasonable time. While initially promising, it is clear that Walksat/M is not powerful enough to challenge Walksat/SKC on these problems and so we seek to improve its performance using techniques from Dynamic local search.

Problem	Walksat/SKC	Walksat/M
uf_50	408	1101
uf_100	2226	6459
uf_200	13514	41288
bw_large.a	10364	27161
bw_large.b	416550	1292778
bw_large.c	9718052	—
ais6	891	4159
ais8	19306	194077
ais10	106752	5065650
ais12	1219293	—

Fig. 1. Comparison of median search cost for Walksat/SKC and Walksat/M on SAT benchmarks

2.2 Dynamic local search by variable weighting

Clause weighting techniques, from dynamic local search, are not suitable in a lifted setting, as we do not explicitly store clauses. A technique for variable

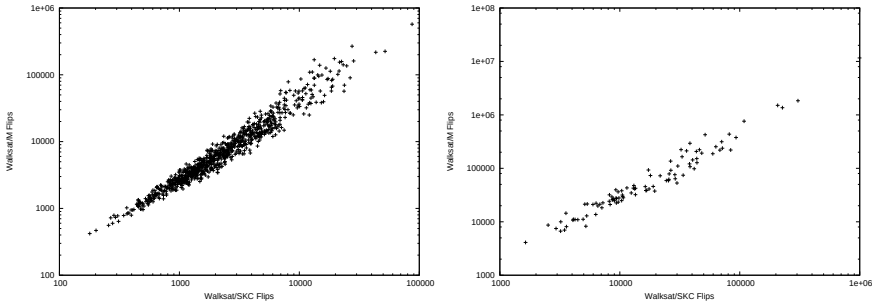


Fig. 2. Scatter plots of the median search cost (local moves) for both Walksat/SKC and Walksat/M for the 100 variable and 200 variable instances.

weighting is proposed in [8] and we adapt it here for use with Walksat/M. Each boolean variable is initially assigned a 0 weight. Every time a variable is flipped we increment its weight. When selecting a variable to flip from a violated clause we choose the variable with a maximum score $m_v + c(\mu - w_v)$, where m_v is the make count of v, w_v is its current weight, μ is the current mean weight, and c is a parameter used to tune the search.[2] This new variable selection system does not affect the subsearch components of Walksat/M and so little flip rate penalty is incurred from its use. We call this algorithm Walksat/MVW.

We compare the median flips to solution of Walksat/MVW to Walksat/SKC and Walksat/M on Blocks World Planning and All Interval Series. The results presented in Figure 3 show that MVW uses far fewer flips than M on all problems. MVW was better than SKC on Blocks World Planning and significantly improves performance on the All Interval Series. We propose Walksat/MVW as a promising algorithm for lifting.

Problem	Walksat/SKC	Walksat/M	Walksat/MVW
bw_large.a	10364	27161	11635
bw_large.b	416550	1292778	384832
bw_large.c	9718052	—	918675
ais6	891	4159	1180
ais8	19306	194077	24690
ais10	106752	5065650	204410
ais12	1219293	—	7384394

Fig. 3. Comparison of median search cost for SKC, M and MVW on SAT benchmarks

[2] The value of c is fine tuned for each instance by experimentation. Future work may examine automatically tuning this value.

3 Constraint-based subsearch

Subsearch refers to the explicit searches contained in many algorithms (both complete and incomplete). Examples include finding a violated clause in local search or finding a unit clause in unit propagation. These searches are usually ignored in a ground setting and may be implemented by linear search through a list or an indexing system. It is shown in [7] that when ground clauses are generated from lifted formulae, the resulting subsearch is NP-complete and so may benefit from more advanced search techniques. Dealing directly with the lifted formulae allows us to apply such techniques from constraint satisfaction.

We represent SAT models using formulae very similar to the *extended axioms* of [2]. We define Boolean predicates, each of which may have any number of integer terms, and constraints can be expressed on the terms. The integer terms in a lifted formula become integer variables in a constraint satisfaction problem. Constraints are derived from the Boolean predicates, by restricting integer values to those that make the predicate false. Examples can be seen in Section 4.2. If subsearch finds no solution (a violated clause) using a given formula then it does not contain a violated clause, so it tries the next formula, until either a violated clause is found or no formulae remain (in the latter case the SAT problem is solved). A complication is that each formula may encode different numbers of clauses. We therefore weight formula according to the number of clauses that they encode, and when searching for a violated clause the formula are selected in an order that is random but biased to select large formula first. Without this bias, clauses occurring in smaller formula are more likely to be selected. The bias helps to correct this.

When using subsearch to update make or break counts we need to find all solutions to our constraint model. In fact we frequently need to use more than one model as our boolean variable appears in multiple formulae. The asymmetry between make and break count means that make count requires using our models to search for violation while break count requires search for satisfaction. The constraint models to search for violation are typically tightly constrained with a small number of solutions. This leads to iteration over a smaller set: constraint models for satisfaction are typically loosely constrained with a far larger set of solutions. We argue, therefore, that make counts are a better candidate for subsearch in a lifted setting.

We implemented a prototype lifted Walksat/MVW in C++ using the EFC constraint library [3], which provides powerful CP techniques such as forward checking (FC), generalized arc-consistency (GAC) [4], conflict-directed backjumping (CBJ) [12] and dynamic variable ordering. The lifted algorithms in [7] did not use constraint propagation; one in [2] did but used static variable ordering. Our algorithm therefore uses more constraint-based techniques than previous lifted algorithms.

To evaluate the benefits of CP techniques in lifted local search we performed experiments to compare the flip rates of ground Walksat/SKC against lifted Schöning's algorithm and Walksat/MVW with FC, GAC, FC with CBJ, and GAC with CBJ, on the quaternary Golomb ruler model and a new model for

Balanced Incomplete Block Design, which is described in Section 4.2. The results in Figure 4 show that the ground and all the lifted versions of Schöning's algorithm (a simple local search algorithm described in [13]) perform similarly on small Golomb ruler problems, but that GAC has superior scaling as the problem size grows. More powerful constraint propagation leads to faster subsearch and therefore faster location of violated clauses. CBJ improved FC but not GAC, which is consistent with previous results in the CP literature. The results in Figure 5 show again that GAC beats FC, this time on MVW applied to Golomb ruler problems (FC and FCCBJ were very poor). The results in Figure 6 show similar results for Walksat/MVW on the Balanced Incomplete Block Design problem (using the model of Section 4).

We conclude that lifting with an appropriate constraint solver pays off on sufficiently large problems (crossover between Walksat and Schöning[GAC] occurs at approximately 600,000 clauses and for WalkSAT/M[GAC] at approximately 4,000,000 clauses). In future work we aim to improve both the implementation and the search heuristics.

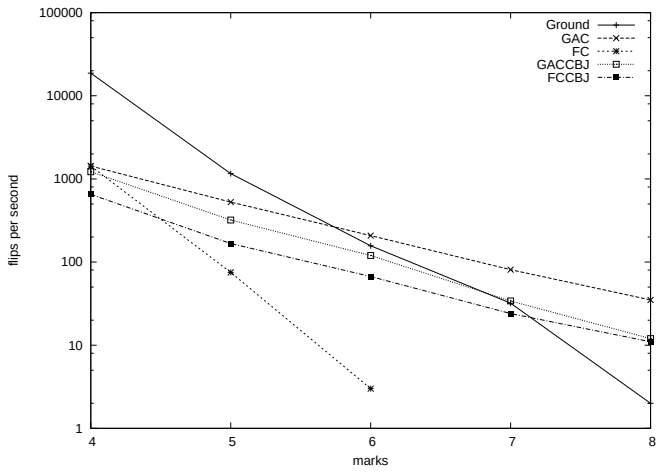


Fig. 4. Flip rate scaling on Golomb rulers: Schöning's algorithm

4 The impact of lifting on problem modelling

Finally we explore the effect of lifting on SAT modelling. Given the option of using a very large model, might this sometimes be better than using one with lower space complexity? We show that this does occur, using Balanced Incomplete Block Designs (BIBDs).

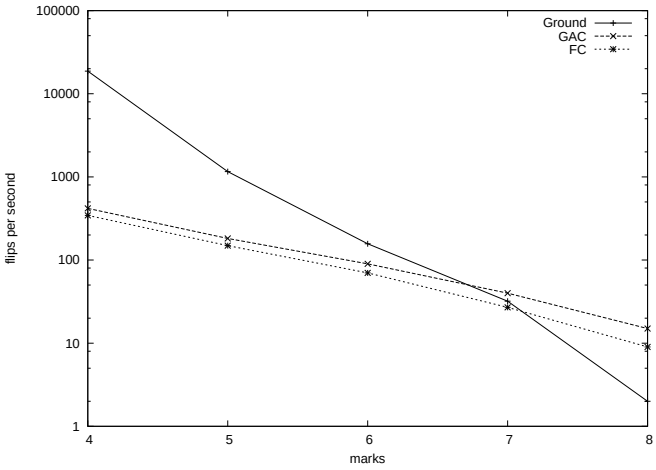


Fig. 5. Flip rate scaling on Golomb rulers: Walksat/MVW

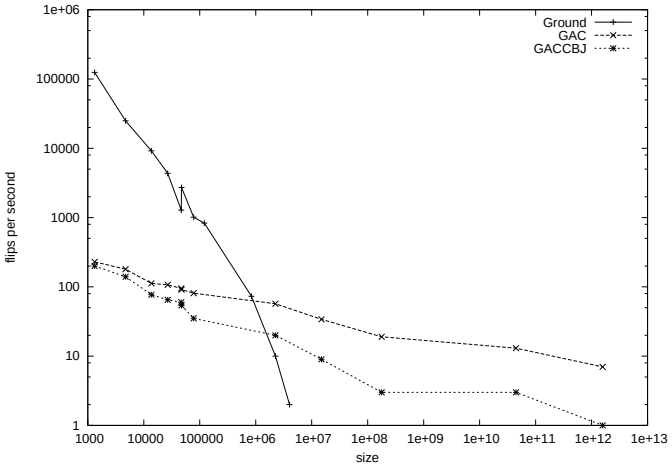


Fig. 6. Flip rate scaling on BIBDs: Walksat/MVW

4.1 A known model

A BIBD is an incidence system (v, k, r, b, λ) in which a set of v points is partitioned into b subsets (blocks) such that any two points determine λ blocks with k points in each block and each point is contained in r different blocks. The problem is in CSPLib(problem 28)[3]. The five parameters are not independent, but satisfy the two relations

$$vr = bk \tag{1}$$

$$\lambda(v-1) = r(k-1) \tag{2}$$

These conditions must be met but are not sufficient to prove the existence of a BIBD. The three parameters v,k and λ determine the remaining two as $r = \frac{\lambda(v-1)}{k-1}$ and $b = \frac{vr}{k}$. Hence a BIBD(v, b, r, k, λ) is often written as (v, k, λ) *design*. The notation 2-(v, k, λ) *design* is also used, since BIBDs are t-designs with $t = 2$.

4.2 A new model

BIBDs have been SAT-encoded before in [10], but with poor results. We now present a new model with higher space complexity. Note that such a model would normally be rejected without even being tested. This model is similar to the pseudo-boolean model presented in [9] and space complexity derives from the expansion of cardinality constraints. To encode the BIBD, with parameters (v, k, λ, r, b), into propositional logic, we define two sets of boolean variables.

$$oneAt_{i,j} \quad 1 \leqslant i \leqslant v, 1 \leqslant j \leqslant b$$

$$zeroAt_{i,j} \quad 1 \leqslant i \leqslant v, 1 \leqslant j \leqslant b$$

These represent the $v \times b$ incidence matrix M, with the meaning that if $oneAt_{1,2}$ is *true* there is a one in position $M_{[1,2]}$ and if $zeroAt_{2,3}$ is *true* there is a zero in position $M_{[2,3]}$.

We define a set of clauses to ensure that each element in the matrix is zero or one.

$$oneAt_{i,j} \vee zeroAt_{i,j} \quad 1 \leqslant i \leqslant v, 1 \leqslant j \leqslant b$$

$$\neg oneAt_{i,j} \vee \neg zeroAt_{i,j} \quad 1 \leqslant i \leqslant v, 1 \leqslant j \leqslant b$$

We define a set of clauses that state the maximum number of ones per row is r.

$$\neg oneAt_{i,x1} \vee \neg oneAt_{i,x2} \vee \ldots \vee \neg oneAt_{i,x(r+1)}$$
$$1 \leqslant x1 < x2 < \ldots < x(r+1) \leqslant b$$

We define a set of clauses that state the maximum number of zeros per row is $b - r$.

$$\neg zeroAt_{i,x1} \vee \neg zeroAt_{i,x2} \vee \ldots \vee \neg zeroAt_{i,x(b-r+1)}$$
$$1 \leqslant x1 < x2 < \ldots < x(b-r+1) \leqslant b$$

[3] `http://www.csplib.org`

These clauses ensure that each row contains exactly r ones. We do the same for columns.

$$\neg oneAt_{x1,j} \vee \neg oneAt_{x2,j} \vee \ldots \vee \neg oneAt_{x(k+1),j}$$
$$1 \leqslant x1 < x2 < \ldots < x(k+1) \leqslant v$$
$$\neg zeroAt_{x1,j} \vee \neg zeroAt_{x2,j} \vee \ldots \vee \neg zeroAt_{x(v-k+1),j}$$
$$1 \leqslant x1 < x2 < \ldots < x(v-k+1) \leqslant v$$

We define a set of clauses that state the maximum scalar product between any pair of rows is λ.

$$\neg oneAt_{i,x1} \vee \neg oneAt_{i',x1} \vee \ldots \vee \neg oneAt_{i,x(\lambda+1)} \vee \neg oneAt_{i',x(\lambda+1)}$$
$$i \neq i', 1 \leqslant x1 \leqslant \ldots \leqslant x(\lambda+1) \leqslant v \qquad (3)$$

4.3 Proof of model correctness

Clause (3) only constrains the scalar product between any pair of rows to be at most λ. This is sufficient to represent the problem according to the following argument.

Lemma 1. *In any $v \times b$ binary matrix with k ones per row and b ones per column the mean scalar product of row pairs is λ.*

Proof. Let $x_{i,i'}$ be the scalar product of rows i and i'. The mean value of all $x_{i,i'}$ is

$$\bar{x}_{i,i'} = \frac{\sum_{1 \leqslant i < i' \leqslant v} x_{i,i'}}{\binom{v}{2}} \qquad (4)$$

We have $\binom{k}{2}$ pairs of ones in any column. Totalled over all columns is $b\binom{k}{2}$ pairs of ones. Each pair of ones is contributes exactly 1 to the total value of all scalar products. This gives

$$\sum_{1 \leqslant i < i' \leqslant v} x_{i,i'} = b\binom{k}{2}$$

We can now rewrite equation(4) as

$$\overline{x_{i,i'}} = \frac{b\binom{k}{2}}{\binom{v}{2}}$$

Which simplifies to

$$\overline{x_{i,i'}} = \frac{bk(k-1)}{v(v-1)}$$

As $bk/v = r$ *from equation*(1)

$$\overline{x_{i,i'}} = \frac{r(k-1)}{v-1} = \lambda \qquad \textit{(from equation(2))}$$

The mean scalar product has to be λ and clause(3) states that no scalar product can be greater than λ. This implies that all scalar products must be exactly λ and so the model is correct.

4.4 Comparison of the grounded models

The model presented in [10] has a polynomial space complexity in (v, k, r, b, λ), while the model presented here has an exponential complexity of $\mathcal{O}(v\binom{b}{r} + b\binom{v}{k} + v\binom{b}{\lambda})$. We can still generate a ground model for smaller instances but this becomes unwieldy beyond a few million clauses. The largest model shown here would have 1.58×10^{12} clauses if ground but could easily be solved by our lifted solver.

4.5 Evaluation of the lifted model

We test the new model using both ground Walksat/SKC and lifted Walksat/MVW, and compare the median flips to solution with those reported in [10]. The results in Table 4.5 show that Walksat/SKC does much better on the new model and can solve more problems than the original in a reasonable time. However, its performance degrades as the memory requirement increases, and the ground model struggles to solve the larger problems. The lifted Walksat/MVW does even better on these models and can easily cope with the larger problems.

We believe these are the largest BIBDs found using SAT techniques. This illustrates an interesting result: that when space complexity is not an issue it might be better to choose an *exponentially larger* SAT model, despite the additional subsearch overhead this might cause. The compromises in model simplicity that we make to reduce space complexity might have a deleterious effect on search performance, but these can be avoided by using a lifted model.

5 Conclusion

We have addressed several practical issues that arise when modelling very large SAT problems. We devised a new dynamic local search algorithm, based on the properties of the underlying constraint problem, that has good heuristics and is more amenable to lifting than all other local search algorithms that we know of. We showed that CP techniques can greatly speed up subsearch, so that lifting becomes applicable to more problems. Finally, we showed that when SAT-encoding constraint problems, the best choice of model may be very different with lifting than without. These advances help to make SAT technology applicable to much larger problems.

In future work we hope to investigate ways of further improving subsearch, using techniques such as global constraints and possibly ideas from [1]. We also hope to extend the SAT solver to a CSP solver.

BIBD	Original SAT Model	New SAT Model	
(v, b, r, k, λ))	Walksat/SKC flips	Walksat/SKC	Walksat/MVW
7,7,3,3,1	51412	406	117
6,10,5,3,2	121259	637	248
9,12,4,3,1	6382697	3102	366
11,11,5,5,2	—	8383	6547
7,14,6,3,2	2028247	1754	339
13,13,4,4,1	—	3725	1216
8,14,7,4,3	—	7644	2385
10,15,6,4,2	—	33913	2959
9,18,8,4,3	—	36046	9386
6,20,10,3,4	—	—	362
7,21,9,3,3	—	—	453
9,24,8,3,2	—	—	664
7,28,12,3,4	—	—	509
6,30,15,3,6	—	—	602
7,35,15,3,5	—	—	606
6,40,20,3,8	—	—	752

Table 1. Comparison of results from [10] with the new model

Acknowledgement

This work was supported by the Boole Centre for Research in Informatics, University College, Cork, Ireland, and is based in part upon works supported by the Science Foundation Ireland under Grant No. 00/PI.1/C075. Many thanks to George Katsirelos, Department of Computer Science, University of Toronto, for implementing new EFC features at our request.

References

1. V. Chandru, J. N. Hooker, A. Shrivastava, G. Rago. A Partial Instantiation Based First Order Theorem Prover. *International Workshop on First Order Theorem Proving*, Vienna, 1998.
2. Satisfiability Algorithms and Finite Quantification. M. L. Ginsberg, A. J. Parkes. *Seventh International Conference Principles of Knowledge Representation and Reasoning*, Morgan Kaufmann, 2000, pp. 690–701.
3. G. Katsirelos. EFC, available at http://www.cs.toronto.edu/~gkatsi/efc/efc.html.
4. A. K. Mackworth. On Reading Sketch Maps. *Fifth International Joint Conference on Artificial Intelligence*, Kaufmann, 1977, pp. 598–606.
5. D. McAllester, B. Selman, H. Kautz. Evidence for Invariants in Local Search. *Fourteenth National Conference on Artificial Intelligence*, AAAI Press, 1997, pp.321–326
6. P. Morris. The Breakout Method for Escaping from Local Minima. *Proceedings of the Eleventh National Conference on Artificial Intelligence*, AAAI Press / MIT Press, 1993, pp. 40–45.
7. A. J. Parkes. Lifted Search Engines for Satisfiability. PhD dissertation, University of Oregon, 1999.

8. S. D. Prestwich. Random Walk With Continuously Smoothed Variable Weights. *Eighth International Conference on Theory and Applications of Satisfiability Testing, Lecture Notes in Computer Science* vol. 3569, Springer, 2005, pp. 203–215.
9. S. D. Prestwich. A Local Search Algorithm for Balanced Incomplete Block Designs. *Recent Advances in Constraints, Joint ERCIM/CologNet International Workshop on Constraint Solving and Constraint Logic Programming, Lecture Notes in Artificial Intelligence* vol. 2627, Springer, 2003, pp. 132–143.
10. S. D. Prestwich. Balanced Incomplete Block Design as Satisfiability *Twelfth Irish Conference on Artificial Intelligence and Cognitive Science* 2001, pp. 189–198.
11. S. D. Prestwich, C. Quirke. Local Search for Very Large SAT Problems. Short paper, *Seventh International Conference on Theory and Applications of Satisfiability Testing*, Vancouver, BC, Canada, 2004.
12. P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence* vol. 9 no. 3, 1993, pp. 268–299.
13. U. Schöning. A Probabilistic Algorithm for k-SAT and Constraint Satisfaction Problems. *Fortieth Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, 1999, pp. 410–414.
14. B. Selman, H. Kautz, B. Cohen. Noise Strategies for Improving Local Search. *Proceedings of the Twelfth National Conference on Artificial Intelligence*, AAAI Press, 1994, pp. 337–343.
15. B. Selman, H. Levesque, D. Mitchell. A New Method for Solving Hard Satisfiability Problems. *Proceedings of the Tenth National Conference on Artificial Intelligence*, MIT Press 1992, pp. 440–446.
16. Z. Wu, B. W. Wah. An Efficient Global-Search Strategy in Discrete Lagrangian Methods for Solving Hard Satisfiability Problems. *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, AAAI Press, 2000, pp. 310–315.

Interpolant based Decision Procedure for Quantifier-Free Presburger Arithmetic

Shuvendu K. Lahiri[1] and Krishna K. Mehra[2]

[1] Microsoft Research
shuvendu@microsoft.com
[2] Department of Computer Science and Engineering,
Indian Institute of Technology, Kharagpur, India.
kmehra@iitkgp.ac.in

Abstract. Recently, off-the-shelf Boolean SAT solvers have been used to construct ground decision procedures for various theories, including Quantifier-Free Presburger (QFP) arithmetic. One such approach (often called the *eager* approach) is based on a satisfiability-preserving translation to a Boolean formula. Eager approaches are usually based on encoding integers as bit-vectors and suffer from the loss of structure and sometime very large size for the bit-vectors.
In this paper, we present a decision procedure for QFP that is based on alternately under and over-approximating a formula, where Boolean interpolants are used to compute the overapproximation. The novelty of the approach lies in using information from each phase (either underapproximation or overapproximation) to improve the other phase. Our preliminary experiments indicate that the algorithm consistently outperforms approaches based on eager and very lazy methods on a set of verification benchmarks.

1 Introduction

Decision procedures for quantifier-free theories are the cornerstones of many automated analysis tools for software and hardware. Software verification tools like SLAM [2] use decision procedures to perform automated predicate abstraction and refinement of programs; ESC-JAVA [12] uses decision procedures to discharge verification conditions in static analysis of programs. High-level hardware verification tools including UCLID [5] use decision procedures for checking first-order formulas arising from bounded model checking or invariant checking for models of microprocessor and cache coherence protocols.

The quantifier-free queries that arise in verification typically have a lot of Boolean structure in addition to theory constraints. This requires an interplay between a search algorithm to case split on the Boolean structure and a decision procedure for the ground theory. In recent years, several approaches have emerged to leverage the rapid improvements in Boolean Satisfiability (SAT) solving [22]. These techniques differ mainly in how closely the SAT solver interacts with theory reasoning. The *eager* approaches rely on translating the quantifier-formula to an equisatisfiable Boolean formula and use a SAT solver to solve the resultant Boolean formula [5, 32]. The (very) *lazy* techniques create a Boolean abstraction of the first-order formula, and refine it based on assignments that are inconsistent with the underlying theory [1, 3, 11]. Both these approaches treat

the SAT-solver as a black-box. Finally, other approaches augment the Davis, Putnam, Logeman and Loveland (DPLL) [8, 9] algorithm for the Boolean SAT solvers with theory specific reasoning [23, 4].[3]

Quantifier-free Presburger (QFP) arithmetic is the quantifier-free fragment of Presburger arithmetic [25], which deals with linear arithmetic constraints along with Boolean connectives. Kroening et al. [18] proposed a framework for integrating the Boolean and the theory reasoning for QFP, based on alternately under and over-approximating the input QFP formula. The algorithm searches for a satisfying solution in a sequence of increasingly large domains, lazily increasing the domain size when no satisfying solution is found in a smaller domain. The proof of unsatisfiability in a domain is used to construct an abstraction of the original formula. The abstractions are checked using a decision procedure for QFP. They illustrate that the procedure terminates and therefore constitutes a decision procedure for QFP.

In this work, we present an alternate implementation of the above framework that is based on Craig's *interpolants* [7, 26] to construct the abstraction, once a formula is unsatisfiable in a given domain. Unlike the approach in [18], our method has the advantage of not requiring a clausal representation of the input QFP formula to construct the abstraction. The algorithm also differs in several other respects: first, we do not require a complete decision procedure for QFP to check the abstraction, and secondly, we leverage the *learning* from the search in the smaller domains when moving to a larger domain. We compare our approach with Kroening et al.'s work in detail in Section 5.

At a high level, our algorithm works as follows: Consider a QFP formula ϕ. Given a domain D, it is possible to construct a Boolean formula ϕ_u whose satisfiability determines the existence of a satisfying solution for ϕ in the domain D. This Boolean formula underapproximates the original formula ϕ. If ϕ_u is unsatisfiable, then we use Boolean interpolant generation to construct a formula ϕ_o in QFP that is an overapproximation of ϕ (we defer the actual details of the procedure to Section 3). Intuitively, the overapproximation abstracts the reason why ϕ was unsatisfiable in the particular domain D, in terms of the constraints in ϕ. The overapproximation ϕ_o generated serves two purposes: first, if ϕ_o is unsatisfiable, then ϕ is unsatisfiable and secondly, we can generate some conflict clauses (tautologies in QFP) from ϕ_o that can be added to ϕ, to prune the search space for future underapproximations.

The theory of QFP has a *small-model* property for any formula ϕ in the theory, i.e., if ϕ is satisfiable, then it has a satisfying assignment in a finite domain D_{max} determined by the formula ϕ. This ensures that we can start with a small domain D and increase the size lazily until the maximum domain D_{max} is reached. The small-model property also allows us to use an incomplete decision procedure for QFP to check the satisfiability of ϕ_o. In our experience, the maximum domain size D_{max} is almost never reached.

We have implemented the algorithm in Zap theorem prover [33] and provide preliminary experimental evaluation of the approach on a set of verification benchmarks. The algorithm seems to consistently outperform an implementation based on purely eager encoding (that use the maximum domain size D_{max} to en-

[3] Although the latter techniques are also refered to as lazy techniques, we mostly refer to the very lazy techniques [1, 3, 11] that treat the SAT-solver as a black box as lazy in this paper.

code an equisatisfiable formula to SAT). It also outperforms an implementation of the very lazy Verifun [11] approach on these benchmarks.

The rest of the paper is organized as follows: In Section 2, we describe the background material including eager and lazy approaches for leveraging SAT solvers. We also describe basics of interpolants. In Section 3, we motivate the algorithm and present the details of the decision procedure for QFP. Section 4 describes the experimental evaluation of the approach. We describe related work in Section 5 and finally conclude. Appendix B presents a variation of the algorithm that does not require computing D_{max} for completeness.

2 Preliminaries

In this section, we provide some background on the logic QFP, eager and the (very) lazy approaches for solving QFP along with Boolean interpolants.

2.1 Quantifier-free Presburger (QFP) Arithmetic

Presburger arithmetic is the first-order theory of structure $\langle \mathbb{N}, 0, 1, +, \leq \rangle$, where $\mathbb{N}$ denote the set of natural numbers. Since every integer variable can be expressed as the difference of two natural numbers, we assume that the underlying domain is the set of integers $\mathbb{Z}$. Quantifier-free Presburger Arithmetic (QFP) is the quantifier-free fragment of Presburger arithmetic. Let X be a set of integer variables. An atomic formula (*atomic-formula*) in this theory (also refered to as a linear constraint) is an expression of the form:

$$\sum_i a_i * x_i \leq c,$$

where $x_i \in X$ and the coefficients a_i and c are constants in $\mathbb{Z}$. A *formula* in this QFP is a Boolean combination of atomic formulas:

$$\begin{aligned} \textit{formula} ::= & \texttt{true} \mid \texttt{false} \mid \textit{atomic-formula} \\ \mid & \ \textit{formula} \wedge \textit{formula} \mid \textit{formula} \vee \textit{formula} \mid \neg \textit{formula} \end{aligned}$$

Observe that the other relational operators $\{=, \neq, <, >, \geq\}$ can be expressed in QFP.

A formula ϕ in QFP is *satisfiable* if there is an assignment $\rho : X \rightarrow \mathbb{Z}$ that maps each $x_i \in X$ to an integer value, such that the evaluation of ϕ under ρ is `true`. A formula ϕ is unsatisfiable if there is no assignment ρ under which ϕ evaluates to `true`.

A *monome* is a conjunction of atomic formulas in QFP. Checking the satisfiability of a monome in QFP is NP-complete [24]. However, efficient algorithms based on branch-and-bound heuristics are implemented in various Integer Linear Programming (ILP) solvers like LP_SOLVE [20] and commercial tools like CPLEX [15] to solve this fragment. The complexity of checking the satisfiability of QFP formulas is no worse than checking satisfiability of a conjunction of atomic formulas. Previous studies have indicated that ILP solvers do not perform well for QFP formulas with a significant Boolean structure, that arise from verification problems [28, 18]. To circumvent this problem, two methods have been proposed in recent years to leverage backtracking search of modern Boolean satisfiability solvers. We describe the two approaches in the next section.

2.2 Lazy and Eager Methods for solving QFP

Lazy Methods The *lazy* methods [1, 27, 3] leverage the backtracking of modern Boolean Satisfiability (SAT) solvers to case-split on the Boolean structure in the QFP and use a ILP solver to check the satisfiability of a conjunction of linear constraints. The algorithms can be loosely described as follows:

1. The QFP formula ϕ is abstracted to a Boolean formula ϕ_a by replacing each atomic constraint with a Boolean variable.
2. The SAT solver enumerates satisfying solutions over ϕ_a and uses the ILP solver to validate the solution over the QFP theory.
3. If the satisfying solution is consistent with the QFP theory, the procedure returns satisfiable. Otherwise, the solver returns a conflict clause (a tautology in QFP) that rules out the current assignment. Typically, the literals that appear in the proof of unsatisfiability of the current assignment, constitute the conflict clause [3, 11]. This helps towards finding a "minimal" unsatisfiable core to rule out more than just the present satisfying assignment. The conflict clause is added to ϕ_a and Step 2 is repeated.

Eager Methods Eager methods (e.g. implemented in UCLID [19]) are based on translating a QFP formula ϕ to an *equisatisfiable* Boolean formula ϕ_{bool}, such that ϕ is satisfiable if and only if ϕ_{bool} is satisfiable. Since the problem of deciding the satisfiability of QFP is in NP, QFP enjoys a *small-model* property — a QFP formula ϕ is satisfiable over $\mathbb{Z}$ if and only if it is satisfiable over a finite domain $\mathbb{D} \subseteq \mathbb{Z}$. The measure $\log(\mathbb{D})$ denotes the number of Boolean variables to represent the domain $\mathbb{D}$.

Let m be the number of atomic formulas in ϕ and n be the number of variables in X. When the set of atomic constraints in ϕ is restricted to *difference logic* constraints (where the atomic formulas are restricted to $x_i - x_j \leq c$), the size of the domain $\log(\mathbb{D})$ is $O(\log(n) + \log(c_{max}))$, where c_{max} is the absolute value of largest constant c that appears in any of the constraints. Seshia and Bryant [28] show that for general linear constraints, $\log(\mathbb{D})$ is bound by

$$O(\log(n) + \log(m) + \log(c_{max}) + k * (\log(a_{max}) + \log(w))), \tag{1}$$

where a_{max} is the maximum absolute value of any coefficient, k is the number of non-difference atomic formulas in ϕ and w is the maximum number of variables in any linear constraint. When the number of non-difference constraints in ϕ is small, the size of $\log(\mathbb{D})$ is almost logarithmic in n and m.

The above bounds suggest that one can encode each variable $x \in X$ using $\log(\mathbb{D})$ Boolean variables and translate ϕ to a Boolean formula. The arithmetic operator $+$ can is encoded as an word adder, $*$ is implemented as a shift operator since only variables are multiplied with constants. Similarly, the relational operator $\leq$ is encoded as word comparator. The size of the resultant Boolean formula ϕ_{bool} incur a polynomial blowup (typically $\log(\mathbb{D})$) over ϕ. Finally, the resultant formula can be checked using any state-of-the-art SAT solvers. We refer to this encoding as *small-model* encoding of a first-order formula.

Comparison of the two methods In this section, we highlight the main weaknesses of the eager and the lazy approaches, that limit the scalability of the approaches:

The appeal of small-model based encoding to SAT lies in the fact that it provides only a polynomial blowup when translating a linear arithmetic formula to a Boolean formula. However, the blowup can be linear in the size in the size of the number of constraints and variables in the first-order formula, when the number of non-difference constraints is large. The small model size also explodes in the presence of large constants in the formula. More seriously, small-model encoding suffer from a loss of structure of the formula. For example, consider the simple formula:

$$\phi = x \leq y \wedge y \leq z \wedge z < x$$

There is a polynomial algorithm for deciding the satisfiability of such conjunction of linear arithmetic formulas, based on negative cycle detection [6]. However, converting to a Boolean formula introduces a lot of disjunctions resulting from the encoding of $\leq$ as a circuit. This results in SAT solver to perform a lot of case splits before detecting unsatisfiability.

The lazy approaches (e.g. Verifun [11]) suffer from the need to invoke a decision procedure for the first-order theory a very large number of times in the presence of a lot of Boolean structure (i.e. disjunctions) in the formula. Moreover, since the decision procedures take over only after the SAT solver has found a Boolean model, the monome handed to the theory can have a lot of theory literals. When the monome is unsatisfiable in the theory, if often happens because of very small subset of literals in the monome. Although the decision procedure can figure out the core reason for unsatisfiability (using the proof of unsatisfiability), the decision procedure is often overwhelmed with the size of the monome that it obtains from the SAT solver. This is particularly problematic for the theory of integer linear arithmetic, for which the decision procedures have exponential worst-case complexity.

2.3 Boolean Interpolants

Consider two satisfiable Boolean formulas A and B such that $A \wedge B = \texttt{false}$. Let V be the set of Boolean variables shared by both A and B. An (Boolean) *interpolant* [7] of the pair of formulas (A, B) is a Boolean formula I, such that:

1. $A \Rightarrow I$,
2. $I \wedge B = \texttt{false}$, and
3. The set of variables in I is a subset of V.

Pudlak [26] showed that given the proof of unsatisfiability of $A \wedge B$, an interpolant I can be obtained in time linear to the size of the proof. A description of the algorithm when both A and B are present in conjunctive normal form (CNF) has been described by McMillan [21]. The motivation for using the interpolant I of (A, B) is usually two-fold: (a) it is an abstraction of A that is sufficient to prove the unsatisfiability with B and (b) it is over the common variables of A and B.

3 Interpolant-based Decision Procedure for QFP

In this section, we describe an algorithm to decide the satisfiability of a QFP formula ϕ. The algorithm alternates between two phases that check the satisfiability of an underapproximation and overapproximation of ϕ, respectively. In

the underapproximation phase, the algorithm creates a formula ϕ_u that is an underapproximation of ϕ by *restricting* the domain of each variable that appear in the formula. If the formula ϕ_u has a satisfying assignment within the small bounds, it reports satisfiable. Otherwise, it computes an abstraction ϕ_o of ϕ by using Boolean interpolants. The abstract formula ϕ_o is then checked for (un) satisfiability. If ϕ_o is unsatisfiable, then ϕ is unsatisfiable. Otherwise, we repeat the phases with an increased domain for the variables. The algorithm adds additional clauses that it discovers while checking ϕ_o to the formula ϕ, to speed up the underapproximation phase in the further iterations. We describe the algorithm in details in the next few paragraphs.

1. [**Input**]. Given a QFP formula ϕ.
2. [**Encoding**]. Construct the Boolean structure of ϕ by replacing an atomic formula e with a Boolean variable b_e in ϕ. The resultant formula ϕ_b will be referred to as the (Boolean) *skeleton* of ϕ. Let $atoms(()\phi)$ denote the set of atomic formulas in ϕ. We introduce a set of fresh variables $B = \{b_e \mid e \in atoms(\phi)\}$ and create a formula

$$\phi_{th} = \left(\bigwedge_{e \in atoms(\phi)} b_e \Leftrightarrow e \right),$$

   called the *theory* portion of ϕ. Finally, the formula representing the conjunction of the Boolean skeleton and the theory component is called ϕ_{bt}:

$$\phi_{bt} \doteq \phi_b \wedge \phi_{th}$$

3. [**Initialize**]. Compute the maximum model size D_{max} for each variable using Equation 1. The initial domain for each variable is restricted to D, i.e. each variable $v \in vars(\phi)$ takes values in $(-D, \ldots, D]$.
4. [**Boolean UNSAT**]. We first check if the skeleton of ϕ_{bt} is unsatisfiable using the SAT solver. If ϕ_b is unsatisfiable, the algorithm returns UNSATISFIABLE.
5. [**Underapproximation**]. We construct a Boolean formula for ϕ_{th} by using the small-model encoding technique described in Section 2.2. We use $BE(\phi_{th}, D)$ to denote the Boolean translation. Let the resultant Boolean formula be ϕ_u, an underapproximation of ϕ_{bt}:

$$\phi_u \doteq \phi_b \wedge BE(\phi_{th}, D)$$

   We check if ϕ_u is satisfiable using a SAT solver. If ϕ_u is satisfiable, the algorithm returns SATISFIABLE. If ϕ_u is unsatisfiable and $D \geq D_{max}$, we return UNSATISFIABLE.
6. [**Overapproximation**]. If ϕ_u is unsatisfiable, compute the Boolean interpolant ϕ_I of ϕ_b and $BE(\phi_{th}, D)$. We construct the formula ϕ_o which is an overapproximation of ϕ_{bt} as follows:

$$\phi_o \doteq \phi_I \wedge \phi_{th}$$

   We check the satisfiability of ϕ_o using a conflict clause generator $CCG()$ for QFP (described later in this section). If $CCG(\phi_o)$ returns UNSATISFIABLE, the algorithm UNSATISFIABLE. Otherwise, $CCG(\phi_o)$ returns a set of (conflict) clauses ϕ_c, representing tautologies in the theory of QFP.

We augment the conflict clauses ϕ_c to ϕ_b to create a more constrained Boolean skeleton of ϕ_{bt}:

$$\phi_b \leftarrow \phi_b \wedge \phi_c$$

7. [**Repeat**]. Increase the bound D for each variable by some predetermined amount $\delta > 0$, i.e. $D = D + \delta$. Goto step 4.

Let us first observe a couple of points about this algorithm.

- When ϕ_u is unsatisfiable in step 5, we know that the two components of ϕ_u, namely ϕ_b and $BE(\phi_{th}, D)$ are each satisfiable in isolation. This is because step 4 ensures that ϕ_b is satisfiable and $BE(\phi_{th}, D)$ is always satisfiable for any domain.
- The common variables in ϕ_b and $BE(\phi_{th}, D)$ are only the variables from B. This allows us to construct an interpolant in terms of the B variables, independent of the integer variables or the variables introduced during translating an integer variable to a symbolic bit vector.

3.1 Conflict Clause Generator (*CCG*)

The conflict clause generator algorithm takes as input the QFP formula $\phi_o \doteq \phi_I \wedge \phi_{th}$, where ϕ_I is a Boolean formula over B variables and checks for the unsatisfiability of the formula:

- If it returns UNSATISFIABLE, then the formula ϕ_o is unsatisfiable.
- Otherwise, it returns a conjunction of clauses $\phi_c \doteq \bigwedge_i c_i$, such that $\phi_o \implies \phi_c$ and each c_i is a disjunction of literals over B.

The conflict clauses generator can be implemented as a simple variation of a lazy SAT-based theorem proving framework as follows: Initially, ϕ_c is assigned `true`. The SAT solver enumerates assignments to B variables that satisfy ϕ_I and checks if the assignment satisfies ϕ_{th} using a (possibly incomplete) decision procedure for a conjunction of linear arithmetic constraints.

If the assignment is found unsatisfiable by the linear arithmetic decision procedure, the decision procedure returns a "proof core", a subset of constraints that are inconsistent. The negation of the proof core is a clause c_i that is added to ϕ_c (after mapping an atomic expression e to the corresponding b_e variable). The clause c_i is also added to the ϕ_I to prevent it from generating the same assignment. However, if the linear arithmetic decision procedure returns satisfiable, we simply add the negation of the assignment to ϕ_I and repeat the loop. The process is continued until the number of satisfying assignments (that are also satisfiable in the linear arithmetic theory) does not exceed some threshold. We currently limit this threshold to be 5, i.e. after obtaining more than 5 satisfying solutions, we return ϕ_c.

The usefulness of the conflict clause generator comes from the fact that it adds some structure back to the Boolean formula ϕ_u. The conflict clauses added aid the SAT-solver to avoid some case-splits when it is checking the satisfiability of the formula ϕ_u in the subsequent iterations with larger D values.

3.2 Correctness

It is not hard to show that the algorithm is sound and complete. The algorithm returns with SATISFIABLE or UNSATISFIABLE in steps 4, 5, and 6. We use the following invariant on the algorithm:

Lemma 1. *At any point in the algorithm, ϕ_{bt} is equisatisfiable with ϕ. In step 5, $\phi_u \implies \phi_{bt}$, and in step 6, $\phi_{bt} \implies \phi_o$.*

Theorem 1. *The algorithm described above is sound and complete for QFP.*

Proofs of the lemma and the theorem can be found in the Appendix.

3.3 Example

In this section, we illustrate the working of the algorithm on a simple example. Consider the following formula ϕ:

$$\begin{aligned}\phi \doteq (x < y) \wedge (y < z) \wedge (z < w \vee z < x) \wedge \\ (\neg (z < w) \vee x < 2.w + 1) \wedge (\neg (z < w) \vee \neg (x < 2.w + 1))\end{aligned} \tag{2}$$

Let us introduce Boolean variables $\{b_{x<y},\ b_{y<z},\ b_{z<w},\ \ldots\}$ for the atomic constraints in ϕ. Let ϕ_{th} denote the constraint $[\bigwedge_e b_e \Leftrightarrow e]$ as before and ϕ_b be the Boolean skeleton of ϕ.

Let us start with a domain D where each variable takes values in $\{0, 1\}$. The underapproximation of ϕ created by restricting each variable to D is unsatisfiable. Let

$$\phi_I = (b_{x<y} \wedge b_{y<z} \wedge (b_{z<w} \vee b_{z<x}))$$

be the interpolant generated in step 6 of the algorithm. In this domain, the formula $(x < y \wedge y < z \wedge (z < w \vee z < x))$ is unsatisfiable. The monome $x < y \wedge y < z \wedge z < w$ can't be satisfied because each variable can only take two values, and the monome $x < y \wedge y < z \wedge z < x$ is always unsatisfiable.

Now $\phi_o \doteq \phi_I \wedge \phi_{th}$ is fed to the conflict clause generator. Even though ϕ_o is satisfiable, it generates a conflict clause

$$\phi_c \doteq (\neg b_{x<y} \vee \neg b_{y<z} \vee \neg b_{z<x})\,.$$

Once we add the conflict clause ϕ_c to ϕ_b, we obtain unsatisfiable in the Boolean skeleton. This example illustrates the interesting case where the unsatisfiability of ϕ is detected in step 4 of the algorithm.

3.4 Discussion

In this section, we illustrated a decision procedure for QFP (or in general for a theory T) that uses a sound procedure for deciding a formula in QFP (or in theory T) as part of the conflict clause generator. The $CCG()$ decision procedure need not be complete for the theory. One can use a fast decision procedure for a subset of integer linear arithmetic (e.g. based on negative cycle detection algorithms), coupled with simple rules for more general arithmetic. This enables us to plug in any fast decision procedure for a subset of QFP (e.g. difference logic) as the $CCG()$. To ensure the completeness of the procedure, we need to

compute the small-model size D_{max} for a formula in the theory and search this domain in the worst case.

However, we can avoid the need to compute the small-model size D_{max}, if we have a sound and complete decision procedure for QFP as the conflict clause generator. In this case, one can ensure that the sequence of overapproximations $\phi_o^1, \phi_o^2, \ldots$, (where ϕ_o^i denotes the overapproximation generated during the iteration i of the above algorithm) for the input formula ϕ, will eventually converge to ϕ. Details of such a decision procedure is described in Appendix B.

4 Experiments

We have implemented a prototype of the technique in the theorem prover Zap [33]. We compare the following three variants of eager and lazy decision procedure implemented in Zap. Since the C# implementation of Zap and SharpSAT introduces some slowdown over other tools (e.g. Mathsat [4] or Barcelogic tools [23]), it is difficult to draw many conclusions about the techniques by comparing the results of these tools. Hence, we evaluate the following three variants implemented in Zap:

Verifun: This is the implementation of the lazy proof-explicating theorem prover described in Section 2.2 based on the Verifun architecture [11]. The linear arithmetic is restricted to Unit Two Variable Per Inequality (UTVPI) constraints [16], for efficiency. We do not consider examples where Verifun returns satisfiable because of the incompleteness of the procedure. The UTVPI decision procedure suffices to prove a lot of examples of our example set even in the presence of more linear arithmetic.

Eager. This implements a small-domain encoding of a quantifier-free formula to a Boolean formula [5] using the eager encoding technique mentioned in Section 2.2. For encoding QFP, it uses the small model size computed by Seshia et al. [28] reported in Equation 1.

EaZI. This is the implementation of the algorithm *Eager Zap with Interpolants* (EaZI) described in Section 3. The initial size of the domain is restricted to assign 3 bits for each integer variable. Each subsequent iteration increases the number of bits per variable by 2. For the $CCG()$ generation, it uses the Verifun procedure described above. Observe that since the Verifun procedure is restricted to UTVPI, $CCG()$ is not complete for general linear arithmetic.

In all the variants, we used SharpSAT as the Boolean SAT solver. SharpSAT is a variant of ZChaff [22] developed by Lintao Zhang at Microsoft. We ran the experiments on a 3GHz machine running Windows with 1 GB of memory. A timeout of 300 seconds was set for each benchmark.

We have performed preliminary experiments on two sets of verification benchmarks[4]: (1) *Mathsat*: This a set of QFP benchmarks obtained from the timed automata verification problems [1]. An analysis of the formulas in previous study [10] suggests that the coefficients of the QFP formulas are restricted to $\{-1, 0, 1\}$ (with more than two variables though), with the number of integer variables ranging from around 10 to around 150. Most of the variables in these

[4] We are experimenting with more arithmetic benchmarks from the SMT-LIB suite [31], and may be able to report them for the final paper

benchmarks are Boolean variables. (2) *SAL*: This set of benchmarks [10] consists of formulas derived from bounded model checking of linear and hybrid systems and from test-case generation for embedded systems. Most of the benchmarks have significant linear arithmetic constraints and the number of integer variables range from tens to hundreds of variables.

4.1 Results

In this section, we present the results of running the three approaches on the Mathsat and SAL benchmarks. We analyze the results for different options:

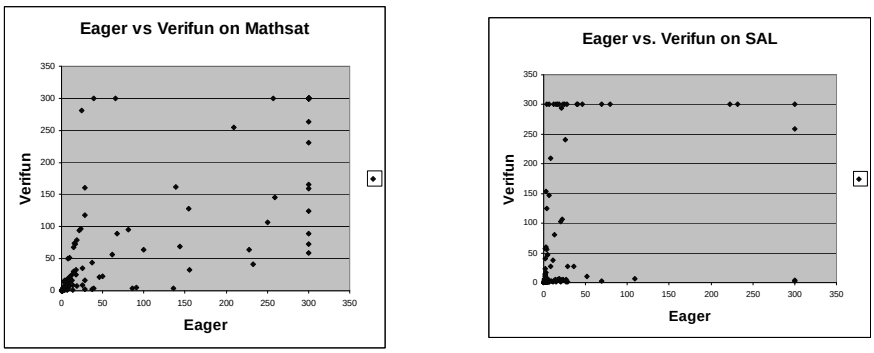


Fig. 1. Eager vs. Verifun on Mathsat and Sal benchmarks.

Verifun vs. Eager Figure 1 compares the Verifun and the Eager approach on the benchmarks. Verifun scales better than Eager on the Mathsat benchmarks whereas Eager outperforms Verifun on SAL benchmarks. We believe the difference in performance of Verifun on the two sets of benchmarks can be explained by the number of times it invokes a theory decision procedure. Column 3 of Figure 4.1 (marked "# Verifun Loops" for "Verifun") shows that the number of calls to the theory decision procedure for SAL benchmarks is at least an order greater than in the case for Mathsat benchmarks. The results suggest that neither Eager nor Verifun is robust enough for a large set of benchmarks.

Verifun vs. EaZI Figure 2 and Figure 3 compares EaZI with Eager and Verifun on Mathsat and SAL benchmarks respectively.

EaZI outperforms Verifun consistently on both the set of benchmarks. To understand the improvement, we extracted some information for a subset of benchmarks. Figure 4.1 compares the two approaches in terms of three metric (a) number of times a theory decision procedure was invoked, (b) the average size of the number of constraints involved in the monome passed to the decision procedure and (c) total time spent in the theory decision procedure.

We observed the following characteristics across a wide set of benchmarks:

– The number of times a theory decision procedure is invoked is usually much smaller in the case of EaZI. This can be explained because the abstraction of

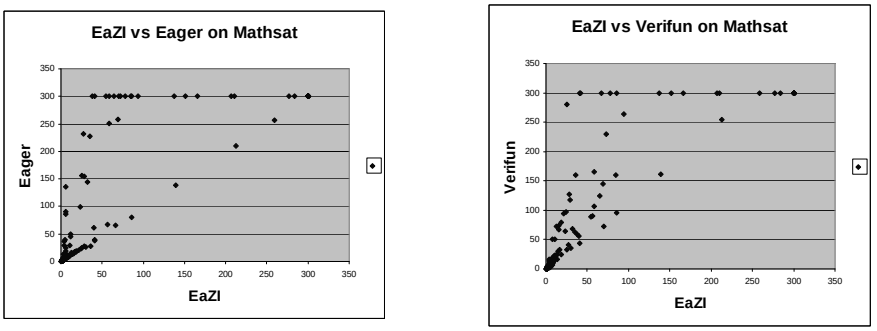


Fig. 2. Eazi vs. Eager and Verifun on Mathsat Benchmarks.

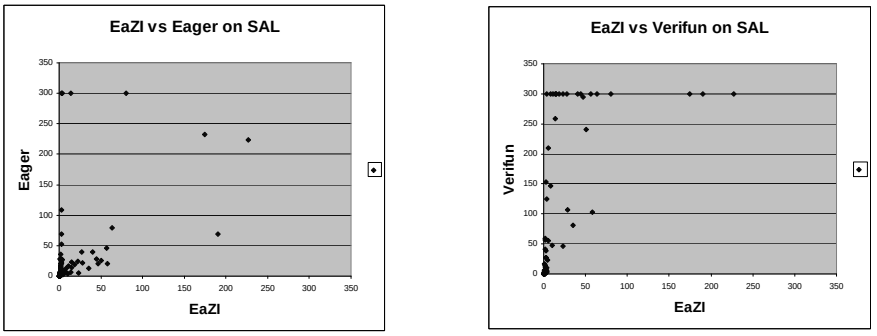


Fig. 3. Eazi vs. Eager and Verifun on SAL Benchmarks.

the original formula ϕ_o has lot more conflicts in the Boolean structure and thus comes to theory less often.

- The size of monome and the total time spent in the theory is reduced at least by an order of magnitude in most cases. This is because the abstraction ϕ_o contains a lot fewer constraints than ϕ.
- The main bottleneck of the EaZI method shifts from the theory decision procedure to SAT. In most cases, the interpolant generation consumes more than 80% of the total time. We believe the performance of our algorithm will further improve with improvement in the the interpolant generation implementation in SharpSAT.

Eager vs. EaZI Figure 2 and Figure 3 indicate that EaZI also outperforms the Eager method in most of the examples in this set.

The small model size for the most examples calculated by Equation 1 requires the Eager method to encode each variable with more than 15 bits in most case. In some cases with large number of non-difference constraints, the number of bits to encode each variable exceeds 100 bits. Although the EaZI method relies on the small-model size for completeness, it never reaches this maximum domain size for either proving unsatisfiability or satisfiability. For satisfiable instances, it finds a satisfying solution using a small number of bits for most examples. For unsatisfiable cases, the procedure exits from the $CCG()$ procedure with UNSATISFIABLE, or the conflict clauses generated from $CCG()$ makes the Boolean part

Example		# Verifun Loops		Avg. Monome size		Total theory time(secs)	
		Verifun	EaZI	Verifun	EaZI	Verifun	EaZI
sal	Carpark2-t1-4	200	2	755	1	222.27	0.06
	fischer3-mutex-5	372	72	399	66	116.84	2.45
	fischer6-mutex-3	143	18	426	19	39.28	0.17
	fischer9-mutex-3	221	40	603	15	107.14	0.25
	inf-bakery-invalid-10	180	0	211	-	47.90	0
	inf-bakery-mutex-9	426	66	200	38	149.77	1.11
	lpsat-goal-12	233	45	2019	19	87.75	0.34
	windowreal-safe-4	185	14	328	22	13.14	0.17
	windowreal-safe2-4	254	8	328	23	19.14	0.17
mathsat	FISCHER10-5-fair	19	3	969	49	30.46	0.34
	FISCHER11-5-fair	21	25	1072	113	42.54	11.53
	FISCHER3-8-fair	93	0	478	-	66.89	0
	FISCHER5-7-fair	81	52	682	93	108.69	8.85
	FISCHER8-6-fair	36	40	935	108	94.72	14.29
	PO3-10-PO3	18	0	686	-	18.89	0
	PO3-9-PO3	38	3	618	20	25.53	0.07

Fig. 4. Analysis of EaZI vs. Verifun on Mathsat and SAL.

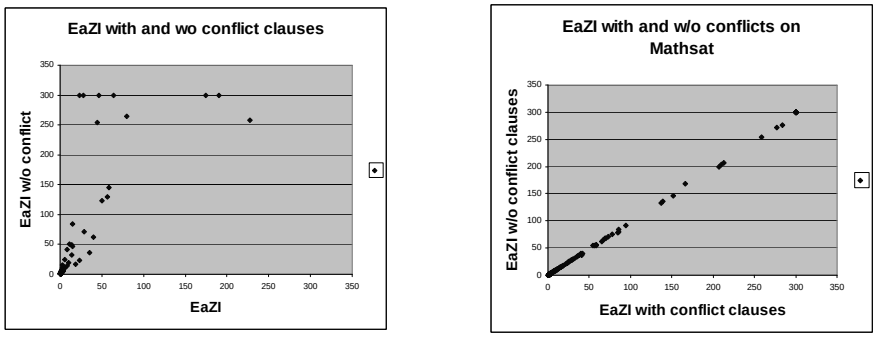


Fig. 5. Effect on conflict clauses in Eazi for SAL and Mathsat Benchmarks.

of the formula ϕ_{bt} unsatisfiable. Even for the unsatisfiable instances, we need to increase the number of bits to at most 8 in most cases.

The Eager approach outperforms the EaZI approach on some of the SAL benchmarks. This is primarily because the number of bits to encode each variable is less than 20 and the overhead of computing interpolants in EaZI offsets the advantage of learning from smaller domain size.

Adding Conflict Clauses We conjectured that the advantage of EaZI over Eager comes from the fact that we add more structure (the conflict clauses from $CCG()$) to ϕ_b and thereby aids the SAT solver to prune away the search space efficiently. Figure 5 compares the EaZI approach with and without the conflict clauses from $CCG()$. The conflict clauses make marked difference in the results for the SAL benchmarks. Since the dominant time in the SAL experiments is in solving the Boolean formula ϕ_u, addition of the conflict clauses help the SAT solver. However, adding the conflict clauses made almost no difference for the Mathsat examples. For most of the Mathsat examples, we found that the abstraction ϕ_o was proved unsatisfiable by the conflict clause generator after the first iteration. Hence the conflict clauses do not play a part in most of these examples.

5 Related Work

In this section, we compare the interpolant based decision procedure for QFP for other decision procedures for QFP or its restricted fragments. The use of interpolants has been recently explored for finite-state model checking [21] and in refining the abstractions for software verification [14, 17].

Eager approaches for translating a QFP formula to an equisatisfiable Boolean formula employ either the small-model encoding discusses in Section 2.2 or add all the theory constraints to the formula [32]. The latter approach can result in an exponential number of constraints being added to the original formula. This translation is often the bottleneck in the method. In [29], the authors encode disjoint set of constraints in a formula using either the small-model encoding or by adding all the theory constraints. The approach was restricted to the difference logic fragment of QFP, and use the number of constraints in the formula to determine the encoding. In contrast, our approach uses small-encoding but increases the size of encoding lazily starting from a small size. Beside, we only add a very small set of theory constraints as conflict clauses in a more lazy manner.

Lazy approaches [3, 11] use the lazy proof-exlicating framework described in Section 2.2. The main bottleneck of these approaches appear to be the large number of invocations of the theory decision procedure. The monomes passed to the decision procedure are often large, even though the reason for unsatisfiability of the monome is often simple and small. Our approach addresses this problem by creating an abstraction of the original formula by using the interpolants and using the lazy approach as a mean to generate conflict clauses. Our experiments indicate that the size of the monome passed to the decision procedure is reduced by more than an order of magnitude and the time spent in the theory decision procedure is reduced considerably. Mathsat [1] use a *layered* approach, where a sequence of increasingly complete (and therefore more complex) decision procedures to decide a monome. However, to our knowledge, the approach still suffers from passing large monomes to the linear arithmetic decision procedures.

DPLL(T) based approaches [13, 23, 4] use a closer integration of the theory decision procedure with the SAT solver. In addition to the Boolean constraint propagation in the SAT solver, the theory participates in the constraint propagation by adding all the theory facts implied by the theory in the current context. This enables detection of early unsatisfiability than the lazy approaches. However, the decision procedures for the theories become more complex as they need to support the generation of all the facts that are implied by a set of constraints. This may increase the complexity of the ILP decision procedures that already have an exponential worst-case complexity. A promising approach has been suggested by Sheini and Sakallah [30], where they integrate an UTVPI decision procedure in DPLL(T) framework and use the general ILP solver in a lazy framework. Currently, implementations based on DPLL(T) framework are most competitive on the SMT-LIB QFP benchmarks.

The approach closest to our work is Kroening et al.'s [18] work on deciding QFP formulas using Boolean proof of unsatisfiability. Kroening et al. construct a equisatisfiable clausal representation of the original QFP formula by introducing additional variables. The clausal form is encoded to a Boolean formula by performing small-model encoding starting with a small size. An abstraction of the original clausal formula is constructed by choosing a subset of clauses that appear in the proof of unsatisfiability of the Boolean formula. This abstrac-

tion is checked using a sound and complete decision procedure for QFP. The sequence is repeated with increasing small-model encoding size. Although similar in many aspects, our approach differ from this work in several ways. First, this method appears to require a clausal representation of the original formula (that can introduce auxiliary variables and destroy some of the structure of the original formula), and the abstraction is limited to be a subset of the clauses in this representation. The abstraction is obtained in a fairly syntactic fashion, by considering the subset of clauses for which there is a corresponding clause in the proof of unsatisfiability for the Boolean formula. We believe that the use of interpolants results in a more semantic method to construct the abstraction. This might often result in more concise abstractions being generated. Secondly, unlike their approach we do not require a complete decision procedure for QFP (for checking the abstraction) to ensure the completeness of the procedure. Their experiments (although on a different set of benchmarks) indicate that the lazy decision procedure for ILP dominates the total time of the procedure. Finally, we add conflict clauses from the abstractions generated from smaller domain sizes; this allows us to prune the search space when searching in a larger domain.

6 Conclusion

In this paper, we present a framework for using simultaneous under and over approximations to decide a quantifier-free first-order formula. The small-model encoding is used to create the underapproximation of the formula and the interpolant generation from the proofs of unsatisfiability gives an overapproximation of the formula. The method also demonstrates a mechanism to leverage some conflict clauses learned during searching in a smaller domain size, to prune the search space during the later search.

One of the directions of future work is to implement Kroening et al.'s scheme and empirically compare the two approaches on the set of benchmarks. We are also working on getting a more extensive evaluation on the SMT-LIB benchmarks.

References

1. G. Audemard, P. Bertoli, A. Cimatti, A. Korniłowicz, and R. Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE-18)*, volume 2392 of *LNCS*, pages 195–210. Springer Verlag, July 2002.
2. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI '01)*, Snowbird, Utah, June, 2001. *SIGPLAN Notices,* 36(5), May 2001.
3. C. W. Barrett, D. L. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 236–249, July 2002.
4. M. Bozzano, R. Bruttomesso, A. Cimatti, T. A. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In *CAV*, volume 3576 of *LNCS 3576*, pages 335–349, 2005.
5. R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted

Functions. In *Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 78–92, July 2002.
6. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
7. W. Craig. Linear reasoning. a new form of the herbrand-gentzen theorem. *J. Symbolic Logic*, 22:250–268, 1957.
8. M. Davis, G.eorge Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.
9. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.
10. Leonardo M. de Moura and Harald Rueß. An experimental evaluation of ground decision procedures. In *Computer Aided Verification (CAV '04)*, LNCS 3114, pages 162–174. Springer-Verlag, 2004.
11. C. Flanagan, R. Joshi, X. Ou, and J. Saxe. Theorem Proving usign Lazy Proof Explication. In *Computer-Aided Verification (CAV 2003)*, LNCS 2725, pages 355–367. Springer-Verlag, 2003.
12. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, pages 234–245, 2002.
13. Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast Decision Procedures. In *Computer Aided Verification (CAV '04)*, LNCS 3114, pages 175–188. Springer-Verlag, 2004.
14. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Symposium on Principles of programming languages (POPL '04)*, pages 232–244. ACM Press, 2004.
15. ILOG CPLEX. Available at `http://ilog.com/products/cplex`.
16. J. Jaffar, M. J. Maher, P. J. Stuckey, and H. C. Yap. Beyond Finite Domains. In Alan Borning, editor, *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, PPCP'94*, volume 874 of *LNCS 874*, pages 86–94. Springer-Verlag, 1994.
17. Ranjit Jhala and Kenneth L. McMillan. Interpolant-based transition relation approximation. In *Computer Aided Verification, 17th International Conference, CAV 2005*, volume 3576 of *LNCS*, pages 39–51. Springer, 2005.
18. Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, and Ofer Strichman. Abstraction-based satisfiability solving of presburger arithmetic. In *Computer Aided Verification (CAV '04)*, LNCS 3114, pages 308–320. Springer-Verlag, 2004.
19. Shuvendu K. Lahiri and Sanjit A. Seshia. The uclid decision procedure. In *Computer Aided Verification (CAV '04)*, LNCS 3114, pages 475–478. Springer-Verlag, 2004.
20. LP_SOLVE. Available at `http://groups.yahoo.com/group/lp_solve/`.
21. K.L. McMillan. Interpolation and sat-based model checking. In *CAV 03: Computer-Aided Verification*, LNCS 2725, pages 1–13. Springer-Verlag, 2003.
22. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *38th Design Automation Conference (DAC '01)*, 2001.
23. R. Nieuwenhuis and A. Oliveras. DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic. In *Computer Aided Verification, 17th International Conference, CAV 2005*, LNCS, pages 321–334. Springer, 2005.
24. Christos H. Papadimitriou. On the complexity of integer programming. *J. ACM*, 28(4):765–768, 1981.
25. M. Presburger. Uber die Vollstfindigkeit eines gewissea Systems Arithmetic ganzer Zahlen in welchem die Addition als eiilzige Operation hervortritt. In *Comptes-rendus du I Congr s de Mathematiciens de Pays Slaves*, pages 395:92–101, 1930.
26. P. Pudl'ak. Lower bounds for resolution and cutting planes proofs and monotone computations. *J. of Symbolic Logic*, 62(3):981–998, 1995.

27. H. Rueß and N. Shankar. Solving linear arithmetic constraints. Technical Report CSL-SRI-04-01, SRI International, January 2004.
28. S. A. Seshia and R. E. Bryant. Deciding quantifier-free presburger formulas using parameterized solution bounds. In *19th IEEE Symposium of Logic in Computer Science(LICS '04)*. IEEE Computer Society, July 2004.
29. S. A. Seshia, S. K. Lahiri, and R. E. Bryant. A Hybrid SAT-based Decision Procedure for Separation Logic with Uninterpreted Functions. In *Proc. Design Automation Conference (DAC)*, pages 425–430, June 2003.
30. Hossein M. Sheini and Karem A. Sakallah. A scalable method for solving satisfiability of integer linear arithmetic logic. In *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005*, volume 3569 of *LNCS*, pages 241–256. Springer, 2005.
31. SMT-LIB: The Satisfiability Modulo Theories Library. Available at `http://combination.cs.uiowa.edu/smtlib/`.
32. Ofer Strichman. On solving presburger and linear arithmetic with sat. In *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, pages 160–170, 2002.
33. Zap Project. Available at `http://www.research.microsoft.com/tvm`.

A Proof of correctness

Proof (Lemma 1 in Section 3). The proof follows simply by induction on the number of iterations of the algorithm.

For the first iteration, ϕ_{bt} is equisatisfiable with ϕ since $\phi = \exists B : \phi_{bt}$. Since ϕ_u simply restricts the domain of each variable in ϕ_{bt}, clearly $\phi_u \implies \phi_{bt}$. In step 6, assume ϕ_u is unsatisfiable. Recall that ϕ_b is not unsatisfiable because we have ensured that in step 4. We also know that the formula $BE(\phi_{th}, D)$ is satisfiable. This is because it just says that the value of b_e is the same as the value of e. There are no constraints on any b_e variables or the atomic constraints e. This explains the reason why we have step 4 explicitly in the algorithm. Moreover, recall that ϕ_u is a Boolean formula. Therefore, we can construct a Boolean interpolant ϕ_I for the pair $(\phi_b, BE(\phi_{th}, D))$. Since $\phi_b \implies \phi_I$, $\phi_{bt} \implies \phi_o$.

Now, let us assume that the lemma holds for some iteration. We want to prove that it holds for the next iteration. Recall, that the only step in which ϕ_b changes is in step 6. We also know that $\phi_o \implies \phi_c$ (from the property of the conflict clause generator), and hence $\phi_{bt} \implies \phi_c$. This in turn implies that $\phi_{bt} \implies (\phi_{bt} \wedge \phi_c)$. Moreover, $(\phi_c \wedge \phi_{bt} \implies \phi_{bt}$. Hence $(\phi_c \wedge \phi_{bt}) \Leftrightarrow \phi_{bt}$. Therefore, the formula ϕ_{bt} is equivalent across all iterations steps. This preserves other parts of the lemma.

Proof (Theorem 1 in Section 3). It is easy to see that the algorithm terminates, since D is monotonically increased and finally crosses D_{max}. Observe that the algorithm returns SATISFIABLE, if and only when it finds ϕ_u SATISFIABLE. Lemma 1 ensures soundness of the algorithm. The algorithm returns UNSATISFIABLE if and only if a formula weaker than ϕ_{bt} is unsatisfiable. The weaker formulas in step 4, step 5 and step 6 are ϕ_b, ϕ_u (only when $D \geq D_{max}$) and ϕ_o respectively.

B Termination in the absence of D_{max}

In the previous section, the termination argument required computing a maximum domain size D_{max}. When $D \geq D_{max}$, then the formula ϕ_u is equisatisfiable

with ϕ_{bt}, and therefore also with ϕ. In this section, we show that we can modify the algorithm slightly to ensure that the computation terminates even in the absence of an upper bound.

The main change required from the previous algorithm is to replace the conflict clause generator (CCG) with a sound and complete lazy SAT-based decision procedure for QFP. Since the conflict clauses generated from the CCG are simply adding more structure to ϕ_b and not required for completeness, we only required a sound decision procedure QFP for the previous algorithm.

The new algorithm is identical to the previous algorithm except for step 3 (where we do not compute D_{max}), step 6 and step 7. The new step 6 and step 7 becomes (for simplicity, we ignore the conflict clauses ϕ_c):

- **[Overapproximation]**. If ϕ_u is unsatisfiable, compute the Boolean interpolant ϕ_I of ϕ_b and $BE(\phi_{th}, D)$. We construct the formula ϕ_o which is an overapproximation of ϕ_{bt} as follows:

  $$\phi_o \doteq \phi_I \wedge \phi_{th}$$

  We check the satisfiability of ϕ_o using a lazy SAT-based decision procedure for QFP. If the decision procedure returns UNSATISFIABLE, the algorithm returns UNSATISFIABLE. If the decision procedure returns SATISFIABLE, it also returns D', the maximum value of any variable in the satisfying assignment. If $\phi_I \Leftrightarrow \phi_b$, return SATISFIABLE.
- **[Repeat]**. Make $D = D'$ and go to step 4.

Lemma 2. *For any two distinct iterations i and j of the above algorithm, let ϕ_I^i and ϕ_I^j be the interpolants computed in step 6 of the algorithm in iterations i and j respectively. Then $\phi_I^i \not\Leftrightarrow \phi_I^j$.*

Proof. Let us assume $\phi_I^i \Leftrightarrow \phi_I^j$. Let us assume w.l.o.g. that $i < j$. Let D_k and D'_k be the value of D at the start of the iterations k (for $k \in \{i, j\}$). Let us also assume that both the iterations reach step 6. This means that $\phi_I^k \wedge BE(\phi_{th}, D_k)$ (for $k \in \{i, j\}$) is unsatisfiable. Moreover $\phi_I^i \wedge BE(\phi_{th}, D'_i)$ is satisfiable. Since $j > i$, $D_j \geq D'_i$. Therefore $\phi_I^j \wedge BE(\phi_{th}, D_j)$ is clearly satisfiable, and therefore ϕ_I^j can't the interpolant during the iteration j. Hence we reach a contradiction.

Theorem 2. *The algorithm described in this section that does not precompute D_{max} is sound and complete for QFP.*

Proof. Since there can only be a finite number of distinct Boolean functions over $|B|$ variables and $\phi_b \implies \phi_I$ for any iteration, Lemma 2 ensures that the algorithm above terminates. Therefore, by Theorem 1, the modified algorithm is sound and complete.

In the above algorithm, we have assumed that the decision procedure for QFP used in the overapproximation step (step 6) can construct a satisfying assignment (and therefore provide D'). We can relax this restriction and only increase D by a fixed amount δ as before and still obtain termination. This is because, using the same idea as Lemma 2, we can ensure that if ϕ_I^i is the interpolant at step i, and $j > i + (D'_i - D_i)/\delta$, then ϕ_I^j is distinct from ϕ_I^i. Although this does not allow us to bound the number of iterations (without refering D_{max}), we can ensure that in the limit some interpolant ϕ_I is going to be identical to ϕ_b for some finite iteration.

[blank page]

Using SAT Encodings to Derive CSP Value Ordering Heuristics

Christophe Lecoutre, Lakhdar Sais, and Julien Vion

CRIL-CNRS FRE 2499,
Université d'Artois
Lens, France
$\{lecoutre, sais, vion\}$@cril.univ-artois.fr

Abstract. In this paper, we address the issue of value ordering heuristics in the context of a backtracking search algorithm that exploits binary branching and the adaptive variable ordering heuristic $dom/wdeg$. Our initial experimentation on random instances shows that (in this context), contrary to general belief, following the $fail\text{-}first$ policy instead of the $promise$ policy is not really penalising. Furthermore, using SAT encodings of CSP instances, a new value ordering heuristic related to the $fail\text{-}first$ policy can be naturally derived from the well-known *Jeroslow-Wang* heuristic. This heuristic, called $min\text{-}inverse$, exploits the bi-directionality of constraint supports to give a more comprehensive picture in terms of domain reduction when a given value is assigned to (resp. removed from) a given variable. An extensive experimentation on a wide range of CSP instances shows that $min\text{-}inverse$ can outperform the other known value ordering heuristics.

1 Introduction

For solving instances of the Constraint Satisfaction Problem (CSP), backtracking search algorithms are commonly used. To limit their combinatorial explosion, various improvements have been proposed (e.g. ordering heuristics, filtering techniques and conflict analysis). It is well known that the ordering used to perform search decisions has a great impact on the size of the search tree. At each stage, one needs to decide the *value* to assign to a *variable*. So far, such decisions have been performed by choosing the variable in a first step (vertical selection) and the value to assign in a second step (horizontal selection).

Many works have been devoted to the first selection step. Variable ordering heuristics that have been proposed can be conveniently classified as static (e.g. deg), dynamic (e.g. dom [15], bz [6], $dom/ddeg$ [4]) and adaptive (e.g. $dom/wdeg$ [5]). The heuristic $dom/wdeg$ has been shown superior to the other ones [5, 21, 16, 28]. However, value ordering (the second step of the decision) has clearly been considered for a long time as potentially of marginal effect to search improvements. The arguments behind this can be related to the fact that selecting a given value is computationally more difficult than selecting a given variable, particularly when one considers dynamic selection. The second reason for considering value ordering as useless is that, when facing unsatisfiable instances or when searching all solutions, one needs to consider all values for each

variable. As clearly shown by Smith and Sturdy [26], these arguments hold when search is based on d-way branching but not on 2-way branching. d-way branching means that, at each node of the search tree, a variable x is selected and d branches are considered where d is the current size of the domain of x: the ith branch corresponds to $x = a_i$ where a_i denotes the ith value of the domain of x. On the other hand, with binary (or 2-way) branching, at each node of the search tree, a pair (x,a) is selected where x is an unassigned variable and a a value in the domain of x, and two branches are considered: the first one corresponds to the assignment $x = a$ and the second one to the refutation $x \neq a$. These two schemes are not equivalent as it has been shown that binary branching is more powerful than non-binary branching [17].

Traditionally, two principles are considered during search: at each step, select the variable which is the most constrained and select then the least constrained value (e.g. $min\text{-}conflicts$ [12]). These principles respectively correspond to two policies called $fail\text{-}first$ and $promise$, and one interesting issue is the adherence assessment of heuristics to both policies [2, 29]. In this paper, we focus on value ordering heuristics, and more precisely, we try to determine if value ordering heuristics should adhere in priority to the $promise$ policy. Of course, one can be surprised that we address this issue as it is commonly admitted that it should be the case. In particular, a lot of works support the idea that a value must be chosen by estimating the number of solutions or conflicts. One has then to prefer the value that maximizes the estimated number of solutions in the remaining network [8, 13, 24, 20] or minimizes the number of conflicts with variables in the neighbourhood [12, 23].

However, we noticed that most of the experimental results are given when d-way branching and/or non adaptive variable heuristics (such as dom, bz, $dom/ddeg$) are used. This is the reason why we decided to solve a wide range of random CSP instances using the MAC algorithm, i.e. the algorithm that maintains arc consistency during search [25]. We tested both branching schemes and both dynamic and adaptive variable ordering heuristics on 7 classes of binary instances situated at the phase transition of search. For each class $\langle n,d,e,t\rangle$, defined as usually, 50 instances have been generated. More precisely, the number of variables n has been set to 40, the domain size d between 8 and 180, the number of constraints e between 753 and 84 (and, so the density between 0.96 and 0.1) and the tightness t (which denotes here the probability that a pair of values is allowed by a relation) between 0.1 and 0.9. The first class $\langle 40,8,753,0.1\rangle$ corresponds to dense instances involving constraints of low tightness whereas the seventh one $\langle 40,180,84,0.9\rangle$ corresponds to sparse instances involving constraints of high tightness. What is interesting here is that a significant sampling of domain sizes, densities and tightnesses is considered.

In Tables 1 and 2, we can observe the results that we have obtained with the classical value ordering heuristic $min\text{-}conflicts$ and the "anti" heuristic $max\text{-}conflicts$[1] identified as $confts$. Here $min\text{-}conflicts$ corresponds to the heuristic called mc in [12] and involves selecting the value with the lowest number of conflicts with values in adjacent domains. Performances are given (on average) in terms of the cpu time (in seconds), the

[1] The value ordering heuristics considered here are static [23]. It means that the order of values is computed in a preprocessing step. In any case, we observed similar behaviours with dynamic versions.

		$dom/ddeg$			$dom/wdeg$		
$Instances$		$min\text{-}conflts$	$max\text{-}confts$	$ratio$	$min\text{-}confts$	$max\text{-}confts$	$ratio$
⟨40-8-753-0.1⟩	cpu	42.0	51.4	1.22	34.5	41.6	1.20
	$ccks$	$22M$	$27M$	1.22	$20M$	$24M$	1.20
	$nodes$	43,269	55,558	1.28	38,104	48,158	1.59
⟨40-11-414-0.2⟩	cpu	30.9	35.0	1.13	29.4	32.7	1.11
	$ccks$	$26M$	$29M$	1.11	$26M$	$29M$	1.11
	$nodes$	58,955	70,007	1.18	58,055	67,905	1.17
⟨40-16-250-0.35⟩	cpu	22.1	28.9	1.30	21.0	26.6	1.26
	$ccks$	$30M$	$40M$	1.33	$30M$	$37M$	1.23
	$nodes$	59,669	83,445	1.39	56,036	75,025	1.33
⟨40-25-180-0.5⟩	cpu	33.1	37.1	1.12	28.6	30.0	1.04
	$ccks$	$62M$	$67M$	1.08	$55M$	$57M$	1.03
	$nodes$	85,122	98,519	1.15	69,805	78,005	1.11
⟨40-40-135-0.65⟩	cpu	25.9	34.6	1.33	20.0	25.1	1.25
	$ccks$	$68M$	$89M$	1.30	$53M$	$66M$	1.24
	$nodes$	52,622	74,592	1.41	36,571	49,211	1.34
⟨40-80-103-0.8⟩	cpu	25.8	52.8	2.04	15.3	36.3	2.37
	$ccks$	$98M$	$193M$	1.96	$59M$	$133M$	2.25
	$nodes$	29,989	72,841	2.42	16,163	45,177	2.79
⟨40-180-84-0.9⟩	cpu	113.1	121.3	1.07	40.6	44.6	1.09
	$ccks$	$554M$	$587M$	1.05	$217M$	$231M$	1.06
	$nodes$	76,788	85,482	1.11	20,077	22,557	1.12

Table 1. MAC with d-way branching, $dom/ddeg$ and $dom/wdeg$

		$dom/ddeg$			$dom/wdeg$		
$Instances$		$min\text{-}confts$	$max\text{-}confts$	$ratio$	$minconfts$	$max\text{-}confts$	$ratio$
⟨40-8-753-0.1⟩	cpu	29.3	35.8	1.22	28.9	28.4	0.98
	$ccks$	$22M$	$27M$	1.22	$24M$	$23M$	0.95
	$nodes$	43,268	55,557	1.28	45,650	46,645	1.02
⟨40-11-414-0.2⟩	cpu	23.0	25.9	1.12	26.1	27.3	1.04
	$ccks$	$26M$	$29M$	1.11	$32M$	$33M$	1.03
	$nodes$	59,002	70,026	1.18	69,111	76,941	1.11
⟨40-16-250-0.35⟩	cpu	18.5	24.5	1.32	23.0	24.4	1.06
	$ccks$	$30M$	$40M$	1.33	$39M$	$41M$	1.05
	$nodes$	59,773	83,531	1.18	72,555	82,459	1.13
⟨40-25-180-0.5⟩	cpu	28.8	31.9	1.33	28.5	30.7	1.07
	$ccks$	$62M$	$67M$	1.08	$65M$	$68M$	1.04
	$nodes$	85,187	98,548	1.15	80,017	91,464	1.14
⟨40-40-135-0.65⟩	cpu	21.4	28.6	1.33	19.8	19.6	0.98
	$ccks$	$68M$	$89M$	1.30	$65M$	$64M$	0.98
	$nodes$	52,569	74,544	1.41	44,120	46,573	1.05
⟨40-80-103-0.8⟩	cpu	20.4	42.3	2.07	12.6	18.6	1.47
	$ccks$	$98M$	$193M$	1.96	$64M$	$89M$	1.39
	$nodes$	29,931	72,747	1.41	16,168	28,087	1.73
⟨40-180-84-0.9⟩	cpu	85.0	92.0	1.08	26.4	27.1	1.02
	$ccks$	$553M$	$587M$	1.06	$192M$	$193M$	1.00
	$nodes$	76,489	85,255	1.11	15,835	16,566	1.04

Table 2. MAC with 2-way branching, $dom/ddeg$ and $dom/wdeg$

number of constraint checks (ccks) and the number of nodes of the explored search tree. What is interesting to note is that while the performance ratio between $max\text{-}conflicts$ and $min\text{-}conflicts$ usually lies between 1.1 and 1.3 when one uses d-way branching or a classical heuristic (here, $dom/ddeg$), it falls around 1 when one uses binary branching and $dom/wdeg$. A noticeable exception is for the class $\langle 40{,}80{,}103{,}0.8\rangle$. One can also remark that the proportion of constraint checks per visited node is weaker when $max\text{-}conflicts$ is used. This is natural since by selecting in priority conflicting values, the size of the search space is reduced faster. Finally, our observation suggests that an analysis as the one performed in [16] deserves to be considered for 2-way branching.

Considering the results of our experience about random instances and the fact that, for some types of constraints, good value ordering can significantly reduce the search effort [26], we decided to further investigate value ordering heuristics (assuming, of course, an underlying 2-way branching scheme). In particular, our attention was attracted by the fact that 2-way branching is the basic scheme in SAT solvers. We thought that this might be very helpful to map SAT heuristics to CSP ones. Indeed, considering any SAT encoding of a CSP instance, selecting a pair composed of a variable and a value corresponds to the selection of a literal in SAT.

In this paper, we propose a new value ordering heuristic that is derived from the well known *Jeroslow-Wang (JW)* heuristic [18]. The obtained heuristic, called $min\text{-}inverse$, exploits the bi-directionality of constraints to give a more comprehensive picture in terms of domain reduction when a given value is assigned to a given variable and *also* when a given value is removed from the domain of a given variable. Let us illustrate this with the following example.

Example 1. Let C be the binary constraint depicted by Figure 1. Note that any value in the domain of x_1 occurs in two allowed tuples and is in conflict with two values in the domain of x_2. Consequently, applying a classical value ordering heuristics such as $min\text{-}conflicts$ (or $max\text{-}conflicts$) do not discriminate between the different values of x_1 since all the values of x_1 have the same number of conflicts in x_2.

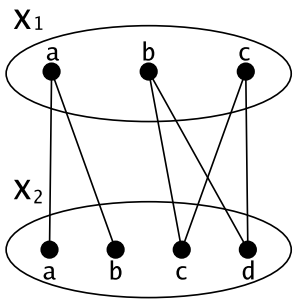


Fig. 1. A constraint between x_1 and x_2. Edges correspond to allowed tuples.

This example shows that it is not always sufficient to only consider the number of conflicts in order to choose the most (or least) promising value. However, considering a

binary branching scheme, when the value a is assigned to x_1 (decision corresponding to the first branch), two values are removed from dom(x_2), and when a is removed from dom(x_1) (decision corresponding to the second branch) two values are also removed from dom(x_2). On the other hand, when the value b or c is assigned to x_1, two values are removed from dom(x_2), and when b or c is removed from dom(x_1) no value is removed from dom(x_2). So, the value a is more constrained than b or c. This illustration shows that it can be important to consider the impact on both branches when evaluating values to be selected by an heuristic.

Our heuristic is then related to $max\text{-}conflicts$, but this last one only gives the estimation of the number of removed values when assigning a value to a given variable (the assignment labelling the first branch of a binary search). In fact, the estimation of the number of removed values when eliminating a given value from the domain of a given variable (the refutation labelling the second branch of a binary search) has not been (to our knowledge) considered so far when devising value ordering heuristics. Interestingly enough, our approach can be used to derive in more general way a suitable value ordering with respect to any type of constraint. We also show a direct correspondence between $min\text{-}conflicts$ (resp. $max\text{-}conflicts$) and the maximum number of literal occurrences in the SAT formula obtained using support (resp. direct) encoding of CSP instances.

The rest of the paper is organized as follows. After some technical background about CSP and SAT, SAT encodings of CSP instances are recalled. Our approach is then presented. Experimental results conducted on a wide range of CSP instances are described and discussed before concluding.

2 Technical Background

2.1 Constraint Satisfaction Problem

A (finite) Constraint Network (CN) P is a pair $(\mathscr{X}, \mathscr{C})$ where $\mathscr{X}$ is a finite set of variables and $\mathscr{C}$ a finite set of constraints. Each variable $x \in \mathscr{X}$ has an associated domain, denoted *dom*(x), which contains the set of values allowed for x. Each constraint $C \in \mathscr{C}$ involves a subset of variables of $\mathscr{X}$, called the scope and denoted *vars*(C), and has an associated relation, denoted *rel*(C), which contains the set of tuples allowed for the variables of its scope. From now on, to simplify and without any loss of generality, we will only consider binary networks, i.e. networks involving binary constraints.

A solution to a constraint network is an assignment of values to all the variables such that all the constraints are satisfied. A constraint network is said to be satisfiable iff it admits at least one solution. The Constraint Satisfaction Problem (CSP) is the NP-complete task of determining whether a given constraint network is satisfiable. A CSP instance is then defined by a constraint network, and solving it involves either finding one (or more) solution or determining its unsatisfiability. To solve a CSP instance, one can modify the constraint network by using inference or search methods. Usually, domains of variables are reduced by removing inconsistent values, i.e. values that can not occur in any solution. Indeed, it is possible to filter domains by considering some properties of constraint networks. Arc Consistency (AC), which remains the central one, guarantees the existence of a support for each value in each constraint.

Algorithm 1 MAC(P = $(\mathscr{X}, \mathscr{C})$: Constraint Network) : Boolean

1: **if** $\mathscr{X} = \emptyset$ **then return** true
2: $P' \leftarrow AC(P)$
3: **if** $P' = \bot$ **then return** false
4: select a pair (x,a) such that $x \in \mathscr{X}$ and $a \in dom(x)$
5: **if** MAC($P'|_{x=a} \backslash x$) **then return** true
6: **if** MAC($P'|_{x \neq a}$) **then return** true
7: **return** false

Definition 1. *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN, $C \in \mathscr{C}$ such that $vars(C) = \{x, y\}$ and $a \in dom(x)$.*

- *The set of supports of (x, a) in C, denoted $supports(C, x, a)$, corresponds to the set $\{b \in dom(y) \mid (a, b) \in rel(C)\}$.*
- *The set of conflicts of (x, a) in C, denoted $conflicts(C, x, a)$, corresponds to the the set $\{b \in dom(y) \mid (a, b) \notin rel(C)\}$.*

Definition 2. *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN. A pair (x,a), with $x \in \mathscr{X}$ and $a \in dom(x)$, is arc consistent (AC) iff $\forall C \in \mathscr{C} | x \in vars(C)$, $supports(C, x, a) \neq \emptyset$. P is AC iff $\forall x \in \mathscr{X}$, dom($x$) $\neq \emptyset$ and $\forall a \in dom(x)$, (x, a) is AC.*

Let us now, briefly describe the well known MAC algorithm [25]. This algorithm aims at solving a CSP instance and performs a depth-first search with backtracking while maintaining arc consistency. More precisely, at each step of the search, a variable assignment is performed followed by a filtering process called constraint propagation which corresponds to enforcing arc-consistency. Algorithm 1 corresponds to a recursive version of the MAC algorithm (using binary branching). It returns $true$ iff the given constraint network P is satisfiable. More precisely, if P can be made arc consistent (i.e. AC(P) $\neq \bot$) then the search for a solution begins. It involves selecting a pair (x,a) and trying first $x = a$ and then $x \neq a$ (if no solution has been found with $x = a$). After any consistent assignment, the assigned variable is eliminated from the network and search is continued (line 5)[2]. When the current constraint network has no more variables (line 1), it means that a solution has been found.

2.2 Encoding CSP into SAT

Propositional satisfiability (SAT) is the problem of deciding whether a Boolean formula in conjunctive normal form (CNF) is satisfiable. A *CNF formula* Σ is a set (interpreted as a conjunction) of *clauses*, where a clause is a set (interpreted as a disjunction) of *literals*. A literal is a positive or negated propositional variable. A *truth assignment* of a Boolean formula is an assignment of truth values $\{true, false\}$ to its variables.

[2] $P|_{x=a}$ denotes the constraint network obtained from P by restricting the domain of x to the singleton $\{a\}$ whereas $P|_{x \neq a}$ denotes the constraint network obtained from P by removing the value a from the domain of x. $P \backslash x$ denotes the constraint network obtained from P by removing the variable x.

A *model* of a formula is a truth assignment that satisfies the formula. SAT is one of the most studied NP-Complete problems because of its theoretical and practical importance. Encouraged by the impressive progress in practical solving of SAT, various applications ranging from formal verification to planning are encoded and solved using SAT. CSP instances can also be reformulated as SAT instances.

In this paper, we consider the most commonly used encodings of CSP into SAT, namely, the direct encoding [7] and the support encoding [14]. In both SAT encodings of a constraint network $P = (\mathscr{X}, \mathscr{C})$, a propositional variable $x_{i,v}$ is associated to each pair (x_i,v) of P with $x_i \in \mathscr{X}$ and $v \in dom(x_i)$. The correspondence is the following: $x_{i,v}$ is true if x_i is assigned the value v (i.e. $x_i = v$) and $x_{i,v}$ is false if v is removed from dom(x_i) (i.e. $x_i \neq v$).

Direct Encoding The direct encoding [7] of a constraint network $P = (\mathscr{X}, \mathscr{C})$ involves two kinds of clauses:

- *At least one*: for each variable $x_i \in \mathscr{X}$ with dom(x_i) = $\{v_1, v_2, \ldots, v_d\}$, a clause of the form $x_{i,v_1} \vee x_{i,v_2} \cdots \vee x_{i,v_d}$ expresses the fact that the variable x_i must be assigned at least one value from its domain.
- *Conflict*: for each triplet (C,x_i,v) such that $C \in \mathscr{C}$, $x_i \in vars(C)$ and $v \in dom(x_i)$, we have, assuming that $vars(C) = \{x_i, x_j\}$ a clause $\neg x_{i,v} \vee \neg x_{j,w}$ for each value $w \in conflicts(C, x_i, v)$.

Support Encoding The idea of encoding support has been first introduced by Kasif in [19] and expanded on by Gent [14]. In support encoding, two kinds of clauses are introduced:

- *At most one*: for each variable $x_i \in \mathscr{X}$ and for any pair $\{v, w\} \subseteq dom(x_i)$, the following clause encodes the fact that the variable x_i must be assigned at most one value in $\{v, w\}$: $\neg x_{i,v} \vee \neg x_{i,w}$.
- *Support*: for each triplet (C,x_i,v) such that $C \in \mathscr{C}$, $x_i \in vars(C)$ and $v \in dom(x_i)$, we have, assuming that $vars(C) = \{x_i, x_j\}$ a clause $\neg x_{i,v} \vee x_{j,w_1} \vee x_{j,w_2} \vee \cdots \vee x_{j,w_k}$ where $supports(C, x_i, v) = \{w_1, w_2, \ldots, w_k\}$.

Note that the following set of clauses $\Sigma_D(C)$ is obtained from the constraint C of Example 1 using the direct encoding:

- *At least*: $(x_{1,a} \vee x_{1,b} \vee x_{1,c}) \wedge (x_{2,a} \vee x_{2,b} \vee x_{2,c} \vee x_{2,d})$
- *Conflict*: $(\neg x_{1,a} \vee \neg x_{2,c}) \wedge (\neg x_{1,a} \vee \neg x_{2,d}) \wedge (\neg x_{1,b} \vee \neg x_{2,a}) \wedge (\neg x_{1,b} \vee \neg x_{2,b}) \wedge (\neg x_{1,c} \vee \neg x_{2,a}) \wedge (\neg x_{1,c} \vee \neg x_{2,b})$

The following set of clauses $\Sigma_S(C)$ is obtained from C using the support encoding:

- *At most*: $(\neg x_{1,a} \vee \neg x_{1,b}) \wedge (\neg x_{1,a} \vee \neg x_{1,c}) \wedge (\neg x_{1,b} \vee \neg x_{1,c}) \wedge (\neg x_{2,a} \vee \neg x_{2,b}) \wedge (\neg x_{2,a} \vee \neg x_{2,c}) \wedge (\neg x_{2,a} \vee \neg x_{2,d}) \wedge (\neg x_{2,b} \vee \neg x_{2,c}) \wedge (\neg x_{2,b} \vee \neg x_{2,d}) \wedge (\neg x_{2,c} \vee \neg x_{2,d})$
- *Support*: $(\neg x_{1,a} \vee x_{2,a} \vee x_{2,b}) \wedge (\neg x_{1,b} \vee x_{2,c} \vee x_{2,d}) \wedge (\neg x_{1,c} \vee x_{2,c} \vee x_{2,d}) \wedge (\neg x_{2,a} \vee x_{1,a}) \wedge (\neg x_{2,b} \vee x_{1,a}) \wedge (\neg x_{2,c} \vee x_{1,b} \vee x_{1,c}) \wedge (\neg x_{2,d} \vee x_{1,b} \vee x_{1,c})$

Remark 1. Let us note that the *at most* (resp. *at least*) clauses are not required in direct (resp. support) encoding for checking satisfiability [30, 14]. One must add such clauses only if a mapping between SAT and CSP solutions is needed.

Support encoding admits interesting features. In [14], it is shown that encoding supports enables arc consistency in the original CSP instance to be established by unit propagation in the translated SAT instances. Last but not least, applying the well known DPLL algorithm to the obtained SAT instance behaves exactly like the MAC algorithm on the original CSP instance. We can also mention that support encoding has been extended to encode non binary constraints in SAT [3]. Interestingly enough, it has been proved in [9] that support clauses can be inferred from direct encoding using HyperBin resolution introduced by Bacchus [1]. These nice results open new interesting perspectives for establishing strong connections between SAT and CSP. The results that we present below on value ordering can be seen as a step in this direction.

3 Value Ordering Heuristics from SAT to CSP

3.1 SAT Branching Heuristics

Many branching heuristics has been proposed in SAT. One can cite the most recent ones, namely the VSIDS and UP heuristics used in Zchaff [31] and Satz solvers [22]. The first one uses literal occurrences in the set of learned no-goods whereas the second one measures the effect of unit propagation on the formula when a literal is assigned a truth value. Previously, CSAT [10] and POSIT [11], among other solvers, used simpler heuristics. Most of them are variants of the well-known Jeroslow-Wang (JW) heuristic [18], and evaluate a given literal according to syntactical properties (e.g. occurrence number of literals, clause length).

In SAT branching heuristics, the score, denoted $H(\Sigma, x)$, of a variable x of a CNF Σ is generally defined as a function f of the weight w associated with its positive and negative literals, i.e. $H(\Sigma, x) = f(w(x), w(\neg x))$. The next variable to assign is then chosen among variables with the greatest score. For example, the two sided Jeroslow-Wang rule is defined as : $H_{JW}(\Sigma, x) = w(x) + w(\neg x)$ where $w(x) = \sum_{x \in c} w(c)$, with $c \in \Sigma$ and $w(c) = 2^{-|c|}$. JW can be seen as a refinement of the MOMS (Maximum Occurrences in clauses of Minimal Size) heuristic [10]. Another basic heuristic that we consider in this paper is $H_{OCC}(\Sigma, x)$ that can be obtained from F_{JW} by instantiating $w(c)$ to 1. With H_{OCC}, we select in priority a variable with the greatest number of occurrences in the formula.

3.2 Mapping SAT Heuristics to CSP

Using direct and support encodings, we present now the CSP value ordering heuristics respectively corresponding to H_{OCC} and H_{JW} SAT branching heuristics. We have to emphasize that, when a CSP solver is based on a 2-way branching scheme, in order to simulate SAT branching heuristics, one needs to evaluate the score of $n * d$ pairs composed of a variable and a value. This can be very time consuming, especially if such evaluation is done at each node of the search tree. Hence, in the following, we

investigate the mapping of SAT branching heuristics to CSP, under the hypothesis that the CSP solver performs at each step of the search, a vertical selection (the choice of a variable) followed by a horizontal selection (the choice of a value for the selected variable). As a consequence, the *at least* and *at most* clauses can be omitted in our analysis since all literals occur the same number of times in such clauses. It is important to remark that this hypothesis corresponds to the current practice of CSP solvers.

Mapping $\mathbf{H_{OCC}}$ Let us show how the basic H_{OCC} SAT branching heuristic on direct (resp. support) encoding corresponds to the CSP value ordering heuristic max-$conflicts$ (resp. min-$conflicts$).

Property 1. Let $P = (\mathscr{X}, \mathscr{C})$ be a constraint network, Σ_D (resp. Σ_S) the CNF formula obtained from P using direct (resp. support) encoding. For a given variable $x_i \in \mathscr{X}$ and value $v \in dom(x_i)$, we have,

- $H_{OCC}(\Sigma_D, x_{i,v}) = 1 + \sum_{C \in \mathscr{C} | x_i \in vars(C)} |conflicts(C, x_i, v)|$
- $H_{OCC}(\Sigma_S, x_{i,v}) = \begin{pmatrix} 2 \\ |dom(x_i)| \end{pmatrix} + \sum_{C \in \mathscr{C} | x_i \in vars(C)} (1 + |supports(C, x_i, v)|)$

Proof. Indeed, for Σ_D, the positive literal $x_{i,v}$ occurs exactly one time positively (*at least* clause) and the negative literal $\neg x_{i,v}$ occurs the same number of times as the number of forbidden tuples with v in all constraints involving x_i (*conflict* clauses). For Σ_S, the negative literal $\neg x_{i,v}$ occurs $\begin{pmatrix} 2 \\ |dom(x_i)| \end{pmatrix}$ times in *at most* clauses and exactly one time in each constraint involving x_i. The number of positive occurrences of $x_{i,v}$ corresponds to the number of tuples supporting the value v of x_i for each constraint involving x_i. □

We can remark that, if the choice is restricted to the values of a given variable (i.e. if we adopt the current model based on a vertical selection followed by an horizontal one), then H_{OCC} on direct (resp. support) encoding delivers the same ordering as max-$conflicts$ (resp. min-$conflicts$). If it is not the case (i.e. if we perform a global selection among all pairs of the form (x,v)), then, whereas H_{OCC} and max-$conflicts$ always correspond with respect to direct encoding, H_{OCC} and min-$conflicts$ may deliver, with respect to support encoding, different orderings since the number of occurrences of a literal in the *at most* clauses depends on the size of the domains.

Mapping $\mathbf{H_{JW}}$ First, for the direct encoding, as the conflict clauses are all binary, H_{JW} admits the same behavior as H_{OCC} when the choice is restricted to the values of a given variable. On the other hand, when considering the support encoding, the length of the clauses, which depends on the number of supports of a value with respect to a given constraint, becomes important. Consequently, considering H_{JW} on support encoding, we derive a new interesting value ordering that corresponds to maximize the function SI defined as follows.

Definition 3. *Let $P = (\mathscr{X}, \mathscr{C})$ be a constraint network. For any pair (x_i,a), with $x_i \in \mathscr{X}$ and $a \in dom(x_i)$, we define:*

$$SI(P, x_i, v_i) = \sum_{c \in \mathscr{C} | vars(c) = \{x_i, x_j\}} [W_\downarrow(C, x_i, v_i) + W_\uparrow(C, x_i, v_i)]$$

where,

$$W_\downarrow(C, x_i, v_i) = w(1 + |supports(C, x_i, v_i)|) \text{ and}$$
$$W_\uparrow(C, x_i, v_i) = \sum_{v_j \in supports(C, x_i, v_i))} w(1 + |supports(C, x_j, v_j)|)$$

Here, w denotes any weighting function.

The following property establishes the connection between the SAT JW heuristic and the new derived CSP value ordering heuristic. Note that the factor α which is introduced below is such that for any variable $x_i \in \mathscr{X}$ and any pair $\{a, b\} \subseteq dom(x_i)$, we have $\alpha_{x_i,a} = \alpha_{x_i,b}$. It simply means that this factor can be discarded when a value must be selected in the domain of a variable (and this is what is done by SI).

Property 2. Let $P = (\mathscr{X}, \mathscr{C})$ be a constraint network and Σ_S the CNF formula obtained from P using the support encoding. For any pair (x_i,a), with $x_i \in \mathscr{X}$ and $a \in dom(x_i)$, we have:

$$H_{JW}(\Sigma_S, x_{i,v}) = SI(P, x_i, v) + \alpha_{x_i,v}$$

where $\alpha_{x_i,v}$ corresponds to the score of H_{JW} applied on *at most* clauses of Σ_S involving $\neg x_{i,v}$.

Proof. We have to show that $H_{JW}(\Sigma_S, x_{i,v}) - \alpha_{x_i,v} = SI(P, x_i, v)$. It simply means that we do not have to take into consideration the *at most* binary clauses involving $\neg x_{i,v}$. Consequently, we only need to consider the *support* clauses. The proof is a direct consequence of the following fact: for each constraint $C \in \mathscr{C} | vars(C) = \{x_i, x_j\}$, the negative literal $\neg x_{i,v}$ occurs exactly in one *support* clause of size $1 + |supports(C, x_i, v_i)|$, and the positive literal $x_{i,v}$ occurs in $|supports(C, x_i, v_i))|$ *support* clauses of different length (i.e. for each $v_j \in supports(C, x_i, v_i)$ the length of the clause is $1 + |supports(C, x_j, v_j)|$). Under the same weighting function $w(c) = 2^{-|c|}$, the two heuristics are equivalent, i.e. compute the same value for a given literal. □

As a summary, while the direct encoding naturally leads to $max\text{-}conflicts$ that adheres to the $fail\text{-}first$ policy, the support encoding also leads to adhere to this policy. Indeed, we have a correspondence between H_{JW} and SI (maximizing the score of H_{JW} is equivalent to maximizing SI). In practice (for our experimentation), we will use a simplified version of SI, denoted SI_s, obtained by substituting $w(1 + |supports(C, x_i, v_i)|)$ (resp. $w(1 + |supports(C, x_j, v_j)|)$) by $|supports(C, x_i, v_i)|$ (resp. $|supports(C, x_j, v_j)|$) in Definition 3. As a result, the new heuristic that we propose, called $min\text{-}inverse$, corresponds to minimize the value of SI_s since instead of using $w(c) = 2^{-|c|}$, we use $w(c) = |c|$.

To illustrate this, let us consider again Example 1. Applying the SI_s function on the variable x_1, we obtain $SI_s(C, x_1, a) = 4$, $SI_s(C, x_1, b) = 6$ and $SI_s(C, x_1, c) = 6$.

The best value according $min\text{-}inverse$ is then a. It is justified as follows. All values in dom(x_1), when assigned to x_1, lead to the removal of 2 (arc inconsistent) values from the domain of x_2. The main difference is that, while removing a from the domain of x_1 leads to the removal of 2 values from the domain of x_2, removing b or c from the domain of x_1 does not lead to any inconsistent value in the domain of x_2. More generally, given a constraint C such that $vars(C) = \{x_i, x_j\}$ and $v_i \in dom(x_i)$, minimizing $W_{\downarrow}(C, x_i, v_i)$ (resp. $W_{\uparrow}(C, x_i, v_i)$) increases the potential number of values of dom(x_j) that can be made arc inconsistent when considering $x_i = v_i$ (resp. $x_i \neq v_i$).

4 Experiments

To prove the practical interest of adhering to the $fail\text{-}first$ policy for value ordering, we have implemented the different heuristics described in the previous sections in our platform $Abscon$ and conducted an experimentation with respect to some scheduling instances and the full set of 1064 instances used as benchmarks of the first CSP Competition [28]. The search algorithm that has been employed is MAC equipped with $dom/wdeg$. All value ordering heuristics are implemented statically: an ordering is established prior to search and remains unchanged during the whole search process[23].

First, we searched to establish a comparison between all heuristics with respect to series of 100 open-shop scheduling instances randomly generated using Taillard's model [27] by fixing 5 jobs and 5 machines. For each instance, the optimal makespan OPT have been computed. We have considered different sets of instances by setting different time windows around the optimal makespan. For $x < OPT$, instances are unsatisfiable whereas for $x \geq OPT$, instances are satisfiable. Note that the hardest instances are those that are unsatisfiable and such that x is close to OPT.

Figure 2 shows the proportion of instances that have been solved (the higher, the better) in a limited amount of time (300 seconds). One can observe that $min\text{-}inverse$ is the most efficient heuristic on the hardest unsatisfiable instances, whereas the classical $min\text{-}conflicts$ is only able to solve a small number of these instances. Note that this statement is true even on hard satisfiable instances (for $1 \leq x \leq 1.01$). On easier satisfiable instances, following the $promise$ policy seems better.

Then, we tested the different heuristics on the 1064 instances of the first CSP Competition. Figure 3 shows the percentage of unsolved instances (the lower, the better) against search time. Although $min\text{-}inverse$ seems better than the other heuristics (in particular, between 200 and 350 allowed seconds), the difference is quite small.

So, let us zoom on the results obtained for a limited set of hard representative instances (Table 3). Here, the time-out was set to $1,000$ seconds. On random instances, there is no clear winner. In fact, on these instances, there is no structure concerning supports, and so, heuristics based on these are not very relevant. On academic instances, results are more spectacular, although sometimes chaotic. On *allIntervalSeries*, *pigeons* and *queen-knights* instances, $min\text{-}inverse$ is clearly better while $min\text{-}conflicts$ badly behaves. However, on *GolombRuler* or *BQWH* instances, $min\text{-}conflicts$ is more efficient (note that these are mainly satisfiable instances, so, the *promise* strategy may be more of practical interest in this case). On some real world instances such as FAPP and RLFAP problems, the impact of value ordering heuristics seems negligible. Finally,

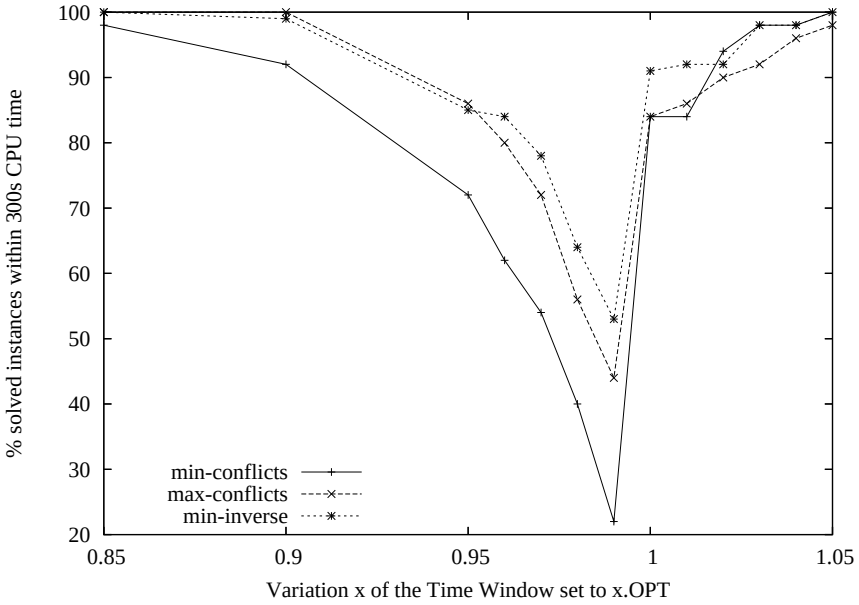


Fig. 2. Performance of value heuristics on 5x5 open-shop instances

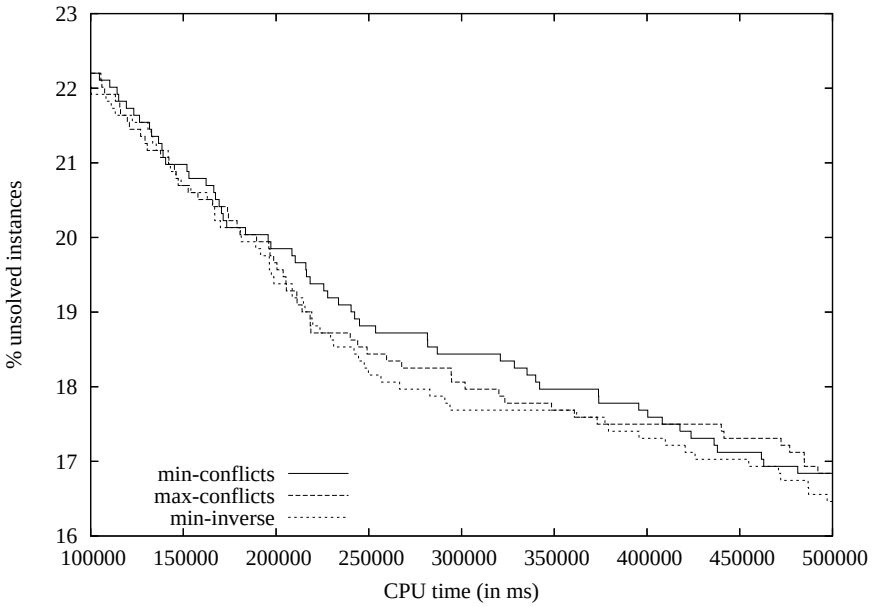


Fig. 3. Performance of value heuristics on competition's instances

$Instances$		$min\text{-}conflicts$	$max\text{-}conflicts$	$min\text{-}inverse$
Random Instances				
frb45-21-3	cpu	665	$timeout$	389
(sat)	$nodes$	896,278		511,101
frb45-21-5	cpu	$timeout$	243	258
(sat)	$nodes$		313,356	321,895
frb50-23-4	cpu	244	$timeout$	$timeout$
(sat)	$nodes$	276,441		
random-2-40-19-443-230-9	cpu	641	441	438
$(unsat)$	$nodes$	904,470	704,712	706,123
random-3-24-24-76-632-8	cpu	$timeout$	884	931
(sat)	$nodes$		1,537,568	1,558,190
Academic Instances				
series-15	cpu	849	59	3
(sat)	$nodes$	2,970,402	213,266	13,972
series-16	cpu	$timeout$	356	15
(sat)	$nodes$		1,074,057	59,314
bqwh-18-141-73	cpu	29	32	26
(sat)	$nodes$	115,916	136,821	106,254
bqwh-18-141-84	cpu	40	166	165
(sat)	$nodes$	157,398	685,424	685,424
gr-44-10	cpu	78	489	492
$(unsat)$	$nodes$	13,213	93,343	92,454
gr-55-10	cpu	25	$timeout$	$timeout$
(sat)	$nodes$	5,334		
pigeons-40	cpu	482	486	167
$(unsat)$	$nodes$	773,274	773,277	280,490
pigeons-45	cpu	255	258	98
$(unsat)$	$nodes$	304,584	304,588	119,820
qk-20-20-5-add	cpu	$timeout$	149	40
$(unsat)$	$nodes$		85,264	23,698
qk-20-20-5-mul	cpu	$timeout$	163	62
$(unsat)$	$nodes$		81,393	31,990
qa-5	cpu	4	2	3
(sat)	$nodes$	17,998	4,956	11,778
qa-6	cpu	245	61	197
(sat)	$nodes$	503,750	81,764	331,212
QCP-20-30	cpu	119	120	$timeout$
(sat)	$nodes$	237,759	237,759	
QCPp-20-14	cpu	$timeout$	$timeout$	6
(sat)	$nodes$			13,028
Real-world Instances				
scen11-f6	cpu	299	545	354
$(unsat)$	$nodes$	216,648	395,008	262,696
scen11-f7	cpu	163	344	220
$(unsat)$	$nodes$	112,952	251,098	164,593
163-TSP-25	cpu	282	530	280
(sat)	$nodes$	121,004	234,098	110,680
e0ddr1-10-by-5-10	cpu	19	643	$timeout$
(sat)	$nodes$	51,715	1,524,492	
e0ddr1-10-by-5-5	cpu	304	0	13
(sat)	$nodes$	1,244,563	50	36,888
enddr1-10-by-5-3	cpu	$timeout$	$timeout$	2
(sat)	$nodes$			1,757

Table 3. MAC with 2-way branching, using different value ordering heuristics

some job-shop instances from the competition show very chaotic results. In fact, we expect these problems to present an heavy-tailed behaviour.

To summarize, although the results must be tempered, we think that our experimentation allows to demonstrate that, with binary branching and an adaptive variable heuristic such as $dom/wdeg$, using a value ordering heuristic such as $max\text{-}conflict$ or $min\text{-}inverse$ that adheres to the $fail\text{-}first$ policy, is not penalizing and can even outperform the classical $min\text{-}conflicts$.

5 Conclusion

It is well known that the right approach to select values during search is to follow the $promise$ policy [2, 29] - the objective is to follow a path that maximizes the likelihood of finding a solution. However, we noticed that most of the experimental studies supporting this intuition have been based on d-way branching or/and non adaptive variable ordering heuristics.

In this paper, we first show that, on random instances, the anti-promise heuristic $max\text{-}conflicts$ was often as efficient as the standard promise $min\text{-}conflicts$ when 2-way branching and the heuristic $dom/wdeg$ were used. Our understanding of this phenomenon is that, as $dom/wdeg$ is able to efficiently refute unsatisfiable sub-trees, the overhead of refuting more unsatisfiable sub-trees (as, more often than not, we guide search toward unsatisfiable sub-trees) is compensated by the benefit of rapidly reducing the search space.

Then, after studying mappings from CSP to SAT, we devise a new heuristic, denoted $min\text{-}inverse$, that adheres to the $fail\text{-}first$ policy and that corresponds to the SAT Jeroslow-Wang branching heuristic. This correspondence supports our intuition that following the $fail\text{-}first$ policy for value ordering, in addition to variable ordering, can pay off. The results that we have obtained when experimenting a large sampling of problems indicate that $min\text{-}inverse$ and $max\text{-}conflicts$ can outperform $min\text{-}conflicts$.

As a perspective of this work, we project to study heuristics to perform global selection of pairs of the form (X,a) during search as it corresponds to the basic mechanism of 2-way branching. It means that we do not have anymore to distinguish between vertical and horizontal selection. The challenge is then to make such selections both cheap to realize and efficient in practice.

References

1. F. Bacchus. Enhancing Davis Putnam with extended binary clause reasoning. In *Proceedings of AAAI'02*, pages 613–619, 2002.
2. J.C. Beck, P. Prosser, and R.J. Wallace. Variable ordering heuristics show promise. In *Proceedings of CP'04*, pages 711–715, 2004.
3. C. Bessiere, E. Hebrard, and T. Walsh. Local consistencies in SAT. In *Selected revised papers from SAT'03*, pages 299–314, 2003.
4. C. Bessiere and J. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings of CP'96*, pages 61–75, 1996.
5. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.

6. D. Brelaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.
7. J. de Kleer. A comparison of ATMS and CSP techniques. In *Proceedings of IJCAI'89*, pages 290–296, 1989.
8. R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.
9. L. Drake, A. Frisch, I. Gent, and T. Walsh. Automatically reformulating SAT-encoded CSPs. In *Proceedings of the RCSP'02 workshop held with CP'02*, 2002.
10. O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. Sat versus unsat. In *Second DIMACS Challenge*, pages 299–314, 1993.
11. J.W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.
12. D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of IJCAI'95*, pages 572–578, 1995.
13. P.A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of ECAI'92*, pages 31–35, 1992.
14. I.P. Gent. Arc consistency in SAT. In *Proceedings of ECAI'02*, pages 121–125, 2002.
15. R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
16. T. Hulubei and B. O'Sullivan. Search heuristics and heavy-tailed behaviour. In *Proceedings of CP'05*, pages 328–342, 2005.
17. J. Hwang and D.G. Mitchell. 2-way vs d-way branching for CSP. In *Proceedings of CP'05*, pages 343–357, 2005.
18. R.G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.
19. S. Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45:275–286, 1990.
20. K. Kask, R. Dechter, and V. Gogate. Counting-based look-ahead schemes for constraint satisfaction. In *Proceedings of CP'04*, pages 317–331, 2004.
21. C. Lecoutre, F. Boussemart, and F. Hemery. Backjump-based techniques versus conflict-directed heuristics. In *Proceedings of ICTAI'04*, pages 549–557, 2004.
22. C.M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of IJCAI'97*, pages 366–371, 1997.
23. D. Meetah and M.R.C. van Dongen. Static value ordering heuristics for constraint satisfaction problems. In *Proceedings of CPAI'05 workshop held with CP'05*, pages 49–62, 2005.
24. N. Prcovic and B. Neveu. Progressive focusing search. In *Proceedings of ECAI'02*, pages 126–130, 2002.
25. D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP'94*, pages 10–20, 1994.
26. B.M. Smith and P. Sturdy. Value ordering for finding all solutions. In *Proceedings of IJCAI'05*, pages 311–316, 2005.
27. E. Taillard. Benchmarks for basic scheduling problems. *European journal of operations research*, 64:278–295, 1993.
28. M.R.C. van Dongen, editor. *Proceedings of CPAI'05 workshop held with CP'05*, volume II, 2005.
29. R.J. Wallace. Heuristic policy analysis and efficiency assessment in constraint satisfaction search. In *Proceedings of CPAI'05 workshop held with CP'05*, pages 79–91, 2005.
30. T. Walsh. SAT v CSP. In *Proceedings of CP'00*, pages 441–456, 2000.
31. L. Zhang, C.F. Madigan, M.W. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of ICCAD'01*, pages 279–285, 2001.

[blank page]

Representing Boolean Functions as Linear Pseudo-Boolean Constraints

Jan-Georg Smaus

Institut für Informatik, Universität Freiburg, Georges-Köhler-Allee 52, 79110 Freiburg im Breisgau, Germany, smaus@informatik.uni-freiburg.de

Abstract. A *linear pseudo-Boolean constraint* (LPB) is an expression of the form $a_1 \cdot l_1 + \ldots + a_m \cdot l_m \geq d$, where each l_i is a *literal* (it assumes the value 1 or 0 depending on whether a propositional variable x_i is true or false) and the $a_1, \ldots, a_m, d$ are natural numbers. The formalism can be viewed as a generalisation of a propositional clause. It has been said that LPBs can be used to represent Boolean functions more compactly than the well-known *conjunctive* or *disjunctive* normal forms. In this paper, we address the question: *how much* more compactly? We compare the expressiveness of a single LPB to that of related formalisms, and we give a statement that outlines how the problem of computing an LPB representation of a given CNF or DNF might be solved recursively. However, there is currently still a missing link for this to be a full algorithm.

1 Introduction

A *linear pseudo-Boolean constraint* (LPB) [1, 2, 4–7] is an expression of the form $a_1 l_1 + \ldots + a_m l_m \geq d$. Here each l_i is a *literal* of the form x_i or $\bar{x}_i \equiv 1 - x_i$, i.e. x_i becomes 0 if x_i is false and 1 if x_i is true, and vice versa for $\bar{x}_i$. Moreover, the $a_1, \ldots, a_m, d$ are natural numbers.

An LPB can be used to represent a Boolean[1] function; e.g. $x_1 + \bar{x}_2 + x_3 \geq 3$ represents the same function as the propositional formula $x_1 \wedge \neg x_2 \wedge x_3$ (in the following we identify propositional formulae with functions). It has been observed that a function can often be represented more compactly as a set of LPBs than as a *conjunctive* or *disjunctive* normal form (CNF or DNF) [4, 6, 7]. In fact, one can easily find an LPB which expresses a function that needs more than one clause to be expressed as DNF, say: $2x_1 + \bar{x}_2 + x_3 + x_4 \geq 2$ corresponds to the DNF $x_1 \vee (\neg x_2 \wedge x_3) \vee (\neg x_2 \wedge x_4) \vee (x_3 \wedge x_4)$.

The interest in Boolean functions, or propositional logic, comes from countless applications in verification and design automation concerning finite state systems [1–9, 13].

Previous works on LPBs [1, 4–7] have focused on generalising techniques applied in CNF-based propositional satisfiability solving [8, 9, 13] to LPBs, emphasising that this is beneficial because of the compactness of LPB representations:

[1] We will omit the qualification "Boolean" when it is clear from the context.

solving a problem based on such a compact representation can often be more efficient than based on a CNF or DNF encoding.

But where do the LPBs come from? One possibility is that for some application domain, one gives a direct representation of the problems as LPBs and in addition argues that the alternative representation as CNF would be less compact [1, 5, 7]. Another possibility is that one considers problem representations given as propositional formulae and transforms these into compact LPB representations. We are not aware that the latter has ever been proposed. In addition, almost all the arguments that we found in favour of LPBs were not *strictly* about LPBs but about *cardinality constraints*, which are a special case of LPBs.

This raises the question: how can a propositional formula be transformed into an LPB representation that is as compact as possible? In this paper, we approach this question from two extreme ends: on the one hand, we analyse, under various aspects, how much can be expressed by *a single* LPB. On the other hand, we consider propositional formulae for which the LPB representation saves nothing.

In fact, one cannot expect that a single LPB could model any useful problem—in particular, satisfiability of a single LPB is trivial. Our work is, we hope, a step towards the ultimate aim of representing an arbitrary Boolean function as a *set* (usually interpreted as conjunction but also a disjunction is thinkable) of LPBs.

In Sec. 3 we show that there is a strict inclusion chain from clauses to cardinality constraints to LPBs to the *monotone* Boolean functions, provided the dimension m is not too small. Monotone functions are those which can be represented by a formula where each variable occurs only in one polarity.

In Sec. 4, we show that in some cases an LPB representation can be exponentially more compact than a CNF or DNF representation, while in other cases one saves nothing at all.

In Sec. 5 we give a first result addressing the question of how a propositional formula can be converted to an LPB: if a DNF can be expressed by an LPB, then the dual CNF can be expressed by a very similar LPB, and vice versa.

In Sec. 6 we propose important components of what could become an algorithm for converting a DNF Φ to an LPB if possible. First of all, it is possible to determine the relative order of the coefficients a_i. Then, put simply, one has to split Φ into two subsets and remove the variable with the largest coefficient. This gives two subformulae for which LPBs can now be computed recursively. If two LPBs I^0, I^1 can be found which are in a certain sense very similar, then they can be composed into an LPB for Φ. The reason why this is not yet an algorithm is the following: if I^0, I^1 are not immediately "very similar", one has to see if there are I'^0, I'^1, equivalent to I^0, I^1, respectively, such that I'^0, I'^1 are "very similar". It is currently unclear how this can be done systematically.

Some proofs have been omitted here and can be found in [10].

2 Preliminaries

We assume the reader to be familiar with the basic notions of propositional logic. We write $\oplus$ to denote exclusive-or and $\leftrightarrow$ for equivalence.

An m-**dimensional Boolean function** f is a function $Bool^m \to Bool$. We say that f **properly depends** on the ith argument if there exists $\beta \in Bool^{i-1}$, $\beta' \in Bool^{m-i}$ with $f(\beta, 0, \beta') \neq f(\beta, 1, \beta')$.

We follow [4]. A **0-1 ILP constraint** is an inequality of the form

$$a_1 x_1 + \ldots + a_m x_m \geq d \qquad a_i, d \in \mathbb{R}, x_i \in Bool \ (Bool \equiv \{0,1\}). \tag{1}$$

We identify 0 with *false* and 1 with *true*. We call the a_i **coefficients** and d the **threshold**.

Using the relation $\bar{x}_i \equiv 1 - x_i$ and noting that it is sufficient to consider integer coefficients, one can rewrite a 0-1 ILP constraint as a **linear pseudo-Boolean constraint** (LPB)

$$a_1 l_1 + \ldots + a_m l_m \geq d \qquad a_i \in \mathbb{N}, d \in \mathbb{N}, l_i \in \{x_i, \bar{x}_i\}. \tag{2}$$

For example, $x_1 - 0.5x_2 - 0.5x_3 \geq 0$ can be written as $2x_1 + \bar{x}_2 + \bar{x}_3 \geq 2$. An occurrence of a **literal** x_i (resp., $\bar{x}_i$) is called an occurrence of x_i in **positive** (resp., **negative**) polarity.

An LPB where $a_i = 1$ or $a_i = 0$ for all $i \in [1..m]$ is called **cardinality constraint** (e.g. for $m = 4$: $1x_1 + 0x_2 + 1x_3 + 0x_4 \geq 1$, short $x_1 + x_3 \geq 1$).

Note that $l_1 + \ldots + l_m \geq 1$ corresponds to disjunction and $l_1 + \ldots + l_m \geq m$ to conjunction.

A **CNF** is a formula of the form $c_1 \wedge \ldots \wedge c_n$ where each **clause** c_j is a disjunction of literals. A **DNF** is defined dually; a conjunction of literals is called **(dual) clause**. Formally, CNFs and DNFs are sets of sets of literals, i.e. the order of clauses or the order of literals within a clause are insignificant. For DNFs and CNFs, we assume without loss of generality that no clause is a subset of another clause (the latter clause would be redundant since it is *subsumed*). Given a CNF, the **dual** DNF is obtained by swapping $\wedge$ and $\vee$. Any Boolean function can be represented by a DNF or CNF [12].

An **assignment** σ is a function $\{x_1, \ldots, x_m\} \to Bool$. The notion σ **satisfies** an LPB I is defined as expected [6].

3 Inclusion Results

The results of this section are not difficult but provide some useful insights into the expressiveness of an LPB or cardinality constraint.

Following [11], we define **monotone** functions as follows.

Definition 3.1. A Boolean function is **monotone** (*unate* [4]) if it can be written as $\vee, \wedge$-combination of literals, where each variable occurs in only one polarity. A monotone function is **isotone** if all variables appear with positive polarity.

In this section, we assume that each variable has positive polarity. This is no loss of generality since the polarity of a particular variable is an issue that is orthogonal to the inclusion results of this section: each monotone function has 2^m variants obtained by modifying the polarity of each variable.

We say that assignment σ **minimally** satisfies the LPB I if σ satisfies I and any assignment obtained from σ by changing some variable of I from *true* to *false* does not satisfy I. We say that a dual clause **corresponds** to an assignment if it consists of the variables assigned *true* by σ.

Proposition 3.2. An LPB I represents the DNF that consists of exactly those clauses that correspond to assignments that minimally fulfill I.

We now give the inclusion results.

Lemma 3.3. Each LPB represents a monotone function. For $m \geq 4$, there is at least one monotone function not represented by any LPB. For $m \leq 3$, each monotone function can be represented as LPB.

Lemma 3.4. Every cardinality constraint is an LPB. For $m \geq 3$, there is at least one LPB not expressible as cardinality constraint. For $m \leq 2$, each monotone function can be represented as a cardinality constraint.

4 Compactness of LPB Representations

4.1 Maximally Compact LPB Representations

Here, we consider the case that an LPB representation is exponentially more compact than a CNF or DNF representation. More precisely, encoding one cardinality constraint as CNF can entail an exponential blowup in formula size (not considering encodings involving auxiliary variables, encodings which are not equivalence preserving). Namely, encoding $x_1 + \ldots + x_m \geq k$ requires $\binom{m}{(m-k)+1} = \binom{m}{k-1}$ clauses of length $m - k + 1$ as CNF [2] and $\binom{m}{k}$ clauses of length k as DNF (in [6], this is said for CNF but in fact it should be DNF). Note that $\binom{m}{\lfloor m/2 \rfloor} \geq 2^{m/2}$.

In addition, one can show that the blowup when encoding an *LPB* as CNF or DNF is not worse. That is to say, while there are obviously DNFs that can be expressed by an LPB but not by a cardinality constraint, an LPB cannot, in general, express *bigger* DNFs than a cardinality constraint can. This result is omitted here for space reasons.

4.2 Non-compact LPB Representations

While the focus so far has been to see which Boolean functions can be represented as a single LPB, the ultimate goal must be to represent *any* Boolean function as a *set* of LPBs that is as small as possible.

Trivially, the LPB representation is at least as compact as a DNF or CNF representation since $l_1 \vee \ldots \vee l_m$ can be written as $l_1 + \ldots + l_m \geq 1$ and $l_1 \wedge \ldots \wedge l_m$ as $x_1 + \ldots + x_m \geq m$ (if one neglects the space taken by "$\geq m$" or "≥ 1").

Here we show that there are Boolean functions for which the LPB representation is not more compact than the CNF or DNF representation. We believe that

to be interesting, this statement must be strengthened: we show that there is a class of Boolean functions for which the minimal CNF or DNF representation has an exponential (in m) number of clauses and for which the representation as LPB set has the same number of LPBs.

Since one LPB can only represent a monotone function, the most straightforward idea is to use here the most "un-monotone" Boolean functions that exist, i.e., the functions for which any flipping of an argument will always flip the result. These are the functions denoted using $\oplus$ and $\leftrightarrow$: the formula $\bigoplus_{i\in[1..m]} x_i$ is true iff an odd number of variables from $x_1, \ldots, x_m$ is true, and $\leftrightarrow_{i\in[1..m]} x_i$ is true iff an even number of variables from $x_1, \ldots, x_m$ is true. It is rather straightforward to show that for these functions, an LPB representation saves nothing. We omit the formal statement here.

Much less obvious, we now show the desired result for a class of *monotone* functions.

To simplify the exposition, we assume that m is an even number, and we divide the variables into two equally sized subsets, which can be denoted most easily by distinguishing the variables with even index from the ones with odd index.

We take as starting point a class of Boolean functions for which the gain of compactness in using an LPB instead of a DNF representation is maximal: the cardinality constraints $x_1 + \ldots + x_m \geq m/2$, see Subsec. 4.1. The DNF representation has $\binom{m}{m/2}$ clauses of length $m/2$. Now we remove half of the dual clauses of this DNF. This is of course still an exponential number in m, and it turns out that the corresponding function cannot be represented any more compactly as disjunction of LPBs.

Lemma 4.1. Consider variables $x_1, \ldots, x_m$, and let Φ be the disjunction of all dual clauses of the form $x_{i_1} \wedge \ldots \wedge x_{i_{m/2}}$ such that $\sum_{j=1}^{m/2} i_j = 0 \mod 2$, i.e., Φ has $\binom{m}{m/2}/2$ clauses of length $m/2$. Then Φ requires $\binom{m}{m/2}/2$ LPBs of $m/2$ variables to be represented as disjunction of LPBs.

Proof. The DNF Φ corresponds to a disjunction of LPBs in the obvious way. This disjunction contains exactly one LPB for each assignment making true exactly a set of variables $x_{i_1}, \ldots, x_{i_{m/2}}$ with $\sum_{j=1}^{m/2} i_j = 0 \mod 2$.

We now show that there cannot be an LPB representation with fewer LPBs. Suppose an LPB I is true for two different assignments making true exactly all x_i with $i \in K$ and all x_j with $j \in J$, respectively, where $K, J \subseteq [1..m]$ are index sets of $m/2$ elements each such that $\sum K = 0 \mod 2$ and $\sum J = 0 \mod 2$. Since there are $m/2$ variables with even and the same number with odd index, it cannot be the case that both K, J contain only even indices or that both K, J contain only odd indices. Thus there exist $i \in K, j \in J$ with different parity. W.l.o.g. assume that $a_i \geq a_j$ (recall that a denotes the coefficients of LPB I). Then the assignment making true exactly all x_l with $l \in J \cup \{i\} \setminus \{j\}$ is a further assignment that fulfils I. But $\sum J \cup \{i\} \setminus \{j\} = 1 \mod 2$ and hence I cannot be part of a disjunction of LPBs representing Φ. □

5 Duality

We now return to the question of what functions can be represented by a single LPB. We show that if a DNF can be represented as an LPB, then the dual CNF can also be represented as an LPB, and the two LPBs are closely related.

As in Sec. 3, we assume that each variable has positive polarity.

Theorem 5.1. If a DNF can be represented by an LPB $I \equiv a_1x_1+\ldots+a_mx_m \geq d$, then the dual CNF is represented by the LPB $a_1x_1+\ldots+a_mx_m \geq \sum_{i=1}^{m} a_i + 1 - d$, and vice versa.

Proof. For a CNF $\Phi \equiv c_1 \wedge \ldots \wedge c_n$ or a DNF $\Phi \equiv c_1 \vee \ldots \vee c_n$, for any c_j, we call the set of variable indices occurring in c_j a *horizontal index set* of Φ. Moreover, we call any set $V \subseteq [1..m]$ such that $\forall j \in [1..n].\ \exists x_i \in c_j.\ i \in V$ holds a *vertical index set* of Φ.

Assume now the DNF $\Phi \equiv c_1 \vee \ldots \vee c_n$ is represented by $I \equiv a_1x_1 + \ldots + a_mx_m \geq d$. If we make all the variables in one c_j true, then Φ must be true. If we make all the variables in a vertical index set false, then Φ must be false. Hence for all horizontal index sets H and all vertical index sets V, it must hold that:

$$\sum_{i \in H} a_i \geq d \qquad \text{and} \qquad \sum_{i \notin V} a_i < d \tag{3}$$

Let Φ' be the CNF dual to Φ. Note that Φ, Φ' have the same horizontal and vertical index sets. If we make all the variables in one c_j false, then Φ' must be false. If we make all the variables in a vertical index set true, then Φ' must be true. So if $I' \equiv a'_1x_1 + \ldots a'_mx_m \geq d'$ is an LPB representing Φ', then for all horizontal index sets H and all vertical index sets V, it must hold that:

$$\sum_{i \notin H} a'_i < d' \qquad \text{and} \qquad \sum_{i \in V} a'_i \geq d' \tag{4}$$

We show that by setting $a'_i = a_i$ for $i \in [1..m]$ and $d' = \sum_{i=1}^{m} a_i + 1 - d$, (4) is fulfilled and thus I' is indeed an LPB representing Φ'.

Let H be an arbitrary horizontal index set of Φ. Then we have

$$\sum_{i \in H} a_i \geq d \Rightarrow \sum_{i \in H} a_i > d - 1 \Rightarrow 0 < \sum_{i \in H} a_i + 1 - d \Rightarrow \sum_{i \notin H} a_i < \sum_{i=1}^{m} a_i + 1 - d,$$

thus the first inequality of (4) holds. Now let V be an arbitrary vertical index set of Φ. Then we have

$$\sum_{i \notin V} a_i < d \Rightarrow \sum_{i \notin V} a_i \leq d - 1 \Rightarrow 0 \geq \sum_{i \notin V} a_i + 1 - d \Rightarrow \sum_{i \in V} a_i \geq \sum_{i=1}^{m} a_i + 1 - d,$$

thus the second inequality of (4) holds.

The proof of the converse is analogous. □

Note the border cases: $a_1x_1 + \ldots + a_mx_m \geq \sum a_i$ represents a conjunction (of literals), $a_1x_1 + \ldots + a_mx_m \geq 1$ represents a disjunction.

Example 5.2. Consider $5x_1 + 2x_2 + 2x_3 + 2x_4 \geq i$ for $i \in [1..11]$. Note first that for $i = 1, 2$ the represented function is the same, and the dual of that function is represented by setting $i = 11, 10$. Similarly one has $i = 3, 4$ vs. $i = 9, 8$. For $i = 5$, the DNF is $x_1 \vee (x_2 \wedge x_3 \wedge x_4)$, and the dual CNF $x_1 \wedge (x_2 \vee x_3 \vee x_4)$ is represented setting $i = 7$. For $i = 6$, the LPB represents $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_1 \wedge x_4) \vee (x_2 \wedge x_3 \wedge x_4)$. According to Thm. 5.1, since $12 - 6 = 6$, the dual CNF is represented by the same LPB, which means that the CNF is equivalent to its dual. This can easily be confirmed.

6 Representing a DNF as LPB

A reader coming from a SAT solving background might object to assuming a DNF as input, since satisfiability solving of DNFs is trivial. However, any results of this section can be applied to CNFs rather than DNFs using Sec. 5. As before, we assume that each variable has positive polarity.

By Prop. 3.2, there is a naïve semi-decision procedure for the problem of converting a DNF to the LPB it represents, involving enumeration of all LPBs. In this section we want to see if one can do better.

6.1 Determining the Order of Coefficients

Given a DNF Φ, one can determine a size order of the potential coefficients of an LPB representing Φ. That is to say, if Φ can be represented as LPB at all, then the coefficients must respect this order.

The following notion is useful for reasoning about the structure of a formula.

Definition 6.1. Variables x and y are **symmetric** in Φ if Φ is equivalent to the formula obtained by exchanging x and y. A set of variables X is **symmetric** in Φ if each pair is symmetric in Φ.

Since the clause order and the order within a clause of a DNF or CNF is insignificant, symmetry is a straightforward syntactic property.

The following lemma relates symmetric variables with identical coefficients.

Lemma 6.2. If LPB $I \equiv a_1x_1 + \ldots + a_mx_m \geq d$ represents a DNF Φ, then $a_i = a_k$ implies that x_i, x_k are symmetric in Φ; moreover, there exists an LPB $I' \equiv a'_1x_1 + \ldots + a'_mx_m \geq d'$ representing Φ such that symmetry of x_i, x_k in Φ implies $a'_i = a'_k$.

For example, $x_1 \vee x_2$ can be represented by $2x_1 + x_2 \geq 1$ or $x_1 + x_2 \geq 1$.

We want to measure how often a variable occurs in a DNF, taking the length of the clauses into account. For this it is useful to consider multisets of natural numbers. We write multisets as $\{\!\{\ldots\}\!\}$. We refrain from an exact formalisation, which is a technicality.

Definition 6.3. Let A, B be two multisets of numbers. We write $B \preceq A$ if B is obtained from A as follows: for each occurrence of a number n in A, either leave this occurrence in B, or replace it by an arbitrary (possibly 0) number of occurrences of numbers $> n$. We write $B \prec A$ if $B \preceq A$ and $A \not\preceq B$.

Example 6.4. We have $\{\!\{2,2,2,2\}\!\} \succ \{\!\{2,2,2\}\!\} \succ \{\!\{2,2,3\}\!\} \succ \{\!\{2,3\}\!\}$.

Note that $\{\!\{2,2,3\}\!\} \succ \{\!\{2,3\}\!\}$ can be established in two ways: removing one occurrence of 2 from $\{\!\{2,2,3\}\!\}$, or removing the occurrence of 3 from $\{\!\{2,2,3\}\!\}$ and replacing one occurrence of 2 from $\{\!\{2,2,3\}\!\}$ by 3.

Definition 6.5. For a DNF Φ, the multiset $OP(\Phi, x)$ is the **occurrence pattern** of x if $OP(\Phi, x)$ has an occurrence of n for each clause of length n in Φ that contains x.

Example 6.6. Consider $\Phi \equiv (x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_1 \wedge x_4) \vee (x_1 \wedge x_5) \vee (x_2 \wedge x_3) \vee (x_2 \wedge x_4) \vee (x_3 \wedge x_4 \wedge x_5)$. The occurrence patterns are $OP(\Phi, x_1) = \{\!\{2,2,2,2\}\!\}$, $OP(\Phi, x_2) = \{\!\{2,2,2\}\!\}$, $OP(\Phi, x_3) = OP(\Phi, x_4) = \{\!\{2,2,3\}\!\}$, and $OP(\Phi, x_5) = \{\!\{2,3\}\!\}$. Φ can be represented by $4x_1 + 3x_2 + 2x_3 + 2x_4 + x_5 \geq 5$.

The next lemma says that the coefficients of an LPB representing a DNF must correspond to the order given by the occurrence patterns.

Lemma 6.7. Let Φ be a DNF represented by the LPB $I \equiv a_1 x_1 + \ldots + a_m x_m \geq d$. Then $a_i \geq a_k$ implies $OP(\Phi, x_i) \succeq OP(\Phi, x_k)$; moreover, there exists an LPB $I' \equiv a'_1 x_1 + \ldots + a'_m x_m \geq d'$ representing Φ such that $OP(\Phi, x_i) = OP(\Phi, x_k)$ implies $a'_i = a'_k$.

Proof. Without loss of generality assume $a_1 \geq \ldots \geq a_m$, and consider some x_i, x_k with $i < k$, i.e. $a_i \geq a_k$. We compare $OP(\Phi, x_i)$ and $OP(\Phi, x_k)$.

For each clause of length n in Φ that contains x_i and x_k, we have an occurrence of n in $OP(\Phi, x_i)$ and a uniquely matching occurrence of n in $OP(\Phi, x_k)$.

For each clause of length n in Φ that contains x_i, and for which replacing x_i with x_k gives another clause in Φ, we have an occurrence of n in $OP(\Phi, x_i)$ and a uniquely matching occurrence of n in $OP(\Phi, x_k)$.

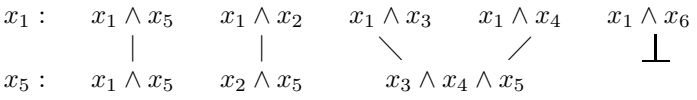
Now consider a clause of length n in Φ that contains x_i, and for which replacing x_i with x_k does not give another clause in Φ (the reason for this must be that the corresponding assignment does not fulfill I). For such a clause, replacing x_i with x_k plus additional variables may give a clause in Φ, in which case the occurrence of n in $OP(\Phi, x_i)$ is mapped to one or more occurrences of numbers $> n$ in $OP(\Phi, x_k)$ (note however that conversely, an occurrence of such a number in $OP(\Phi, x_k)$ is not necessarily *uniquely* mapped to the occurrence of n in $OP(\Phi, x_i)$). Or it may be the case that no way of replacing x_i with x_k plus additional variables gives a clause in Φ, in which case the occurrence of n in $OP(\Phi, x_i)$ is mapped to 0 occurrences of numbers $> n$ in $OP(\Phi, x_k)$.

Summarising, we have $OP(\Phi, x_i) \succeq OP(\Phi, x_k)$, showing the first statement.

In particular, if there exists a clause of length n in Φ that contains x_i, and for which replacing x_i with x_k does not give another clause in Φ, then $OP(\Phi, x_i) \succ$

$OP(\Phi, x_k)$. Therefore $OP(\Phi, x_i) = OP(\Phi, x_k)$ implies that for each clause of length n in Φ that contains x_i, swapping x_i with x_k also gives a clause in Φ. I.e., x_i and x_k are symmetric. Hence an LPB I' as required exists by Lemma 6.2. □

Example 6.8. Consider $(x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_1 \wedge x_4) \vee (x_1 \wedge x_5) \vee (x_1 \wedge x_6) \vee (x_2 \wedge x_3) \vee (x_2 \wedge x_4) \vee (x_2 \wedge x_5) \vee (x_2 \wedge x_6) \vee (x_3 \wedge x_4 \wedge x_5) \vee (x_3 \wedge x_4 \wedge x_6)$. The following picture illustrates the occurrence patterns of x_1 and x_5, and thereby all four cases of the proof of Lemma 6.7:



The crucial point is the following: it is not possible that a clause containing x_1 is mapped to a shorter clause containing x_5, since $a_1 \geq a_5$.

Occurrence patterns capture in which ways replacing one variable in a clause of Φ yields or does not yield another clause of Φ. In Ex. 6.6, $OP(\Phi, x_1) \succ OP(\Phi, x_2)$ reflects that replacing x_1 by x_2 does not always yield a clause of Φ.

The following is a corollary of Lemmas 6.2 and 6.7.

Corollary 6.9. If the DNF Φ is represented by an LPB I, then Φ is also representable by an LPB such that x_i, x_k are symmetric in Φ iff x_i, x_k have identical occurrence patterns iff x_i, x_k have identical coefficients.

The results so far can be used to make statements about which DNFs can definitely not be represented as a single LPB. For example, it has been said that a single LPB can express an implication [6]. In [7], implications of the form $y \rightarrow (x_1 \wedge x_2)$ are expressed as LPB. However, a simple implication like $(x_1 \vee x_2) \rightarrow (y_1 \wedge y_2)$ is beyond what can be expressed by a single LPB. We omit the general statement of this for space reasons.

6.2 Computing an LPB Recursively

We aim for an algorithm that given a DNF Φ finds an LPB representing Φ if possible. By Lemma 6.7 we can use the occurrence patterns to establish the relative order of the coefficients. Assume the numbering of the variables is such that we have $OP(\Phi, x_1) \succeq \ldots \succeq OP(\Phi, x_m)$. Consider now the set $X = \{x_1, \ldots, x_l\}$ such that $OP(\Phi, x_1) = \ldots = OP(\Phi, x_l)$ (=: $OP(\Phi, X)$). It might be that $l = 1$, which corresponds to the intuitive explanation in the introduction. If X is not symmetric in Φ, we know by Cor. 6.9 that Φ cannot be represented as LPB and we can stop. Otherwise, we partition Φ according to how many variables from X each clause contains. We then remove the variables from X from each clause and solve the problem of finding an LPB for each partitition recursively. It turns out that the LPB for Φ can be obtained by combining the LPBs for the subproblems, although it is not obvious that this combination is an effective operation.

Definition 6.10. Given a DNF Φ and a subset X of its variables, we denote by $\mathit{Cut}(\Phi, X, k)$ the disjunction of clauses from Φ containing exactly k variables from X, with those variables removed.

Example 6.11. Let $\Phi \equiv (x_2 \wedge x_3 \wedge x_4) \vee (x_2 \wedge x_5) \vee (x_1 \wedge x_2) \vee (x_1 \wedge x_3 \wedge x_4)$ and $X = \{x_1\}$. Then $\mathit{Cut}(\Phi, X, 0) = (x_2 \wedge x_3 \wedge x_4) \vee (x_2 \wedge x_5)$ and $\mathit{Cut}(\Phi, X, 1) = x_2 \vee (x_3 \wedge x_4)$.

We illustrate our way of proceeding by giving examples.

Example 6.12. Consider the DNF $\Phi \equiv (x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_1 \wedge x_4) \vee (x_2 \wedge x_3 \wedge x_4)$ and $X = \{x_1\}$. Then $\mathit{Cut}(\Phi, X, 0) = x_2 \wedge x_3 \wedge x_4$, represented by $x_2 + x_3 + x_4 \geq 3$. Moreover, $\mathit{Cut}(\Phi, X, 1) = x_2 \vee x_3 \vee x_4$, represented by $x_2 + x_3 + x_4 \geq 1$.

Since the coefficients of the two LPBs agree, it turns out that Φ can be represented by $2x_1 + x_2 + x_3 + x_4 \geq 3$. The coefficient of x_1 is given by the difference of the two thresholds, i.e. $3 - 1$.

In the following example, X contains more than one variable.

Example 6.13. Consider $\Phi \equiv (x_1 \wedge x_2) \vee (x_1 \wedge x_3 \wedge x_4) \vee (x_2 \wedge x_3 \wedge x_4)$ and $X = \{x_1, x_2\}$. We have $\mathit{Cut}(\Phi, X, 0) = \mathit{false}$, represented by $x_3 + x_4 \geq 4$, $\mathit{Cut}(\Phi, X, 1) = x_3 \wedge x_4$, represented by $x_3 + x_4 \geq 2$, and $\mathit{Cut}(\Phi, X, 2) = \mathit{true}$, represented by $x_3 + x_4 \geq 0$. The DNF Φ is represented by $2x_1 + 2x_2 + x_3 + x_4 \geq 4$. The coefficient of x_1, x_2 is given by $4 - 2 = 2 - 0 = 2$ (we say that the thresholds are *equidistant*).

The previous example shows where the problem for an effective procedure of combining the LPBs lies: for each subproblem, the LPBs are not always the simplest ones; one has to be "smart" to find LPBs that agree in their coefficients.

We now give the theorem.

Theorem 6.14. Let Φ be a DNF in variables $x_1, \ldots, x_m$ and suppose $X = \{x_1, \ldots, x_l\}$ are symmetric variables such that $\mathit{OP}(\Phi, X)$ is maximal w.r.t. $\preceq$ in Φ. Then Φ is represented by an LPB $a_1 x_1 + \ldots + a_m x_m \geq d$, where $a_1 = \ldots = a_l$, iff for all $k \in [0..l]$, the DNF $\mathit{Cut}(\Phi, X, k)$ is represented by $a_{l+1} x_{l+1} + \ldots + a_m x_m \geq d - k \cdot a_1$.

Proof. For an assignment σ and a set of variables V, we denote by $\sigma \setminus V$ the assignment that is undefined on V and else equal to σ. We denote by $\sigma \cup \{V \mapsto \mathit{true}\}$ the assignment that maps all variables of V to *true* and is else equal to σ. For a clause c, we denote by $c \setminus V$ the clause obtained from c by deleting the variables in V.

Since the variables in X are symmetric, the following holds:

$$\text{If } c \in \mathit{Cut}(\Phi, X, k), \text{ then for every } V \subseteq X \text{ with } \#V = k,\ c \cup V \in \Phi. \quad (5)$$

Throughout, we use Prop. 3.2.

"$\Leftarrow$": We assume that each $\mathit{Cut}(\Phi, X, k)$ is represented by $a_{l+1} x_{l+1} + \ldots + a_m x_m \geq d - k \cdot a_1$, and show that the assignments minimally fulfilling $a_1 x_1 + \ldots + a_m x_m \geq d$ correspond exactly to the clauses of Φ.

a) Consider an assignment σ that minimally fulfils $a_1x_1 + \ldots + a_mx_m \geq d$ and makes exactly k variables from X true, say, the set $V \subseteq X$. Let c be the clause corresponding to σ. Then $\sigma \setminus V$ minimally fulfils $a_{l+1}x_{l+1} + \ldots + a_mx_m \geq d - k \cdot a_1$, and hence $c \setminus V \in \mathit{Cut}(\Phi, X, k)$. By (5) this implies $c \in \Phi$.
b) Conversely, consider a clause c in Φ that contains exactly k variables from X, say the set $V \subseteq X$. Let σ be the assignment that corresponds to c. Then $c \setminus V \in \mathit{Cut}(\Phi, X, k)$ and thus $\sigma \setminus V$ minimally fulfils $a_{l+1}x_{l+1} + \ldots + a_mx_m \geq d - k \cdot a_1$, and thus σ minimally fulfils $a_1x_1 + \ldots + a_mx_m \geq d$.

"⇒": We assume that Φ is represented by $a_1x_1 + \ldots + a_mx_m \geq d$ and show that for each k the assignments minimally fulfilling $a_{l+1}x_{l+1} + \ldots + a_mx_m \geq d - k \cdot a_1$ correspond exactly to the clauses of $\mathit{Cut}(\Phi, X, k)$.

c) For arbitrary $k \in [1..l]$, let σ be an assignment that minimally fulfils $a_{l+1}x_{l+1} + \ldots + a_mx_m \geq d - k \cdot a_1$ and c the clause corresponding to σ. Then for any $V \subseteq X$ with $\#V = k$, it holds that $\sigma \cup \{V \mapsto \mathit{true}\}$ fulfils $a_1x_1 + \ldots + a_mx_m \geq d$, and since $a_1 = \ldots = a_l > a_{l+1} \geq \ldots \geq a_m$, $\sigma \cup \{V \mapsto \mathit{true}\}$ fulfils $a_1x_1 + \ldots + a_mx_m \geq d$ *minimally* and hence $c \cup V \in \Phi$ and hence $c \in \mathit{Cut}(\Phi, X, k)$.
d) Conversely, consider a clause $c \in \mathit{Cut}(\Phi, X, k)$ and let σ be the assignment corresponding to c. Then by (5), for any $V \subseteq X$ with $\#V = k$, we have that $c \cup V \in \Phi$ and thus $\sigma \cup \{V \mapsto \mathit{true}\}$ is an assignment that minimally fulfils $a_1x_1 + \ldots + a_mx_m \geq d$, and thus σ minimally fulfils $a_{l+1}x_{l+1} + \ldots + a_mx_m \geq d - k \cdot a_1$. □

Unfortunately, Thm. 6.14 does not immediately dictate an algorithm for deciding whether a DNF can be represented as LPB and if yes, computing the LPB. The reason is that a DNF might be represented by alternative LPBs, and so even if the LPBs computed recursively do not have agreeing coefficients and equidistant thresholds, it might be possible to find alternative LPBs for which this is the case, a process we call *unification* of LPBs (see Ex. 6.13).

The unification problem can be divided into the problems of making the coefficients agree and making the thresholds equidistant. Concerning the second problem, the idea is to record, for each LPB, to what extent its threshold can be shifted without changing the meaning. We then try to shift the thresholds of the LPBs to be unified to make the thresholds equidistant. We have some formal statements solving this problem, but in this workshop paper, we prefer to focus on the first problem, for which we have no complete solution yet.

6.3 Making Coefficients Agree

Some adaptations of LPBs are straightforward: disjunctions and conjunctions, *true*, and *false*, can be represented using arbitrary coefficients and choosing d as $\min a_i$, $\sum a_i$, some number ≤ 0, some number $> \sum a_i$, respectively.

We have the following thesis for how unification of several LPBs works, which we have confirmed on a dozen examples: if the coefficients of several LPBs can

be made to agree, then the coefficients of the resulting LPB can be obtained by a linear combination of the individual LPBs. Formally, given LPBs $I^k \equiv a^k_1 x_1 + \ldots + a^k_m x_m \geq d^k$, for $k \in [0..l]$, if $a_1, \ldots, a_m$ exist such that each I^k is equivalent to $I'^k \equiv a_1 x_1 + \ldots + a_m x_m \geq d'^k$, then $a_1, \ldots, a_m, d'^0, \ldots, d'^l$ can be chosen so that there are numbers $\mu^0, \ldots, \mu^l$ such that for all $j \in [1..m]$, we have $a_j = \mu^0 a^0_j + \ldots + \mu^l a^l_j$. This will be the main focus of our future work.

Example 6.15. Consider Φ represented by $7x_1+6x_2+4x_3+4x_4+3x_5+3x_6+3x_7 \geq 15$. It turns out that the minimal representation of $Cut(\Phi, x_1, 0)$ is $I^0 \equiv 2x_2+x_3+x_4+x_5+x_6+x_7 \geq 5$ and that of $Cut(\Phi, x_1, 1)$ is $I^1 \equiv 4x_2+3x_3+3x_4+2x_5+2x_6+2x_7 \geq 6$. There are alternative LPBs $I'^0 \equiv 6x_2+4x_3+4x_4+3x_5+3x_6+3x_7 \geq 15$, $I'^1 \equiv 6x_2 + 4x_3 + 4x_4 + 3x_5 + 3x_6 + 3x_7 \geq 8$, and so Thm. 6.14 is applicable. Note that $lhs(I'^0) = lhs(I'^1) = 1 \cdot lhs(I^0) + 1 \cdot lhs(I^1)$.

In Ex. 6.15, for I^0 we have $a_3 = \ldots = a_7$, whereas for I^1 we have $a_3 = a_4 \neq a_5 = a_6 = a_7$. Generally, given two subproblems $Cut(\Phi, X, k)$, $Cut(\Phi, X, k')$, it can happen that we have two variable sets Y, Z with $OP(Cut(\Phi, X, k), Y) \neq OP(Cut(\Phi, X, k), Z)$ while $OP(Cut(\Phi, X, k'), Y) = OP(Cut(\Phi, X, k'), Z)$. That is to say, one subproblem says: Y, Z should have different coefficients, the other says: Y, Z should have the same coefficient.

This raises the question: how different or how similar must coefficients be? The following lemma is one key to the answer.

Lemma 6.16. Let $\Phi \equiv (\bigwedge_{i \in J_1} x_i \wedge c) \vee (\bigwedge_{i \in J_2} x_i \wedge c) \vee \ldots$. If Φ is representable as LPB $I \equiv a_1 x_1 + \ldots + a_m x_m \geq d$, then the following hold:

$$\sum_{i \in J_1} a_i > \sum_{i \in J_2} a_i - \min_{i \in J_2} a_i \quad \text{and} \quad \sum_{i \in J_2} a_i > \sum_{i \in J_1} a_i - \min_{i \in J_1} a_i. \tag{6}$$

Proof. We write J_c for the set of variable indices in c. Then, since $\bigwedge_{i \in J_1} x_i \wedge c$ and $\bigwedge_{i \in J_2} x_i \wedge c$ are clauses of Φ, the following hold

$$\begin{array}{l} \sum_{i \in J_c} a_i + \sum_{J_1} a_i \geq d > \sum_{i \in J_c} a_i + \sum_{J_1} a_i - \min_{i \in J_1} a_i \\ \sum_{i \in J_c} a_i + \sum_{J_2} a_i \geq d > \sum_{i \in J_c} a_i + \sum_{J_2} a_i - \min_{i \in J_2} a_i. \end{array}$$

These imply the result. □

The following two lemmas are consequences. Both can be useful to show how coefficients must be modified, or show that such a modification is impossible.

Lemma 6.17. Let Φ be a DNF and Y and X be sets of symmetric variables of it. Suppose that Φ contains clauses $c \wedge c_X$, $c \wedge c_Y$ with $c_X \subseteq X$, $c_Y \subseteq Y$, and that Φ can be represented as an LPB $I = a_1 x_1 + \ldots a_m x_m \geq d$, where all variables in X (resp., Y) have the same coefficient a_X (resp., a_Y). Then (assuming for convenience that division by 0 gives ∞)

$$\frac{|c_Y| - 1}{|c_X|} < \frac{a_X}{a_Y} < \frac{|c_Y|}{|c_X| - 1} \tag{7}$$

Example 6.18. Let Φ be the DNF such that $Cut(\Phi, x_1, 0)$ is represented by $I^0 \equiv 3x_2 + 3x_3 + 3x_4 + 2x_5 + 2x_6 + 2x_7 + 2x_8 \geq 9$ and $Cut(\Phi, x_1, 1)$ by $I^1 \equiv 4x_2 + 4x_3 + 4x_4 + x_5 + x_6 + x_7 + x_8 \geq 4$. Inspecting those DNFs shows by Lemma 6.17 that $Cut(\Phi, x_1, 0)$ requires $\frac{a_2}{a_5} < \frac{3}{2-1}$ and $Cut(\Phi, x_1, 1)$ requires $\frac{4-1}{1} < \frac{a_2}{a_5}$ which is a contradiction, showing that I^0 and I^1 cannot be unified in the sense of Thm. 6.14 and so Φ cannot be represented by a single LPB.

Lemma 6.19. Let Φ be a DNF and $X = \{x_1, \ldots, x_l\}$ a set of symmetric variables of it. Suppose Φ is represented by an LPB I with $a_1 \geq \ldots \geq a_l$. Then for each k such that Φ contains a clause with exactly k variables from X, we have that $\sum_{i=l-k+1}^{l} a_i > \sum_{i=1}^{k-1} a_i$ (in words, the sum of the k smallest coefficients must be greater than the sum of the $k-1$ greatest coefficients).

Note that the cases $k = 1$ and $k = l$ are genuine special cases: $\sum_{i=l}^{l} a_i > \sum_{i=1}^{0} a_i$ and $\sum_{i=1}^{l} a_i > \sum_{i=1}^{l-1} a_i$ are tautologies and thus nothing interesting can be derived from them. However, if Φ contains some clause of at least 2 and at most $l-1$ variables from X, we can conclude $a_{l-1} + a_l > a_1$ which is quite a strong statement.

Consider Ex. 6.15. For $Cut(\Phi, \{x_1\}, 0)$, one must find a_3, a_5 such that $3a_5 > 2a_3$. The coefficients of I^1, $a_5 = 2$ and $a_3 = 3$, differ too much. But $a_5 = 3$ and $a_3 = 4$ of I'^0, I'^1 are close enough.

So we know some points about how unification of LPBs works, and we can use these to compute LPBs in a more or less ad-hoc way, but we have not solved the problem of making coefficients agree, in general.

7 Conclusion

Linear pseudo-Boolean constraints have attracted interest because they can be used to represent Boolean functions more compactly than CNFs or DNFs, and because techniques applied in CNF-based propositional satisfiability solving can be generalised to LPBs, which can be more efficient than solving a problem based on a CNF representation [1, 4–7]. What is implicit is that there is a problem from some application domain, which has a natural encoding as LPB, and for which the CNF encoding would be larger. There are mainly three points that gave us the impression that the theoretical understanding of LPBs could be improved.

Firstly, several authors have emphasised that an LPB representation of a function can be exponentially more concise than a CNF (or DNF) representation [1, 4–7]. However, it is shown in fact that *cardinality constraints* already can be exponentially more concise than a CNF. Also the examples are mostly about cardinality constraints rather than full LPBs. Thus it is not argued convincingly that the additional expressive power that LPBs have compared to cardinality constraints is useful.

Secondly, it has been noted *en passant* that a single LPB can be used to express an implication [6], but it remains unclear what kind of implications can or cannot be expressed. In fact, an implication like $(x_1 \vee x_2) \rightarrow (y_1 \wedge y_2)$ cannot be expressed.

But most importantly, since an LPB representation can be more compact than a CNF representation, could one not apply the approach of [1, 4–7] to problems that are naturally encoded as CNF, i.e. could one not convert a CNF to an LPB and then apply [1, 4–7]? And if a CNF cannot be represented as a single LPB, can one find a relatively small *set* of LPBs that represents the CNF? As became apparent in Subsec. 6.3, our answer is not conclusive, but it is here that we see the potential for practical application of our work.

We add some more evidence to the first point above. Barth [2] mentions that LPBs arise in artificial intelligence applications [3]. Since he used a solver that could only deal with cardinality constraints, he proposes a transformation of LPBs to cardinality constraints.

In [7], LPBs are used for bounded model checking. At one point, an LPB of the form $x_1 + x_2 + 2\bar{y} \geq 2$ (which is not a cardinality constraint) is used.

Apart from that, the above works say little about where the problem instances come from, and if anything, then these are in fact cardinality constraints rather than LPBs. In [1], problems Min-Cover, Max-SAT and MAX-ONEs are mentioned. E.g., Max-SAT is the problem of finding a variable assignment that maximises the number of satisfied clauses of an unsatisfiable SAT instance. Furthermore, applications from design automation [4], the pigeonhole problem [5] and gate level netlists [6] are mentioned as applications.

However, we are not suggesting that our approach of converting a CNF or DNF to an LPB is the only way to go. If for a problem domain, there is a natural direct translation to an LPB not going via CNF or DNF, then this should definitely be considered.

We summarise our contributions to the understanding of LPBs. We demonstrated that the functions expressible as one LPB constraint are a strict subset of the monotone functions. We showed that in some cases an LPB representation can be exponentially more compact than a CNF or DNF representation, while in other cases the LPB representation has exponential size just like the CNF or DNF representation. We showed that the problems of encoding a DNF or a CNF as LPB have a very simple duality. Finally, we outlined a recursive procedure for computing an LPB representation for a DNF, where the link currently still missing is the problem of making coefficients agree.

Acknowledgements I thank Markus Behle, Martin Fränzle, Marc Herbstritt, Christian Herde, Bernhard Nebel, and the other colleagues from the AVACS project, for useful discussions.

References

1. Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Generic ILP versus specialized 0-1 ILP: an update. In Lawrence T. Pileggi and Andreas Kuehlmann, editors, *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design*, pages 450–457. ACM, 2002.

2. Peter Barth. Linear 0-1 inequalities and extended clauses. In Andrei Voronkov, editor, *Proceedings of the 4th International Conference on Logic Programming and Automated Reasoning*, volume 698 of *LNCS*, pages 40–51. Springer-Verlag, 1993.
3. Peter Barth and Alexander Bockmayr. Solving 0-1 problems in CLP(PB). In *Proceedings of the 9th Conference on Artficial Intelligence for Applications*. IEEE, 1993.
4. Donald Chai and Andreas Kuehlmann. A fast pseudo-Boolean constraint solver. In *Proceedings of the 40th Design Automation Conference*, pages 830–835. ACM, 2003.
5. Heidi E. Dixon and Matthew L. Ginsberg. Combining satisfiability techniques from AI and OR. *The Knowledge Engineering Review*, 15:31–45, 2000.
6. Martin Fränzle and Christian Herde. Efficient SAT engines for concise logics: Accelerating proof search for zero-one linear constraint systems. In Moshe Y. Vardi and Andrei Voronkov, editors, *Proceedings of the 10th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 2850 of *LNCS*, pages 302–316. Springer-Verlag, 2003.
7. Martin Fränzle and Christian Herde. Hysat: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 2006. To appear.
8. João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
9. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535. ACM, 2001.
10. Jan-Georg Smaus. Representing Boolean functions as linear pseudo-Boolean constraints. Technical Report 227, Institut für Informatik, Universität Freiburg, 2006.
11. Vetle Ingvald Torvik and E. Trintaphyllou. Inference of monotone Boolean functions. In Chris A. Floudas and Panos M. Pardalos, editors, *Encyclopedia of Optimization*, pages 472–480. Kluwer Academic Publishers, 2001.
12. Ingo Wegener. *The Complexity of Boolean Functions*. Wiley & Sons, `http://eccc.uni-trier.de/eccc-local/ECCC-Books/wegener_book_readme.html`, 1987.
13. Hantao Zhang. SATO: An efficient propositional prover. In William McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction*, volume 1249 of *LNCS*, pages 272–275. Springer-Verlag, 1997.

[blank page]

Dealing with SAT and CSPs in a single framework

Belaïd BENHAMOU, Lionel PARIS, and Pierre SIEGEL

Université de Provence,
LSIS - UMR CNRS 6168,
Marseille, FRANCE,
email: {Belaid.Benhamou,Lionel.Paris,Pierre.Siegel}@cmi.univ-mrs.fr

Abstract. We investigate in this work a generalization of the known *CNF* representation which allows an efficient Boolean encoding for n-ary CSPs. We show that the space complexity of the Boolean encoding is identical to the one of the classical CSP representation and introduce a new inference rule whose application until saturation achieves arc-consistency in a linear time complexity for n-ary CSPs expressed in the Boolean encoding. Two enumerative methods for the Boolean encoding are studied: the first one (equivalent to MAC in CSPs) maintains full arc-consistency on each node of the search tree while the second (equivalent to FC in CSPs) performs partial arc-consistency on each node. Both methods are experimented and compared on some instances of the Ramsey problem and randomly generated 3/4-ary CSPs and promising results are obtained.

1 Introduction

Constraint solving is a well known framework in artificial intelligence. Mainly, two approaches are well used: the propositional calculus holding the satisfiability problem (SAT) and the formalism of discrete constraint satisfaction problems (CSPs).

Methods to solve the SAT problem in propositional calculus are former than the CSP ones. They have been widely studied for many years since the Davis and Putnam (DP for abbreviation) procedure [Davis and Putnam, 1960] was introduced, and are still a domain of investigation of a large community of researchers. Several improvements of the DP method have been provided recently [Silva and Sakallah, 1996], [Malik et al., 2001] and [Goldberg and Novikov, 2002]. These new methods are able to solve large scale of SAT instances and are used to tackle real applications. The advantage of SAT methods is their flexibility to solve any constraints given in the CNF [1] form. They are not sensitive to constraint arity[2] as it is often the case for CSP methods. One drawback of the CNF formulation is the loss of the problem structure which is well represented in CSP formalism.

The discrete CSP formalism was introduced by Montanari in 1974 [Montanari, 1974]. The asset of this formalism is its capability to express the problem and explicitly represent its structure as a graph or a hyper-graph. This structure helps to achieve constraint propagation and to provide heuristics which render CSP resolution methods efficient. In

[1] A CNF formula is a conjunction of disjunction of literals

[2] Every SAT solver can deal with any clause length with the same efficiency, except for binary clause

comparison to the propositional calculus, the CSP resolution methods are sensitive to constraint arity. Most of the known methods [Haralick and Elliott, 1980] apply on binary [3] CSPs. This makes a considerable restriction since most of the real problems (see CSPlib[4]) need constraints of unspecified arity for their natural formulation. To circumvent this restriction problem, some works provide methods to deal with more general constraints [Bessière et al., 2002], but this approach is still not well investigated.

Both SAT and CSP formulations are closely linked. Several works on transformations of binary CSP to SAT forms exist [Kleer, 1989]. The transformation in [Kasif, 1990] is improved in [Gent, 2002] and generalized to non-binary CSPs in [Bessière et al., 2003]. Transformations of a SAT form to a CSP form are described in [Bennaceur, 1996,Walsh, 2000]. Unfortunately, most of the transformations from CSP to SAT result in an overhead of space and lose the problem structure. Both factors combined may slow the resolution of the problem. Our aim in this paper is to provide a more general Boolean representation which includes both the CNF and the CSP formulations and which preserves their advantages. That is, a representation which does not increase the size of the problem and which keeps the problem structure. We show particularly how non-binary CSPs are well expressed in this new formulation and efficiently solved. We propose two enumerative methods for the general Boolean formulation which are based on the DP procedure. To enforce constraint propagation we implement a new inference rule which takes advantage of the encoded structure. The application of this rule to the considered problem is achieved with a linear time complexity. We will show that the saturation of this rule on the considered problem is equivalent to enforcing arc consistency on the corresponding CSP. We proved good time complexity to achieve arc consistency for non-binary CSPs. This allows to maintain efficiently full arc consistency at each node of the search tree. This makes the basis of the first enumerative method which is equivalent to the known method MAC [Sabin and Freuder, 1997] in CSP. We also prove that a partial exploitation of this inference rule leads to a second enumerative method which is equivalent to a forward checking (FC) method for non-binary CSPs. Authors of [Bessière et al., 2002] showed that FC is not easy to be generalized for n-ary CSPs, they obtain six versions. In our encoding, FC is obtained naturally by applying the inference rule to the neighborhood of the current variable under instantiation.

The rest of the paper is organized as follows: first we recall some background on both propositional calculus and discrete CSP formalisms. Then we introduce a general normal form which we use to express both SAT and CSP problems. After that, we define a new inference rule which we use to show results on arc consistency. We describe two enumerative algorithms (the FC and MAC versions) based on two different uses of the introduced inference rule. Next, we give some experiments on both randomly generated non-binary CSPs and Ramsey problems to show and compare the behaviors of our resolution methods. Finally, we summarize some previous related works and conclude the work.

[3] A binary CSP contains only two-arity constraints
[4] http://csplib.org

2 Background

A CNF formula f in propositional logic is a conjunction $f = C_1 \wedge C_2 \ldots C_k$ of clauses. Each clause C_i is itself a disjunction of literals. That is, $C_i = l_1 \vee l_2 \ldots l_m$ where each literal l_i is an occurrence of a Boolean variable either in its positive or negative parity. An interpretation I is a mapping which assigns to each Boolean variable the value *true* or *false*. A clause is satisfied if at least one of its literals l_i is given the value *true* in the interpretation I ($I[l_i] = \textit{true}$). The empty clause is unsatisfiable and is denoted by $\square$. The formula f is satisfied by I if all its clauses are satisfied by I, thus I is a *model* of f. A formula is satisfiable if it admits at least one model, otherwise it is unsatisfiable.

On the other hand a CSP is a statement $P = (X, D, C, R)$ where $X = \{X_1, X_2, \ldots, X_n\}$ is a set of n variables, $D = \{D_1, D_2, \ldots, D_n\}$ is a set of finite domains where D_i is the domain of possible values for X_i, $C = \{C_1, C_2, \ldots, C_m\}$ is a set of m constraints, where the constraint C_i is defined on a subset of variables $\{X_{i_1}, X_{i_2}, \ldots, X_{i_{a_i}}\} \subset X$. The arity of the constraint C_i is a_i and $R = \{R_1, R_2, \ldots, R_m\}$ is a set of m relations, where R_i is the relation corresponding to the constraint C_i. R_i contains the permitted combinations of values for the variables involved in the constraint C_i. A binary CSP is a CSP whose constraints are all of arity two (binary constraints). A CSP is non-binary if it involves at least a constraint whose arity is greater than 2 (a n-ary constraint). An *instantiation* I is a mapping which assigns each variables X_i a value of its domain D_i. A constraint C_i is satisfied by the instantiation I if the projection of I on the variables involved in C_i is a tuple of R_i. An instantiation I of a CSP P is consistent (or called a solution of P) if it satisfies all the constraints of P. A CSP P is consistent if it admits at least one solution. Otherwise P is not consistent. Both propositional satisfiability (SAT) and constraint satisfaction problems (CSPs) are two closely related NP-complete problems. For the sequel we denote by n the number of variables of the CSP, by m its number of constraints, by a its maximal constraint arity and by d the size of its largest domain.

Example 1. Let's consider a simplified form of a car-production problem.
The variables and the variable domains are defined as follows :

- Bumper : White
- Sliding roof : Red
- Hub caps : Pink or Red
- Bonnet and Doors : Pink, Red or Black
- Body : White, Pink, Red or Black

And the conceptor's constraints are the following :

- The body must be darker than the bumper,the sliding roof and the Hub caps.
- The doors, the body and the cap must be of the same color.

The constraint network representing the problem is given in Figure 1.

3 An encoding including SAT and CSPs

The idea of translating a CSP into an equivalent SAT form was first introduced by De kleer in [Kleer, 1989]. He proposed the well known *direct encoding* and since that Kasif

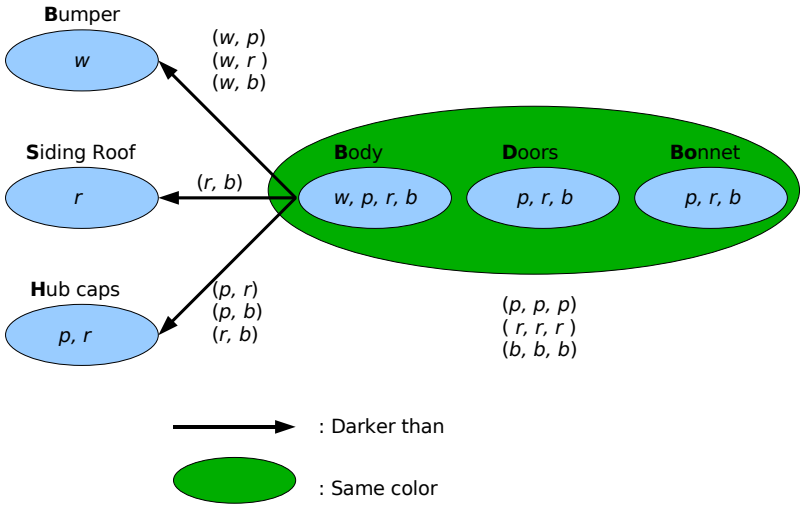


Fig. 1. The constraint network of the car-producing problem.

[Kasif, 1990] proposed the *AC encoding* for binary CSPs. More recently Bessière et al [Bessière et al., 2003] generalized the *AC encoding* to non-binary CSPs. Our approach is different, it consists in providing a general Boolean form including both CNF (SAT) and CSP representations, rather than translating CSPs into SAT forms. We describe in this section, a new Boolean encoding which generalizes the CNF formulation, and show how n-ary CSPs are naturally represented in an optimal way (no overhead in size in comparison to the CSP representation) in this encoding.

3.1 The generalized normal form (GNF)

A generalized clause C is a disjunction of Boolean formulas $f_1 \vee \ldots \vee f_m$ where each f_i is a conjunction of literals, i.e $f_i = l_1 \wedge l_2 \wedge \ldots \wedge l_n$. A formula is in Generalized Normal Form (GNF) if and only if it is a conjunction of generalized clauses. The semantic of generalized clauses is trivial: the generalized clause C is satisfied by an interpretation I if at least one of its conjunctions f_i ($i \in [1, m]$) is given the value *true* in I, otherwise it is falsified by I. A classical clause is a simplified generalized clause where all the conjunctions f_i are reduced to single literals. This proves that GNF is a generalization of CNF. We show in the sequel that each constraint C_i of a given n-ary CSP is represented by a generalized clause. We reach the optimal size representation by using the cardinality formulas $(\pm 1, L)$ which means "exactly one literal among those of the list L have to be assigned the value *true* in each model", to express efficiently that a CSP variable has to be assigned a single value in its domain. We denote by *CGNF* the *GNF* augmented by the cardinality.

3.2 The CGNF encoding for n-ary CSPs

Given an n-ary CSP $P = (X, D, C, R)$, first, we define the set of Boolean variables which we use in the Boolean representation, and two types of clauses: the domain clauses and the constraint clauses necessary to encode the domains and respectively the constraints.

- The set of Boolean variables: as in the existing encodings [Kleer, 1989], [Kasif, 1990] and [Bessière et al., 2003] we associate a Boolean variable Y_v with each possible value v of the domain of each variable Y of the CSP. Thus, Y_v =*true* means that the value v is assigned to the variables Y of the CSP. We need exactly $\sum_{i=1}^{n} |D_i|$ Boolean variables. The number of Boolean variables is bounded by nd.
- The domain clauses: let Y be a CSP variable and $D_Y = \{v_0, v_1, \ldots, v_k\}$ its domain. The cardinality formula $(\pm 1, Y_{v_0} \ldots Y_{v_k})$ forces the variable Y to be assigned to only one value in D_Y. We need n cardinality formulas to encode the n variable domains.
- The constraint clauses: each constraint C_i of the CSP P is represented by a generalized clause C_{C_i} defined as follows: Let R_i be the relation corresponding to the constraint C_i involving the set of variables $\{X_{i_1}, X_{i_2}, \ldots, X_{i_a}\}$. Each tuple t_j = $(v_{j_1}, v_{j_2}, \ldots, v_{j_a})$ of R_i is expressed by the conjunction $f_j = X_{v_{j_1}} \wedge X_{v_{j_2}} \wedge \ldots \wedge X_{v_{j_a}}$. If R_i contains k tuples $t_1, t_2, \ldots, t_k$ then we introduce the generalized clause $C_{C_i} = f_1 \vee f_2 \vee \ldots \vee f_k$ to express the constraint C_i. We need m generalized clauses to express the m constraints of the CSP.

As the domain clauses are encoded in $O(nd)$, the constraint clauses in $O(mad^a)$, then the *CGNF* encoding of a CSP P is in $O(mad^a + nd)$ in the worst case. In fact, it is in $O(mad^a)$ since nd is often negligible. This space complexity is identical to the one of the original CSP encoding. This justifies the optimality in space of the *CGNF* encoding. Authors in [Bessière et al., 2003] gave a spatial complexity in $O(mad^a)$ in the worst case for the k-AC encoding. As far as we understand, this complexity is rounded and does not take into account neither the at-least-one nor the at-most-one clauses. This increases the complexity to $O(mad^a + nd^2)$.

Example 2. Take the car-producing problem of example 1 :

- We have the following propositional variables :
  - $Body \Rightarrow B_w, B_p, B_r, B_b$
  - $Doors \Rightarrow D_p, D_r, D_b$
  - $Bonnets \Rightarrow Bo_p, Bo_r, Bo_b$
  - ...
- We have two types of clauses :
  - The domain clauses :
    $C_{Body} = (\pm 1, B_w\ B_p\ B_r\ B_b)$,
    $C_{Doors} = (\pm 1, D_p\ D_r\ D_b) \ldots$
  - The constraint clauses : for instance, the constraint linking the Body, the Doors and the Bonnet is as follows :
    $C_{BodyDoorsBonnet} = (B_p \wedge D_p \wedge Bo_p) \vee$
    $(B_r \wedge D_r \wedge Bo_r) \vee (B_b \wedge D_b \wedge Bo_b)$

Now we deal with the correctness and completeness of the *CGNF* encoding.

Definition 1. *Let P be a CSP and C its corresponding CGNF encoding. Let I be an interpretation of C. We define the corresponding equivalent instantiation I_p in the CSP P as the instantiation verifying the following condition: for all CSP variable X and each value v of its domain, $X = v$ if and only if $I[X_v]$ =true.*

Theorem 1. *Let P be a CSP, C its CGNF corresponding encoding, I an interpretation of the CGNF encoding and I_p the corresponding equivalent instantiation of I in the CSP P. I is a model of C if and only if I_p is a solution of the CSP P.*

Proof. Let I be a model of C and I_p its corresponding instantiation in P. I satisfies each clause of C, since it is one of its models. We have to prove that each CSP variable is assigned a single value in I_p and I_p satisfies all the constraints of P. A domain clause $C_{D_X} = (\pm 1, X_{v_1}\ X_{v_2} \ldots X_{v_d})$of C means that one and only one Boolean variable among $\{X_{v_1}\ X_{v_2} \ldots X_{v_d}\}$ is true in I. By construction $X = v$ in I_p, the unity is guaranteed since I_p is a mapping. Each constraint clause $C_{C_i} = f_1 \vee f_2 \vee \ldots \vee f_m$, has at least one f_i satisfied in I. This means that the values assigned to the CSP variables involved in the constraint C_i associated to C_{C_i} form a tuple of the relation R_i. Thus, C_i is satisfied by I_p and satisfies all the constraints of P. I_p is a solution of P.
Now, we suppose I_p is a solution of P and shall prove that I is a model of C. Because I_p is a solution, each CSP variable X is assigned to a value v of its domain. $X = v$, then $I[X_v] = true$ and clearly the cardinality formula corresponding to D_X is satisfied by I. Beside, each constraint C_i of P is satisfied. The projection of I_p on the constraint C_i results in a tuple t_j of the relation R_i. This implies that the conjunction f_j of the constraint clause C_{C_i} corresponding to the tuple t_j is true in I , and the clause C_{C_i} is satisfied by I. Thus, I is a model of C.

The *CGNF* encoding allows an optimal representation of CSPs, however it does not capture the property of arc consistency which is the key of almost all the enumerative CSP algorithms. We introduce in the following a simple inference rule which applies on the *CGNF* encoding C of a CSP P and prove that arc consistency is achieved with a linear complexity by application of the rule on C until saturation.

4 A new inference rule for the CGNF encoding

The rule is based on the preserved CSP structure represented by both the domain and the constraint clauses.

4.1 Definition of the inference rule IR

Let P be a CSP, C its *CGNF* encoding, C_{C_i} a generalized constraint clause, C_{D_X} a domain clause, L_C the set of literals appearing in C_{C_i} and L_D the set of literals of C_{D_X}. If $L_C \cap L_D \neq \emptyset$ then we infer each negation $\neg X_v$ of a positive literal[5] appearing in L_D which does not appear in L_C. We have the following rule:
IR: if $L_C \cap L_D \neq \emptyset$, $X_v \in L_D$ and $X_v \notin L_C$ then $C_{D_X} \wedge C_{C_i} \vdash \neg X_v$[6].

[5] The CGNF encoding of a CSP contains only positive literals.
[6] $\vdash$ denotes logical inference.

Example 3. Let's consider the two following clauses of the car-producing problem :
$C_{Body} = (\pm 1, B_w\ B_p\ B_r\ B_b)$ and
$C_{BumpersBody} = (Bu_w \wedge B_p) \vee (Bu_w \wedge B_r) \vee (Bu_w \wedge B_b)$
The application of the rule IR on both clauses infers :$\neg B_w$. That is, $C_{Body} \wedge C_{BumpersBody} \vdash \neg B_w$.

Proposition 1. *The rule IR is sound (correct).*

Proof. Let X_v be a literal appearing in L_D but not in L_C and I a model of $C_{C_i} \wedge C_{D_X}$. C_{C_i} is a disjunction of conjunctions f_i and each conjunction f_i contains one literal of L_D. At least one of the conjunctions f_i, say f_j is satisfied by I since I is a model of C_{C_i}. Thus, there is a literal $X_{v'}$ ($X_{v'} \neq X_v$ since $X_v \notin L_C$) of f_j appearing in L_D, such that $I[X_{v'}] =$*true*. Because of the mutual exclusion of of literal of C_{D_X}, the $X_{v'}$ is the single literal of L_D satisfied by I. Thus, $I[\neg X_v] =$*true* and I is a model of $\neg X_v$.

4.2 The inference rule and arc-consistency

A CSP P is arc consistent iff all its domains are arc consistent. A domain $D_{X_{i_1}}$ is arc consistent iff for each value v_{i_1} of $D_{X_{i_1}}$ and for each k-arity constraint C_j involving the variables $\{X_{i_1}, \ldots, X_{i_k}\}$, there exists a tuple $(v_{i_2}, \ldots, v_{i_k}) \in D_{X_{i_2}} \times \ldots \times D_{X_{i_k}}$ such that $(v_{i_1}, v_{i_2}, \ldots, v_{i_k}) \in R_j$. We use the inference rule IR on the *CGNF* encoding C of a CSP P to achieve arc-consistency. We show that by applying IR on C until saturation (a fixed point is reached) and by propagating each inferred negative literal we maintain arc-consistency on C. Since, this operation is based on unit propagation, it can be done in a linear time complexity.

Proposition 2. *Let P be a CSP and C its CGNF encoding. A value $v \in D_Y$ is removed by enforcing arc-consistencyon P iff the negation $\neg Y_v$ of the corresponding Boolean variable is inferred by application of IR to C.*

Proof. Let v be a value of the domain D_Y which does not verify arc-consistency. There is at least one constraint C_j involving Y such that v does not appear in any of the allowed tuples of the corresponding relation R_j. The Boolean variable Y_v does not appear in the associated constraint clause C_{C_j}, but it appears in the domain clause C_{D_Y} associated to D_Y. By applying the the rule IR on both C_{D_Y} and C_{C_j} we infer $\neg Y_v$.
The proof of the converse can be done in the same way.

Theorem 2. *Let P be a CSP and C its CGNF encoding. The saturation of IR on C and the propagation of all inferred literals is equivalent to enforcing arc-consistency on the original CSP P.*

Proof. Is a consequence of proposition 2.

4.3 Arc-consistency by application of IR

To perform arc consistency on C by applying IR, we need to define some data structures to implement the inference rule. We suppose that the nd Boolean variables of C are encoded by the first integers $[1..nd]$. We define a table OCC_j of size ad for each constraint clause C_{C_j} such that $OCC_j[i]$ gives the number of occurrences of the variable i

in C_{C_j}. There is a total of m such tables corresponding to the m constraint clauses. If $OCC_j[i] = 0$ for some $i \in \{1 \dots nd\}$ and some $j \in \{1 \dots m\}$ then the negation $\neg i$ of the Boolean variable i is inferred by IR. This data structure adds to the space complexity a factor of $O(mad)$. The total space complexity of C is $O(mad^a + nd + mad)$, but the factors nd and mad are always lower than the mad^a factor, and the space complexity of C remains $O(mad^a)$.

The principle of the arc-consistency method consists first in reading the m tables OCC_j to detect the variables $i \in [1 \dots nd]$ having a number of occurrences equal to zero ($OCC_j[i] = 0$). This is achieved by the steps 3 to 9 of algorithm 1 in $O(mad)$, since there are m tables of size ad. After that we apply unit propagation on the detected variables, and propagate the effect until saturation (i.e no new variable i having $OCC_j[i] = 0$ is detected). The procedure of arc-consistency is sketched in Algorithm 1. This procedure calls the procedure *Propagate* described in algorithm 2.

Algorithm 1 Arc Consistency

Procedure Arc_consistency
Input: A CGNF instance C
1: **var** L : list of literals
2: $L = \emptyset$
3: **for** each constraint clauses C_{C_j} **do**
4: **for** each literal i of OCC_j **do**
5: **if** $OCC_j[i]$ = 0 **then**
6: add i in L
7: **end if**
8: **end for**
9: **end for**
10: **while** $L \neq \emptyset$ and $\square \notin C$ **do**
11: extract l from L
12: propagate (C, l, L)
13: **end while**

The complexity of the arc consistency procedure is mainly given by the propagation of the effect of literals of the list L (lines 10 to 13 of algorithm 1). It is easy to see that in the worst case there will be nd calls to the procedure $Propagate$. All the propagations due to the previous calls are performed in $O(mad^a)$ in the worst case. Indeed, there are at most d^a conjunctions of a literals for each constraint clause of C. The total number of conjunctions treated in line 4 of algorithm 2 can not exceed md^a since each considered conjunction f in line 3 is suppressed in line 4 (f is interpreted to false). As there is a literals by conjunction f, the total propagation is done in $O(mad^a)$. Thus, the complexity of the arc consistency procedure is $O(mad^a + mad)$. But the factor (mad) is necessarily smaller than mad^a and the total complexity is reduced to $O(mad^a)$. It is linear *w.r.t* the size of C.

Algorithm 2 Propagate

Procedure Arc_consistency
Input: A CGNF instance C, literal i, List L

```
1: if i is not yet assigned then
2:    assign i the value false
3:    for each unassigned conjunction f of C containing i do
4:       assign f the value false
5:       for each literal j of f such that i ≠ j do
6:          withdraw 1 to OCC_k[j] {k is the label of the constraint clause containing f}
7:          if OCC_k[j] = 0 then
8:             add j to L
9:          end if
10:      end for
11:   end for
12: end if
```

5 Two enumerative methods for the CGNF encoding

We study in the following two enumerative methods: the first one (MAC) maintains full arc consistency during the search while the second (FC) maintains partial arc consistency as does the classical Forward Checking method in CSPs [Haralick and Elliott, 1980]. Both methods perform Boolean enumeration and simplification. They are based on an adaptation of the DP procedure to the *CGNF* encoding.

5.1 The MAC method

MAC starts by a first call to the *Arc-consistency* algorithm (Figure 1) to verify arc consistency at the root of the search tree. It then calls the procedure *Propagate* described in Figure 2 at each node of the search tree to maintain arc consistency during the search. That is, the mono-literals of each node are inferred and their effects are propagated. The code of *MAC* is sketched in Figure 3.

5.2 The FC method

It is easy to obtain from MAC a *Forward Checking (FC)* algorithm version for the CGNF encoding. The principle is the same as in *MAC* except that instead of enforcing full arc-consistency at each node of the search tree, (FC) does it only on the near neighbors of the current variable under assignment. This is done by restricting the propagation effect of the current literal assignment to only the literals of the clauses in which it appears. It is important to see that finding the *FC* version in our encoding is trivial, whereas it is not the case for n-ary CSP where six versions of *FC* are provided [Bessière et al., 2002].

5.3 Heuristics for literal choice

Because the CGNF encodings keeps the CSP structure, We can find easily equivalent heuristics to all the well known CSP variable/value heuristics. For instance, the minimal

Algorithm 3 MAC

Procedure MAC
Input: A CGNF instance C
Output: The satisfiability of the CGNF C

```
1: Arc consistency(C)
2: if □ ∈ C then
3:    return unsatisfiable
4: else
5:    choose a literal l ∈ C
6:    if Satisfiable(C, l, true) then
7:       return satisfiable
8:    else if Satisfiable(C, l, false) then
9:       return satisfiable
10:   else
11:      return unsatisfiable
12:   end if
13: end if
```

domain heuristic (MD) which consists in choosing during the search the CSP variable whose domain is the smallest is equivalent to select in the CGNF encoding, a literal appearing in the shortest domain clause. This heuristic is implemented in both MAC and FC methods.

6 Experimentations

We experiment both *MAC* and *FC* methods and compare their performances on Ramsey problem, and on 3/4-ary randomly generated CSPs encoded in CGNF. The programs are written in C, compiled and run on a Windows operating system with a Pentium IV 2.8GHz processor and 1GB of RAM.

6.1 Ramsey problem

Ramsey problem (see CSPlib) consists in coloring the edges of a complete graph having n vertices with k colors, such that no monochromatic triangle appears. Tables 2 shows the results of *MAC* and *FC* on some instances of Ramsey problem where $k = 3$ and n varies from 5 to 14. The first column defines the problem: Rn_k denotes a Ramsey instance with n vertices and k colors, the second and third columns show the number of nodes and the performances in CPU times of *FC* respectively *MAC* augmented by the heuristic (MD) described previously.

We can see that *FC* is better in time than *MAC* for small size instances ($n \leq 6$) but visits more nodes. However, *MAC* is better than *FC* in time and number of visited nodes when the problem size increases ($n \geq 7$). *MAC* outperforms *FC* in most of the cases for this problem.

Algorithm 4 Satisfiable

Function Satisfiable
Input: A CGNF instance C, variable l, Boolean val
Output: Boolean {TRUE or FALSE}

var L : list of literals
$L = \emptyset$
if $val = true$ **then**
  assign l the value *true*
  Add each literal $i \neq l$ of the domain clause containing l in L
  while $L \neq \emptyset$ and $\square \notin C$ **do**
    extract i from L
    propagate (C, i, L)
  end while
else
  repeat
    $Propagate(C, l, L)$
  until $L \neq \emptyset$ and $\square \notin C$
end if
if $C = \emptyset$ **then**
  return TRUE
else if $\square \in C$ **then**
  return FALSE
else
  Choose a literal p of C
  if Satisfiable$(C, p, true)$ **then**
    return TRUE
  else if Satisfiable$(C, p, false)$ **then**
    return TRUE
  else
    return FALSE
  end if
end if

6.2 Random problems

The second class of problems is randomly generated CSPs. We carried experiment on 3/4-ary random CSPs where both the number of CSP variables and the number of values is 10. Our generator is an adaptation to the CGNF encoding of the Bessière et al CSP generator. It uses five parameters: n the number of variables, d the size of the domains, a the constraint arity, *dens* the constraint density which is the ratio of the number of constraint to the maximum number of possible constraints, t the constraint tightness which is the proportion of the forbidden tuples of each constraints. The random output CSP instances are given in the CGNF encoding.

Figure 3 (respectively, Figure 4) shows the average curves representing the CPU time of both *MAC* and *FC*, both augmented by the *MD* heuristic, with respect to a variation of the tightness on 3-ary (respectively, 4-ary) CSPs. The two curves on the right of Figure 3 (respectively, Figure 4) are those of *MAC* and *FC* corresponding to weak densities: dens=0,20 for to the curves A (respectively, dens=0.096 for the curves D), the two ones

Problem	FC + MD		MAC + MD	
	Nodes	Times	Nodes	Times
R5_3	*12*	*0 s 48 µs*	*12*	*0 s 85 µs*
R6_3	*27*	*0 s 231 µs*	*21*	*0 s 297 µs*
R11_3	*237*	*0 s 5039 µs*	*103*	*0 s 4422 µs*
R12_3	*538*	*0 s 21558 µs*	*130*	*0 s 11564 µs*
R13_3	*1210*	*0 s 28623 µs*	*164*	*0 s 9698 µs*
R14_3	*216491*	*9 s 872612 µs*	*6735*	*1 s 752239 µs*

Fig. 2. Results of *MAC* and *FC* on the Ramsey problem

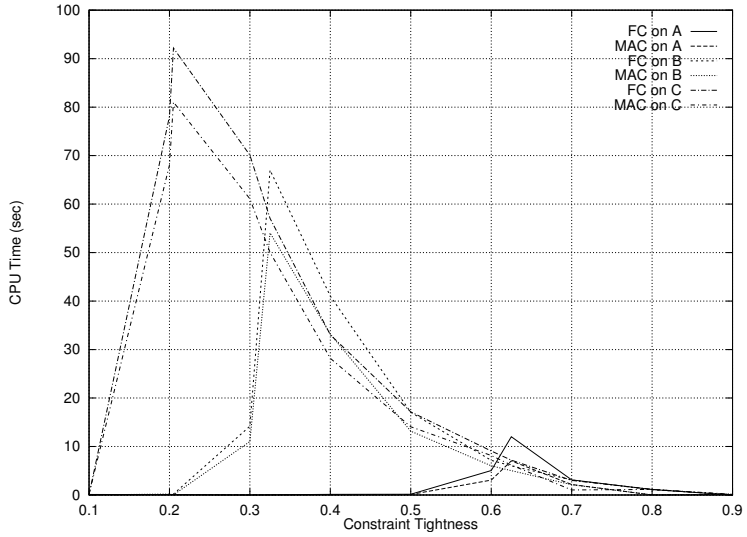


Fig. 3. Results of MAC and FC on 3-ary CSPs having the densities: A: dens=0,20, B: dens=0,50 and C: dens=0,83.

in the middle are those corresponding to average densities: dens=0,50 for the curves B (respectively, dens=0.5 for the curves E) and the two ones on the left are those corresponding to high densities: dens=0,83 for the curves C (respectively, dens=0.96 for the curves F). We can see that the peak of difficulty of the hard region of each problem class matches with a critical value of the tightness and the hardness increases as the density increases. The smaller the density is, the greater the corresponding critical tightness is. We can also see that *MAC* beats definitely *FC* on the six classes of problems. These results seem to compare with those obtained by the generalized FC [Bessière et al., 2002] on 3/4-ary CSPs having equivalent parameters (n, d, $dens$, t).

7 Related works and discussion

We summarize in the following some well known translations of CSPs to SAT and compare their properties.

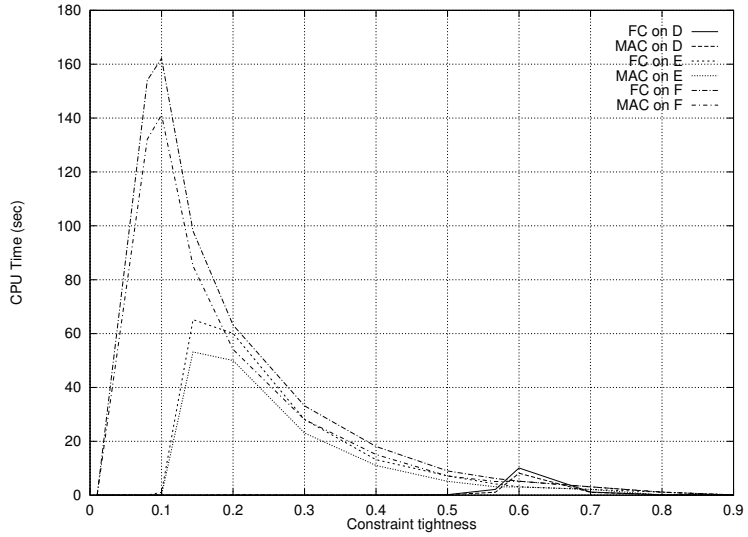


Fig. 4. Results of MAC and FC on 4-ary CSPs having the densities: dens=0,096, dens=0,50 and dens=0,95.

- *The direct encoding* [Kleer, 1989] is the most used translation of CSPs into SAT. The basic idea is to associate a Boolean variable X_v for each possible instantiation $X = v$ of the CSP variable X. That is, if I is an interpretation, then $I[X_v] = true$ means that the value v is assigned to the variable X. These Boolean variables will appear in three types of clauses: the At-least-one clauses, the At-most-one clauses, and the Conflict clauses. Given a CSP P, its direct encoding has a worst case space complexity of $O(nd + \frac{nd(d-1)}{2} + mad^a)$ and does not capture arc consistency.
- *The AC encoding* [Kasif, 1990,Gent, 2002] is defined for binary CSPs. Its particularity is the representation of the arc consistency property in the encoding. It allows to a SAT procedure to achieve arc consistency by applying only unit propagation. This encoding differs from the direct encoding by only the conflict clauses which are replaced by the support clauses. Its space complexity in the worst case is the same as the direct encoding.
- *The k-AC encoding* [Bessière et al., 2003] is a generalization of the AC encoding to non-binary CSP. It can represent supports on a subset S of variables of any size involved in a constraint C, for the instantiation of another variable subset T of any size of the same constraint C, rather than only supports of a single variable on another single variable as in the AC encoding. Its space complexity is identical to the ones of both previous encodings, and slightly greater than the one of the CGNF encoding which is in $O(nd + mad^a)$. The k-AC encoding captures a larger family of consistency than both the AC and the CGNF encodings. That is, it encodes the *relational k-arc-consistency* [Bessière et al., 2003]., but the great number of clauses needed to maintain the encoding in CNF, and the extra variables introduced to express the supports can in some cases slow a SAT solver.
- *Comparison and discussion:*

Our approach is different from the previous works [Kleer, 1989], [Kasif, 1990] and [Bessière et al., 2003], since it is not a translation from CSP to SAT. It consists in a generalization of the CNF representation which allows a natural and optimal encoding for n-ary CSPs keeping the CSP structure. The purpose of this paper is first to provide a more general framework including both SAT and CSP and which captures their advantages, then provide promising methods to solve problems expressed in this framework. We are not interested for the moment in code optimization and heuristics to compete other optimized methods. We just provide a first implementation, whose results look to compare well with those of the nFCs for n-ary CSPs given in [Bessière et al., 2002]. But, it seems that most of the background (like *non-chronological back-tracking*, or *clauses recording*) added to DP procedure to produce sophisticated SAT solvers like *Chaff* [Malik et al., 2001] can be adapted for our methods. Such optimizations would increase the performance of our methods, this question will be investigated in a future work.

8 Conclusion

We studied a generalization of the known *CNF* representation which allows a compact Boolean encoding for n-ary CSPs. We showed that the size of a CSP in this encoding is identical to the one of its original representation. We implemented a new inference rule whose application until saturation achieves arc-consistency for n-ary CSPs expressed in the Boolean encoding with a linear time complexity. Two enumerative methods are proposed: the first one (MAC) maintains full arc-consistency on each node of the search tree while the second (FC) performs partial arc-consistency. Both methods are well known in CSPs and are found easily in our Boolean encoding. These methods are experimented on some instances of Ramsey problem and randomly generated 3/4-ary CSPs and the obtained results showed that maintaining full arc-consistency in the Boolean encoding is the best idea. These results are promising, but code optimizations are necessary to compete with sophisticated SAT solvers. As a future work, we are looking to extend the inference rule to achieve path consistency in the Boolean encoding. An other interesting point is to look for polynomial restrictions of the Boolean encoding. On the other hand, detecting and breaking symmetries in the Boolean encoding may increase the performances of the defined enumerative methods.

References

[Bennaceur, 1996] Bennaceur, H. (1996). The satisfiability problem regarded as constraint satisfaction problem. *Proceedings of the European Conference on Artificial Intelligence (ECAI'96)*, pages 155–159.

[Bessière et al., 2003] Bessière, C., Hebrard, E., and Walsh, T. (2003). Local consistency in sat. *Selected revised papers from SAT'03, LNCS 2919, Springer*, pages 299–314.

[Bessière et al., 2002] Bessière, C., Meseguer, P., Freuder, E. C., and J.Larrosa (2002). On forward checking for non-binary constraint satisfaction. *Journal of Artificial Intelligence*, 141:205–224.

[Davis and Putnam, 1960] Davis, M. and Putnam, H. (1960). A computing procedure for quantification theory. *Journal of ACM 7*, pages 201–215.

[Gent, 2002] Gent, I. P. (2002). Arc consistency in sat. *Proceedings of the European Conference on Artificial Intelligence (ECAI'02)*, pages 121–125.

[Goldberg and Novikov, 2002] Goldberg, E. and Novikov, Y. (2002). Berkmin: A fast and robust sat solver. *Proceedings of the 2002 Design Automation and Test in Europe*, pages 142–149.

[Haralick and Elliott, 1980] Haralick, R. and Elliott, G. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Journal of Artificial Intelligence*, 14:263–313.

[Kasif, 1990] Kasif, S. (1990). On the parallel complexity of discrete relaxation in constraint satisafaction networks. *Journal of Artificial Intelligence*, 45:275–286.

[Kleer, 1989] Kleer, J. D. (1989). A comparison of atms and csp techniques. *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'89)*, pages 290–296.

[Malik et al., 2001] Malik, S., Zhao, Y., Madigan, C. F., Zhang, L., and Moskewicz, M. W. (2001). Chaff: Engineering an efficient sat solver. *Proceedings of the 38th conference on Design automation (IEEE 2001)*, pages 530–535.

[Montanari, 1974] Montanari, U. (1974). Networks of constraints : Fundamental properties and application to picture processing. *Journal Inform. Sci.*, 9(2):95–132.

[Sabin and Freuder, 1997] Sabin, D. and Freuder, E. (1997). Understanding and improving the mac algorithm. *Proceedings of International Conference on Principles and Practice of Constraint Programming (CP'97)*, pages 167–181.

[Silva and Sakallah, 1996] Silva, J. and Sakallah, K. (1996). Grasp – a new search algorithm for satisfiability. *Proceedings of International Conference on Computer-Aided Design (IEEE 1996)*, pages 220–227.

[Walsh, 2000] Walsh, T. (2000). Sat v csp. *Proceedings of International Conference on Principles and Practice of Constraint Programming (CP'00)*, pages 441–456.

[blank page]

Interval Constraint Solving Using Propositional SAT Solving Techniques

Martin Fränzle[1], Christian Herde[1], Stefan Ratschan[2], Tobias Schubert[3], and Tino Teige[1⋆]

[1] Dept. of CS, Carl von Ossietzky Universität Oldenburg, Germany
{fraenzle|herde|teige}@informatik.uni-oldenburg.de

[2] Inst. of CS, Academy of Sciences of the Czech Republic, Prague
stefan.ratschan@cs.cas.cz

[3] FAW, Albert-Ludwigs-Universität Freiburg, Germany
schubert@informatik.uni-freiburg.de

Abstract. In order to facilitate automated reasoning about large Boolean combinations of non-linear arithmetic constraints involving transcendental functions, we extend the paradigm of lazy theorem proving to interval-based arithmetic constraint solving. Algorithmically, our approach deviates substantially from "classical" lazy theorem proving approaches in that it directly controls arithmetic constraint propagation from the SAT solver rather than completely delegating arithmetic decisions to a subordinate solver. From the constraint solving perspective, it extends interval-based constraint solving with all the algorithmic enhancements that were instrumental to the enormous performance gains recently achieved in propositional SAT solving, like conflict-driven learning combined with non-chronological backtracking.

1 Introduction

Within many application domains, e.g. the analysis of programs involving arithmetic operations or the analysis of hybrid discrete-continuous systems, one faces the problem of solving large Boolean combinations of non-linear arithmetic constraints over the reals, where solving means to find a satisfying valuation or to prove nonexistence thereof. This gives rise to a plethora of problems, in particular (a) how to effectively and efficiently solve conjunctive combinations of constraints in the in general undecidable domain of non-linear constraints involving transcendental functions, and (b) how to efficiently maneuver the large search spaces arising from the rich Boolean structure of the overall formula.

While promising solutions for these two individual sub-problems exist, it seems that their combination has hardly been attacked. Arithmetic constraint solving based on interval constraint propagation [5, 4, 3], on the one hand, has

⋆ This work has been partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, www.avacs.org).

proven to be an efficient means for solving robust combinations of otherwise undecidable arithmetic constraints [22]. Here, robustness means that the constraints maintain their truth value under small perturbations of the constants in the constraints. Modern SAT solvers, on the other hand, can efficiently find satisfying valuations of very large propositional formulae (e.g., [20, 17]), as well as —using the lazy theorem proving paradigm— of complex propositional combinations of atoms from various decidable theories (e.g., [10, 8, 1, 9]).

Within this paper, we describe a tight integration of the paradigm of lazy theorem proving with interval-based arithmetic constraint solving, thus providing a lazy theorem proving approach that reasons over the undecidable arithmetic domain of Boolean combinations of non-linear constraints involving transcendental functions. Within our approach, a DPLL-based propositional satisfiability solver traverses the proof tree originating from the Boolean structure of the constraint formula, as is characteristic for the lazy theorem proving approach. Yet, in contrast to traditional lazy theorem proving approaches ranging over some *decidable* theory T, we do not pass a corresponding conjunctive constraint system over the respective theory T to a subordinate decision procedure serving as an oracle for consistency of the constraint set. Instead, we exploit the algorithmic similarities between DPLL-based propositional SAT solving and constraint solving based on constraint propagation for a much tighter integration, where the DPLL solver has full introspection in and direct control over constraint propagation within the theory T rather than completely delegating theory-related decisions to a subordinate solver. This tight integration has a number of advantages. First, by sharing the common core of the search algorithms between the propositional and the theory-related, interval-constraint-propagation-based part of the solver, we are able to transfer algorithmic enhancements from one domain to the other: in particular, we thus equip interval-based constraint solving with all the algorithmic enhancements that were instrumental to the enormous performance gains recently achieved in propositional SAT solving, like non-chronological backtracking and conflict-driven learning. Second, the introspection into the constraint propagation process allows fine-granular control over the necessarily incomplete arithmetic deduction process, thus enabling a stringent extension of lazy theorem proving to an undecidable theory. Finally, due to the availability of learning, we are able to implement an almost lossless restart mechanisms within an interval-based arithmetic constraint propagation framework, thus being able to substantially accelerate incremental proof searches, where the individual constraint propagations are depth-constrained, yet incrementally less depth-constrained proof searches are iterated until a solution is found (or absence of such is proved). Such iteration is essential to quasi-completeness, i.e. termination on all constraint formulas that are robust in the sense that their truth value does not change under some small variation of constants [22].

Structure of the paper. We start our exposition in Section 2 with a description of the syntactic structure and the semantics of the arithmetic satisfiability problems we are going to address. Section 3 provides a brief introduction to the technologies that our development builds on. Thereafter, we provide a detailed

explanation of our new algorithm (Section 4) and benchmark results (Section 5). We conclude with an overview over ongoing work and planned extensions.

2 Logics

Aiming at automated analysis of programs operating over the reals, e.g. bounded model checking of hybrid systems, our constraint solver addresses satisfiability of non-linear arithmetic constraints over real-valued variables plus Boolean variables for encoding the control flow. The user thus may input constraint formulae built from quantifier-free constraints over the reals and from propositional variables using arbitrary Boolean connectives. The atomic real-valued constraints are relations between potentially non-linear terms involving transcendental functions, like $\sin(x + \omega t) + ye^{-t} \leq z + 5$. By the front-end of our constraint solver, these constraint formulae are rewritten to quantifier-free constraints in conjunctive normal form, with atomic propositions ranging over propositional variables and arithmetic constraints confined to a form resembling three-adress code (cf. the *primitive constraints* of interval constraint propagation, see Section 3). Thus, the *internal* syntax of constraint formulae is as follows:

$$
\begin{aligned}
\mathit{formula} &::= \{\mathit{clause} \wedge\}^* \mathit{clause} \\
\mathit{clause} &::= (\{\mathit{bound} \vee\}^* \mathit{bound}) \mid (\mathit{bound} \vee \mathit{equation}) \\
\mathit{bound} &::= \mathit{variable} \geq \mathit{rational\_const} \mid \mathit{variable} > \mathit{rational\_const} \\
&\quad \mid \mathit{variable} < \mathit{rational\_const} \mid \mathit{variable} \leq \mathit{rational\_const} \\
\mathit{variable} &::= \mathit{real\_variable} \mid \mathit{boolean\_variable} \\
\mathit{equation} &::= \mathit{triplet} \mid \mathit{pair} \\
\mathit{triplet} &::= \mathit{real\_variable} = \mathit{real\_variable}\ \mathit{bop}\ \mathit{real\_variable} \\
\mathit{pair} &::= \mathit{real\_variable} = \mathit{uop}\ \mathit{real\_variable} \\
\mathit{bop} &::= + \mid - \mid * \mid / \mid \ldots \\
\mathit{uop} &::= - \mid \sin \mid \exp \mid \ldots
\end{aligned}
$$

Such constraint formulae are interpreted over valuations $\sigma \in (BV \xrightarrow{\text{total}} \mathbb{B}) \times (RV \xrightarrow{\text{total}} \mathbb{R})$, where BV is the set of Boolean and RV the set of real-valued variables. $\mathbb{B}$ is identified with the subset $\{0, 1\}$ of $\mathbb{R}$ such that any rational-valued bound on a Boolean variable v corresponds to a literal v or $\neg v$. The definition of satisfaction is standard: a constraint formula ϕ is satisfied by a valuation iff all its clauses are satisfied, i.e. iff at least one atom is satisfied in any clause, where the term *atom* refers to both bounds and equations. Satisfaction of atoms is wrt. the standard interpretation of the arithmetic operators and ordering relations over the reals. We assume all arithmetic operators to be total and therefore extend their codomain (as well as, for compositionality, their domain) with a special value $\mho \notin \mathbb{R}$. It is understood that $\mho$ does not satisfy any inequation, i.e. $\mho \not\sim c$ for any constant c and any relation $\sim$.

Instead of real-valued valuations of variables, our constraint solving algorithm manipulates interval-valued valuations $\rho \in (BV \xrightarrow{\text{total}} \mathbb{I}_{\mathbb{B}}) \times (RV \xrightarrow{\text{total}} \mathbb{I}_{\mathbb{R}})$, where

$\mathbb{I}_{\mathbb{B}} = 2^{\mathbb{B}} \setminus \emptyset$ and $\mathbb{I}_{\mathbb{R}}$ is the set of non-empty convex subsets of $\mathbb{R}$.[4] Slightly abusing notation, we write $\rho(l)$ for $\rho_{\mathbb{I}_{\mathbb{B}}}(l)$ when $\rho = (\rho_{\mathbb{I}_{\mathbb{B}}}, \rho_{\mathbb{I}_{\mathbb{R}}})$ and $l \in BV$, and similarly $\rho(x)$ for $\rho_{\mathbb{I}_{\mathbb{R}}}(x)$ when $x \in RV$. If both σ and η are interval assignments then σ is called a *refinement* of η iff $\sigma(v) \subseteq \eta(v)$ for each $v \in BV \cup RV$. We call an interval valuation ρ *weakly satisfying* for a constraint formula ϕ iff each clause of ϕ contains at least one weakly satisfied atom. Weak satisfaction of atoms is defined as follows:

$$\begin{array}{lll} \rho \models_{\mathrm{w}} x \sim c & \text{iff } \rho(x) \subseteq \{u \mid u \in \mathbb{R}, u \sim c\} & \text{for } x \in RV \cup BV, c \in \mathbb{Q} \\ \rho \models_{\mathrm{w}} x = y \circ z & \text{iff } \rho(x) \supseteq \rho(y) \hat{\circ} \rho(z) & \text{for } x, y, z \in RV, \circ \in bop \\ \rho \models_{\mathrm{w}} x = \circ y & \text{iff } \rho(x) \supseteq \hat{\circ} \rho(y) & \text{for } x, y \in RV, \circ \in uop \end{array}$$

where $\hat{\circ}$ is a conservative interval extension of operation $\circ$, i.e. satisfies $i_1 \hat{\circ} i_2 \supseteq \{x \circ y \mid x \in i_1, y \in i_2\}$ for all intervals i_1 and i_2 [19]. Note that equality is interpreted as set inclusion rather than set equality in the interval interpretation. This is motivated by the fact that under appropriate side-conditions, inclusion is sufficient for deciding real-valued (un-)satisfiability, as expressed in Lemma 1 below. In order to formalize these side conditions, we call ρ *strongly satisfying* for ϕ, denoted $\rho \models_{\mathrm{s}} \phi$, iff there is a finite family $a_1, \ldots, a_n$ of atoms such that the following three conditions hold:

1. Each clause in ϕ contains at least one atom a that *matches* some atom a_i in $\{a_1, \ldots, a_n\}$ in the following sense: a_i is a reshuffling of a obtained by using partial inverses of the operation entailed in a, e.g. a being $x = y + z$ and a_i being $z = x - y$. Note that such a reshuffling has no impact on real-valued satisfiability, yet yields additional freedom for interval satisfaction, as it allows to reorder the directions of the set inclusions.
2. The atoms a_1 to a_n are weakly satisfied by ρ.
3. If a_i is a triplet $x = y \circ z$ or a pair $x = \circ y$ then x is interpreted by a point interval (i.e., $|\rho(x)| = 1$), or x does neither occur in any a_j with $j > i$ nor on the right-hand side of a_i (i.e., $x \neq y$ and $x \neq z$).

Due to the ordering condition on equality constraints that are satisfied by non-point intervals, we obtain the following tight correspondence between strong interval satisfiability and real-valued satisfiability:

Lemma 1. *Assume that the interval extensions of the operations are tight on point-interval arguments, i.e. that* $\{a\} \hat{\circ} \{b\} = \{a \circ b\}$ *for each* $a, b \in \mathbb{R}$ *and each binary operation* $\circ$*, and analogously for unary operations. Then*

1. *If* $\sigma \models \phi$ *for some real-valued valuation* σ *then* $\rho \models_{\mathrm{s}} \phi$ *for some interval valuation* ρ *with* $\forall v \in BV \cup RV . \sigma(v) \in \rho(v)$.
2. *If* $\rho \models_{\mathrm{s}} \phi$ *then there exists a real-valued valuation* σ *such that* $\sigma \models \phi$ *and* $\forall v \in BV \cup RV . \sigma(v) \in \rho(v)$.

[4] Note that this definition covers the open, half-open, and closed non-empty intervals over $\mathbb{R}$, including unbounded intervals.

Proof. 1. Take $\rho(v) = \{\sigma(v)\}$ for each $v \in BV \cup RV$. Due to tightness of the interval extensions on point intervals, $\sigma \models \phi$ implies $\rho \models_{\mathrm{w}} \phi$, which in turn implies $\rho \models_{\mathrm{s}} \phi$ due to all intervals assigned by ρ being point intervals such that strong and weak satisfaction coincide.

2. If $\rho \models_{\mathrm{s}} \phi$ then we can recursively define a real-valued valuation σ exploiting the structure of the family of witnesses a_i as follows:
   (a) For each $v \in BV$ and for each $v \in RV$ not occurring on the left hand side of any equation in $\{a_1, \ldots, a_n\}$, select $\sigma(v) \in \rho(v)$ arbitrarily;
   (b) For $i = n$ downto 1, process the constraints a_n to a_1 in reverse sequence as follows:
      i. if a_i is a triplet $v = x \circ y$ then take $\sigma(v) = \sigma(x) \circ \sigma(y)$,
      ii. if a_i is a pair $v = \circ x$ then take $\sigma(v) = \circ\sigma(x)$.

   Note that solutions to the equation system in (b) do exist because the hierarchical order of variable dependencies in a_1 to a_n enforces that each $\sigma(v)$ either is subject to at most one defining equation or is picked from a point interval $\rho(v) \supseteq \rho(x)\hat{\circ}\rho(y)$ or $\rho(v) \supseteq \hat{\circ}\rho(y)$, respectively. Furthermore, $\sigma(v) \neq \mho$ as $\mho \notin \rho(v) \supseteq \rho(x)\hat{\circ}\rho(y) \ni \sigma(x) \circ \sigma(y)$ and $\mho \notin \rho(v) \supseteq \hat{\circ}\rho(y) \ni \circ\sigma(y)$, respectively. It is straightforward to check that $\sigma \models \phi$. □

When solving satisfiability problems of formulae with Davis-Putnam-like procedures, we will build interval valuations incrementally by successively contracting intervals.

We say that an interval valuation ρ is *weakly (or strongly) consistent with a formula (or clause or atom)* ϕ iff there exists a refinement η of ρ that weakly (strongly, resp.) satisfies ϕ. Otherwise, we call ρ *weakly (strongly, resp.) inconsistent with* ϕ. Note that deciding weak consistency of an interval valuation with a single bound is straightforward, as is deciding weak satisfaction of an arbitrary atom. If ρ is neither weakly satisfying for ϕ nor weakly inconsistent with ϕ then we call ϕ *inconclusive on* ρ.

3 Algorithmic Basis

Our constraint solving approach builds upon the well-known techniques of interval constraint propagation, propositional SAT solving by the DPLL procedure plus its more recent algorithmic enhancements, and lazy theorem proving.

Interval constraint propagation (ICP) is one of the sub-topics of the area of constraint programming where constraint propagation techniques are studied in various, and often discrete, domains. For the domain of the real numbers, given a constraint ϕ and a floating-point box B, so-called *contractors* compute another floating-point box $C(\phi, B)$ such that $C(\phi, B) \subseteq B$ and such that $C(\phi, B)$ contains all solutions of ϕ in B (cf. the notion of *narrowing operator* [2]).

There are several methods for implementing such contractors. The most basic method [5, 4] decomposes all atomic constraints (i.e., constraints of the form $t \geq 0$ or $t = 0$, where t is a term) into conjunctions of so-called primitive constraints (i.e., constraints such as $x + y = z$, $xy = z$, $z \in [\underline{a}, \overline{a}]$, or $z \geq 0$)

by introducing additional auxiliary variables (e.g., decomposing $x + \sin y \geq 0$ to $\sin y = v_1 \wedge x + v_1 = v_2 \wedge v_2 \geq 0$). Then it applies a contractor for these primitive constraints until a fixpoint is reached.

We illustrate contractors for primitive constraints using the example of a primitive constraint $x + y = z$ with the intervals $[1, 4]$, $[2, 3]$, and $[0, 5]$ for x, y, and z, respectively: We can solve the primitive constraint for each of the free variables, arriving at $x = z - y$, $y = z - x$, and $z = x + y$. Each of these forms allows us to contract the interval associated with the variable on the left-hand side of the equation: Using the first solved form we subtract the interval $[2, 3]$ for y from the interval $[0, 5]$ for z, concluding that x can only be in $[-3, 3]$. Intersecting this interval with the original interval $[1, 4]$, we know that x can only be in $[1, 3]$. Proceeding in a similar way for the solved form $y = z - x$ does not change any interval, and finally, using the solved form $z = x + y$, we can conclude that z can only be in $[3, 5]$.

Contractors for other primitive constraints can be based on interval arithmetic in a similar way. There is extensive literature [21, 13] providing precise formulae for interval arithmetic for addition, subtraction, multiplication, division, and the most common transcendental functions. The floating point results are always rounded outwards, such that the result remains correct also under rounding errors. There are several variants, alternatives and improvements of the basic approach described above, e.g. [14, 2, 18, 12, 16].

Propositional SAT solving. The *Propositional Satisfiability Problem* (SAT) is a well-known NP-complete problem, with extensive applications in various fields of computer science and engineering. In recent years a lot of developments in creating powerful SAT algorithms have been made, leading to state-of-the-art approaches like BerkMin [11], Mira [17], and zChaff [20]. All of them are enhanced variants of the classical backtrack search DPLL procedure [6, 7]. In contrast to local search strategies only such complete algorithms are able to prove the unsatisfiability of a problem instance, which is often the final objective in many applications, e.g. circuit verification and automated theorem proving.

Given a Boolean formula ϕ in *Conjunctive Normal Form* (CNF) and a partial valuation ρ, which is empty at the beginning of the search process, a backtrack search algorithm incrementally extends ρ until either $\rho \models \phi$ holds or ρ turns out to be inconsistent for ϕ. In the latter case another extension of ρ is tried through backtracking.

Extensions are constructed by performing *decision steps*, which entail selecting an unassigned variable and assigning a value to it. Since the days of the original DPLL procedure many variable selection strategies have been introduced, among them the *Variable State Independent Decaying Sum* (VSIDS) heuristic from zChaff [20]. The main idea of VSIDS is to prefer those variables that often occur in recently deduced *conflict clauses*.

Each decision step is followed by the *deduction phase*, involving the search for *unit clauses*, i.e. clauses that have only one unassigned literal left while all other literals are assigned incorrectly in the actual valuation ρ. Obviously, unit clauses require certain assignments in order to preserve their satisfiability, where

the execution of the implied assignments itself might force further assignments. In the context of SAT solving such necessary assignments are also referred to as *implications*. To perform the deduction phase in an efficient manner zChaff introduced a lazy clause evaluation technique based on *Watched Literals* (WL): for each clause two literals are selected in such a way, that they either are both unassigned or at least one of them is satisfying the clause. So, if at some point during the search one of the WLs is getting assigned incorrectly, a new WL for the corresponding clause has to be found. If such a literal does not exist and the second WL is still unassigned, the clause is forcing an implication. As a consequence of this method there is no need to check all clauses after making a decision step, but only those ones, where a WL is getting assigned incorrectly.

However, deduction may also yield a *conflicting clause* which has all its literals assigned false, indicating the need for backtracking. In order to avoid repeating the same unsatisfying valuation ρ multiple times, modern SAT algorithms incorporate *conflict-driven learning* to derive a sufficiently general reason (a combination of variable assignments) for the actual conflict. Based on that ideally minimal number of assignments that triggered the particular conflict, a *conflict clause* is generated and added to the clause set to guide the subsequent search. Additionally, the conflict clause is used to compute the backtrack level, which is defined as the maximum level the SAT algorithm has to backtrack to in order to solve the conflict. This approach often leads to a *non-chronological backtracking* operation, jumping back more than just one level and making conflict-driven learning combined with non-chronological backtracking a powerful mechanism to prune large parts of the search space [24].

4 Integrating interval constraint propagation and SAT

By combining interval constraint propagation with an interval splitting scheme to obtain a branch-and-prune algorithm, as shown on the left of Table 1, a constraint solving algorithm for constraints over the reals incorporating transcendental functions can be achieved being based on interval splitting over real-valued intervals as a branching step and on ICP as a forward inference step, it does closely resemble the core algorithm of DPLL SAT solving. In fact, DPLL-SAT can be viewed as its counterpart over the Boolean intervals $\mathbb{I}_{\mathbb{B}}$, where again interval splitting is the decision step and Boolean constraint propagation in the form of unit propagation provides the forward inference mechanism, cf. right-hand side of Table 1. This similarity motivates a tighter integration of propositional SAT and arithmetic reasoning than in classical lazy theorem proving, cf. Fig. 1. This tight integration shares the common algorithmic parts, thereby providing the SAT solver with full control over and full introspection into the ICP process. This way, recent algorithmic enhancements of propositional SAT solving, like lazy clause evaluation, conflict-driven learning, and non-chronological backjumping carry over to ICP-based arithmetic constraint solving. In particular, we are able to learn forms of conflicts that are considerably more general than

	Interval constraint solving	DPLL SAT
Given:	Constraint set $C = \{c_1, \ldots, c_n\}$, initial box $B \subseteq \mathbb{R}^{\lvert\text{free}(C)\rvert}$	Clause set $C = \{c_1, \ldots, c_n\}$, initial box $B \subseteq \mathbb{B}^{\lvert\text{free}(C)\rvert}$
Goal:	Find box $B' \subseteq B$ containing satisfying valuations throughout *or* show non-existence of such B'.	
Alg.: 1.	$L := \{B\}$	$L := \{B\}$
2.	If $L \neq \emptyset$ then take some box $b :\in L$, otherwise report "unsatisfiable" and stop.	If $L \neq \emptyset$ then take most recently added box $b :\in L$, otherwise report "unsatisfiable" and stop.
3.	Use contractor C to determine subbox $b' \subseteq b$ containing all solutions in b.	Use unit propagation to determine subbox $b' \subseteq b$ containing all solutions in b.
4.	If $b' = \emptyset$ then set $L := L \setminus \{b\}$, goto 2.	If $b' = \emptyset$ then set $L := L \setminus \{b\}$, goto 2.
5.	Check whether b' strongly satisfies all constraints in C; if so then report b' as satisfying and stop.	Check whether all clauses in C are satisfied throughout b'; if so then report b' as satisfying and stop.
6.	If $b' \subset b$ then set $L := L \setminus \{b\} \cup \{b'\}$, goto 2.	If $b' \subset b$ then set $L := L \setminus \{b\} \cup \{b'\}$, goto 2.
7.	Split b into subintervals b_1 and b_2, set $L := L \setminus \{b\} \cup \{b_1, b_2\}$, goto 2.	Split b into subintervals b_1 and b_2, set $L := L \setminus \{b\} \cup \{b_1, b_2\}$, goto 2.

Table 1. Interval constraint solving (left) vs. basic DPLL SAT (right).

classical as well as generalized nogoods [15] in search procedures for constraint solving.

Interval constraint solving as a multi-valued SAT problem. The underlying idea of our algorithm is that the two central operations of ICP-based arithmetic constraint solving —interval contraction by constraint propagation and by interval splitting— correspond to asserting bounds on real-valued variables $v \sim c$ with $v \in RV$, $\sim \in \{<, \leq, \geq, >\}$ and $c \in \mathbb{Q}$. Likewise, the decision steps and unit propagations in DPLL proof search correspond to asserting literals. A unified DPLL- and ICP-based proof search on a formula ϕ from the formula language of Sect. 2 can thus be based on asserting or retracting atoms of the formula language, thereby in lockstep refining or widening an interval valuation ρ that represents the current set of candidate solutions:

1. Proof search on ϕ starts with an empty set of asserted atoms and the interval valuation ρ being the minimal element wrt. the refinement relation on interval valuations, i.e. all intervals being maximal ($\{\texttt{false}, \texttt{true}\}$ for Boolean variables and $\mathbb{R}$ or —if the variable has a bounded range— a maximal subrange thereof for real-valued variables) .
2. It continues with searching for *unit clauses* in ϕ, i.e. clauses that have only one inconclusive (on ρ) atom left and all other atoms being weakly inconsistent with the current interval valuation ρ. If such a clause is found then its unique unassigned atom is asserted. The asserted atom stems from the formula ϕ or some learned nogood and may thus be an arbitrary bound, triplet, or pair.

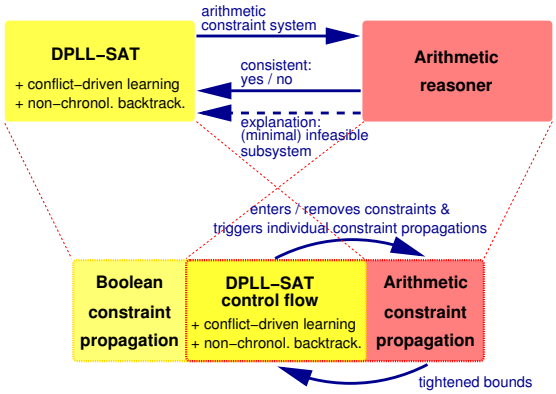


Fig. 1. Classical lazy theorem proving (top) vs. iSAT's tight integration of interval constraint solving and propositional SAT (bottom)

Step 2 is repeated until all unit clauses have been processed.

3. If there is an asserted atom a that is not weakly satisfied by the current interval valuation ρ then the contractors corresponding to a are applied to ρ. In the case of triplets and pairs, these contractors are the usual contractors for the primitive constraints of ICP, as explained in Sect. 3. For bounds $a = v \sim c$, contraction amounts to replacing $\rho(v)$ with $\rho(v) \cap \{x \in \mathbb{R} \mid x \sim c\}$, no matter whether they are literals or bounds on real-valued variables. In case of triplets and pairs, the contractions obtained are in turn asserted as bounds (this is redundant for contractions stemming from bounds, as the asserted atoms would be equal to the already asserted bound which effected the contraction).
   This step is repeated until no further contraction is obtained,[5] or until contraction detects a conflict in the sense of some interval $\rho(v)$ becoming empty. In case of a conflict, some previous splits (cf. step 4) have to be reverted, which is achieved by backtracking —thereby undoing all assertions being consequences of the split— and by asserting the complement of the previous split. Furthermore, a reason for the conflict can be recorded as a nogood, thus pruning the remaining search space (see below).
   If no conflict arose then, if new unit clauses resulted from the contraction, the algorithm continues at step 2, otherwise at 4.
4. The algorithm checks whether ρ strongly satisfies ϕ and stops with result "satisfiable" if so, as this implies real-valued satisfiability by Lemma 1. Otherwise, it applies a *splitting step*: it selects a variable $v \in BV \cup RV$ that is interpreted by a non-point interval (i.e., $|\rho(v)| > 1$) and splits its interval $\rho(v)$ by asserting a bound that contains v as a free variable and which is inconclusive on ρ.[6] Thereafter, the algorithm continues at 2. If no such vari-

[5] In practice, one stops as soon as the changes become negligible.
[6] Note that the complement of such an assertion also is a bound and is inconclusive on ρ too.

able v exists then the search space has been exhausted and the algorithm stops with result "unsatisfiable".

By its similarity to DPLL algorithms, this base algorithms lends itself to all the algorithmic enhancements and sophisticated data structures that were instrumental to the impressive recent gains in propositional SAT solver performance.

Lazy clause evaluation. In order to save costly visits to and evaluations of disjunctive clauses, we employ the lazy clause evaluation scheme of zChaff [20] to our more general class of atoms as follows: within each clause, we select two atoms which are inconclusive wrt. the current valuation ρ, called the "watched atoms" of the clause. Instead of scanning the whole clause set for unit clauses in step 2 of the base algorithm, we do only visit the clause if a free variable of one of its two watched atoms is contracted, i.e. a bound assigning a tighter bound is asserted. In this case, we evaluate the atoms truth value. If found to be inconsistent wrt. the new interval assignment, the algorithm tries to substitute the atom by a currently unwatched and not yet inconsistent atom to watch in the future. If this substitution fails due to all remaining atoms in the clause being inconsistent, the clause has become unit and the second watched atom has to be asserted.

Maintaining and compactifying an implication graph. In order to be able to tell reasons for conflicts (i.e., empty interval valuations) encountered, our solver maintains an implication graph akin to that known from propositional SAT solving [24]: all asserted atoms are recorded in a stack-like data structure (unwound upon backtracking, when the assertions are retracted). Within the stack, each assertion not originating from a split, i.e. each assertion a originating from a contraction (including unit propagations), comes equipped with pointers to its antecedents. In this case, a is a bound, i.e. a literal or a real-valued inequation $v \sim c$. The antecedent of a is an atom b containing the variable v plus a set of bounds for the other free variables of b which triggered the contraction a. As ICP often gives rise to long linear chains of contractions originating from mutually contracting via reshuffles of the same constraint, we compactify the implication stack by removing such chains, replacing them by their initial and final chain elements.

Conflict-driven learning and non-chronological backtracking. By following the antecedents of a conflicting assignment, a reason for the conflict can be obtained: reasons correspond to cuts in the antecedent graph, and such reasons can be "learned" for pruning the future search space by adding a *conflict clause* containing the disjunction of the negations of the atoms in the reason. We use the unique implication point technique [24] to derive a conflict clause which is general in that it contains few atoms and which is asserting upon backjumping to the last decision level contributing to the conflict, i.e. upon undoing all decisions and contractions younger than the chronologically youngest decision among the antecedents of the conflict, as shown in Fig. 2.

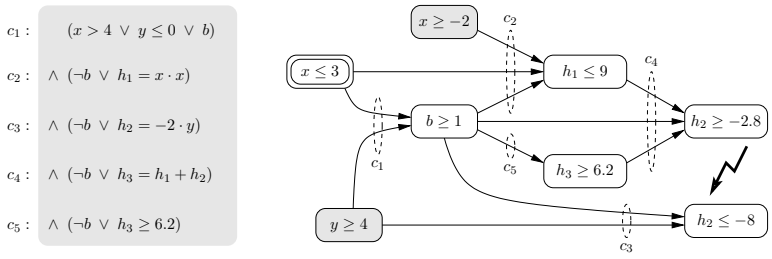


Fig. 2. Conflict analysis: Let the clause set $c_1, \ldots, c_5$ be a fragment of a formula to be solved. Assume $x \geq -2$ and $y \geq 4$ have been asserted on decision levels k_1 and k_2, resp., and another decision level is opened by asserting $x \leq 3$. The resulting implication graph, ending in a conflict on h_2, is shown on the right. Edges relate implications to their antecedents, dashed ellipses indicate the propagating clauses. Following the implication chains from the conflict yields the conflict clause $\neg(x \geq -2) \vee \neg(x \leq 3) \vee \neg(y \geq 4)$ which becomes unit after backjumping to decision level $max(k_1, k_2)$, propagating $x > 3$.

Note that, while adopting the conflict detection techniques from propositional SAT solving, our conflict clauses are still more general than those generated in propositional SAT solving: as the antecedents of a contraction may involve arbitrary atoms, so do the conflict clauses. In order to save us from being forced to generate (and handle in constraint propagation) negated triplets and pairs, we decided to allow those to occur only in the guarded form of a binary clause (*bound* $\vee$ *equation*) in our formulae (cf. syntax in sect. 2). Therefore, we can always replace a negated triplet or pair in a reason by a corresponding bound, to be found among its antecedents. Furthermore, in contrast to nogood learning, we are not confined to learning forbidden combinations of value assignments in the search space, which here would amount to learning disjunctions of interval disequations $x \notin I$ with x being a problem variable and I an interval. Instead, our algorithm may learn arbitrary combinations of atoms $x \sim c$, which provides stronger pruning of the search space: while a nogood $x \notin I$ would only prevent a future visit to any subinterval of I, a bound $x \geq c$, for example, blocks visits to any interval whose left endpoint is at least c, no matter how it is otherwise located relative to the current interval valuation.

Enforcing progress and termination. The naive base algorithm described above would apply unbounded splitting, thus risking nontermination due to the density of the order on $\mathbb{R}$. It traverses the search tree until either no further splits are possible due to the search space being fully covered by conflict clauses or until a strongly satisfying interval interpretation is found. In contrast to purely propositional SAT solving, where the split depth is bounded by the number variables in the SAT problem, this entails the risk of non-termination due to infinite sequences of splits being possible on each real-valued interval. Even worse, by pursuing depth-first search, the algorithm risks infinite descent into a branch of the search tree even if other branches may yield definite, strongly satisfying results.

We tackle this problems by either limiting the splitting width or the maximum number of splits a priori, later on refining it if necessary. I.e., we exclude a variable x from further splitting if the width of its interval $\rho(x)$ falls below a certain threshold δ, or if its predefined number of splits has been exhausted. If no further splitting is possible due to the bound having been reached for each variable, the solver derives a "pseudo-conflict clause" from that situation which —exactly as a true conflict clause would— causes the engine to backtrack and address another part of the search space, thereby abandoning the branch it has investigated previously. Besides guiding backtracking, the pseudo-conflict clause serves as a witness for the existence of an undecided branch, i.e. a branch which has not been fully explored. Note that we can re-open that branch simply by deleting the associated pseudo-conflict clause.

Achieving almost-completeness through restarts. With a given bound δ on split width or a fixed number k of maximum splits, the above procedure may terminate with inconclusive result: none of the visited boxes may be strongly satisfying, yet undecided branches in the search space —corresponding to inconclusive interval interpretations— remain. In this case, the solver simply is restarted with smaller splitting width or number of splits greater than k, respectively. Before restarting, all "pseudo conflicts" are removed from the clause database such that only the "real" conflict clauses are preserved from the previous run. These learned conflict clauses prevent the solver from re-visiting failed boxes such that the restart incurs a very low penalty: essentially, it does only visit those interval interpretations that were previously left in an inconclusive state, and it extends the proof tree precisely at these inconclusive leafs.

By iterating this scheme for incrementally smaller split-widths converging to zero, we obtain an "almost-complete" procedure determining the truth values of *robust* constraint formulae. I.e., it is able to determine the truth of a formula provided it is robust in the sense that the truth value does not change under some small variation of the constants in the formula.

5 Benchmark results

In order to demonstrate the potential of our approach, in particular the benefit of conflict-driven learning adapted to interval constraint solving, we compare the performance of our tool "iSAT" to a stripped version thereof, where learning and backjumping are disabled (but the optimized data structures, in particular watched atoms, remain functional). The benchmark results cover search for conflicts up to a given split width, yet omit the justification of conflict-free valuations thus found by checking for strong satisfaction, as the latter is not implemented yet. The benchmarks were performed on a 2.5 GHz AMD Opteron machine with 4 GByte physical memory, running Linux.

We considered bounded model checking problems, i.e. proving a property of a hybrid discrete-continuous transition system for a fixed unwinding depth k. Without learning, the interval constraint solving system failed on every moderately interesting hybrid system due to complexity problems exhausting memory

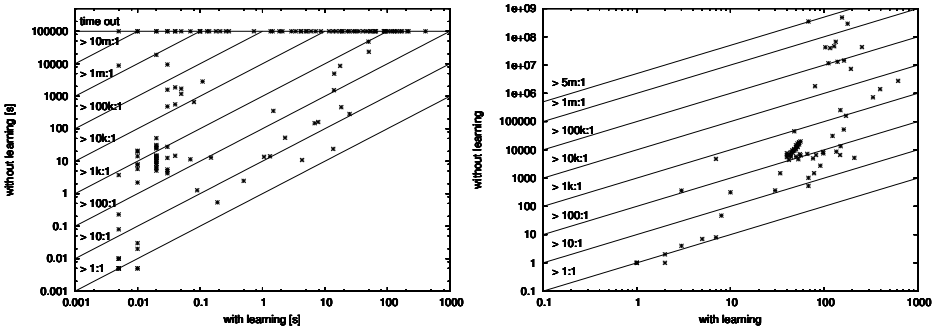


Fig. 3. Performance impact of conflict-driven learning and non-chronological backtracking: runtime in seconds (left) and number of conflicts encountered (right)

and runtime. This could be expected because the *expected* number of boxes to be visited grows exponentially in the number of variables in the constraint formula, which in turn grows linearly in both the number of problem variables in the hybrid system and in the unwinding depth k. When checking a model of an *elastic approach to train distance control* [9], the version without learning exhausts the runtime limit of 3 days already on unwinding depth 1, where formula size is 140 variables and 30 constraints. In contrast, the version with conflict-driven learning solves all instances up to depth 10 in less than 3 minutes, thereby handling instances with more than 1100 variables, a corresponding number of triplets and pairs, and 250 inequality constraints. For simpler hybrid systems, like the model of a *bouncing ball* falling in a gravity field and subject to non-ideal bouncing on the surface, the learning-free solver works due to the deterministic nature of the system. Nevertheless, it fails for unwinding depths > 11, essentially due to enormous numbers of conflicting assignments being constructed (e.g., $> 348 \cdot 10^6$ conflicts for $k = 10$), whereas learning prevents visits to most of these assignments (only 68 conflicts remain for $k = 10$ when conflict-driven learning is pursued). Consequently, the learning-enhanced solver traverses these problems in fractions of a second; it is only from depth 40 that our solver needs more than one minute to solve the bouncing ball problem (2400 variables, 500 constraints). Similar effects were observed on chaotic real-valued maps, like the *gingerbread map*. Without conflict-driven learning, the solver ran into approx. $43 \cdot 10^6$, $291 \cdot 10^6$, and $482 \cdot 10^6$ conflicts for $k = 9$ to 11, whereas only 253, 178, and 155 conflicts were encountered in the conflict-driven approach, respectively. This clearly demonstrates that conflict-driven learning is effective within interval constraint solving: it does dramatically prune the search space, as witnessed by the drastic reduction in conflict situations encountered and by the frequency of backjumps of non-trivial depth, where depths of 47 and 55 decision levels were observed on the gingerbread and bouncing ball model, respectively. Similar effects were observed on two further groups of benchmark examples: an oscillatory *logistic map* and some geometric decision problems dealing with the

intersection of n-dimensional geometric objects. On random formulae, we even obtained backjump distances of more than 70000 levels. The results of all the aforementioned benchmarks (excl. random formulae) are presented in Fig. 3.

6 Discussion

Within this paper, we have shown that a tight integration of DPLL-style SAT solving and interval constraint propagation can canonically lift to interval-based arithmetic constraint solving the crucial algorithmic enhancements of modern propositional SAT solvers, in particular lazy clause evaluation, conflict-driven learning, and non-chronological backtracking. First benchmarks on a prototype implementation demonstrate significant performance gains up to multiple orders of magnitude. Equally important, the performance gains were consistent throughout our set of benchmarks, with only one trivial instance incurring a negligible performance penalty due to the more complex algorithms.

The development of our constraint solver "iSAT" still is in an early phase, with some parts of the algorithm still under implementation. Of the algorithm described above, this does in particular apply to the justification of a solution by checking the interval valuation for strong satisfaction. Plans for future extensions do, furthermore, deal with three major topics: first, we will extend the base engine with specific optimizations for bounded model checking of hybrid systems, akin to the optimizations discussed in [9] for the case of linear hybrid automata. Second, we will use linear programming on the linear subset of the asserted atoms, i.e. on bounds and linear equations, to obtain stronger forward and backward inferences, including additional size reduction of conflicts to be learned. This would lower the overhead when reasoning over timed and (partially) linear hybrid automata, where polyhedral sets provide a more concise description of state sets than the rectangular regions provided by intervals. Finally, we are addressing native support for ordinary differential equations via ICP-based reasoning over safe numerical approximations of the solution in the interval domain, as pursued in [23].

References

1. G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowics, and R. Sebastiani. A SAT-based approach for solving formulas over boolean and linear mathematical propositions. In A. Voronkov, editor, *Proc. of the 18th International Conference on Automated Deduction*, volume 2392 of *LNCS, subseries LNAI*, pages 193–208. Springer-Verlag, 2002.
2. F. Benhamou, D. McAllester, and P. Van Hentenryck. CLP(Intervals) revisited. In *International Symposium on Logic Programming*, pages 124–138, Ithaca, NY, USA, 1994. MIT Press.
3. F. Benhamou and W. J. Older. Applying interval arithmetic to real, integer and Boolean constraints. *Journal of Logic Programming*, 32(1):1–24, 1997.
4. J. G. Cleary. Logical arithmetic. *Future Computing Systems*, 2(2):125–149, 1987.

5. E. Davis. Constraint propagation with interval labels. *Artif. Intell.*, 32(3):281–331, 1987.
6. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.
7. M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.
8. L. de Moura, S. Owre, H. Ruess, J. Rushby, and N. Shankar. The ICS decision procedures for embedded deduction. In *2nd International Joint Conference on Automated Reasoning (IJCAR)*, volume 3097 of *Lecture Notes in Computer Science*, pages 218–222, Cork, Ireland, July 2004. Springer-Verlag.
9. M. Fränzle and C. Herde. Efficient proof engines for bounded model checking of hybrid systems. *Formal Methods in System Design*, 2006.
10. H. Ganzinger. Shostak light. In A. Voronkov, editor, *Automated Deduction – CADE-18*, volume 2392 of *LNCS*, pages 332–346. Springer-Verlag, 2002.
11. E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust SAT-Solver. In *Design, Automation, and Test in Europe*, 2002.
12. T. J. Hickey. Metalevel interval arithmetic and verifiable constraint solving. *Journal of Functional and Logic Programming*, 2001(7), October 2001.
13. T. J. Hickey, Q. Ju, and M. H. van Emden. Interval arithmetic: from principles to implementation. *Journal of the ACM*, 48(5):1038–1068, 2001.
14. L. Jaulin, M. Kieffer, O. Didrit, and É. Walter. *Applied Interval Analysis, with Examples in Parameter and State Estimation, Robust Control and Robotics*. Springer, Berlin, 2001.
15. G. Katsirelos and F. Bacchus. Unrestricted nogood recording in CSP search. In F. Rossi, editor, *Principles and Practice of Constraint Programming — CP 2003*, volume 2833 of *LNCS*, pages 873–877. Springer-Verlag, 2003.
16. Y. Lebbah, M. Rueher, and C. Michel. A global filtering algorithm for handling systems of quadratic equations and inequations. In P. Van Hentenryck, editor, *Proc. of Principles and Practice of Constraint Programming (CP 2002)*, number 2470 in LNCS. Springer, 2002.
17. M. Lewis, T. Schubert, and B. Becker. Speedup Techniques Utilized in Modern SAT Solvers – An Analysis in the MIRA Environment. In *8th International Conference on Theory and Applications of Satisfiability Testing*, 2005.
18. O. Lhomme, A. Gotlieb, and M. Rueher. Dynamic optimization of interval narrowing algorithms. *Journal of Logic Programming*, 37(1–3):165–183, 1998.
19. R. E. Moore. *Interval Analysis*. Prentice Hall, NJ, 1966.
20. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. of the 38th Design Automation Conference (DAC'01)*, June 2001.
21. A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge Univ. Press, Cambridge, 1990.
22. S. Ratschan. Continuous first-order constraint satisfaction. In J. Calmet, B. Benhamou, O. Caprotti, L. Henocque, and V. Sorge, editors, *Artificial Intelligence, Automated Reasoning, and Symbolic Computation*, number 2385 in LNCS, pages 181–195. Springer, 2002.
23. O. Stauning. *Automatic Validation of Numerical Solutions*. PhD thesis, Kgs. Lyngby, Denmark, 1997.
24. L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *IEEE/ACM International Conference on Computer-Aided Design*, 2001.

[blank page]

Nogood Recording from Restarts

Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal

CRIL (Centre de Recherche en Informatique de Lens)
CNRS FRE 2499
rue de l'université, SP 16
62307 Lens cedex, France
{lecoutre,sais,tabary,vidal}@cril.univ-artois.fr

Abstract. In this paper, nogood recording is investigated for CSP within the randomization and restart framework. Our goal is to avoid the same situations to occur from one run to the next one. More precisely, nogoods are recorded when the current cutoff value is reached, i.e. before restarting the search algorithm. Such a set of nogoods is extracted from the last branch of the current search tree and managed using the structure of watched literals originally proposed for SAT. Interestingly, the number of nogoods recorded before each new run is bounded by the length of the last branch of the search tree. As a consequence, the total number of recorded nogoods is polynomial in the number of restarts. Experiments over a wide range of CSP instances demonstrate the effectiveness of this approach.

1 Introduction

Nogood recording (or learning) has been suggested as a technique to enhance CSP (Constraint Satisfaction Problem) solving in [9]. The principle is to record a nogood whenever a conflict occurs during a backtracking search. Such nogoods can then be exploited later to prevent the exploration of useless parts of the search tree. The first experimental results obtained with learning were given in the early 90's [9, 13, 27].

Contrary to CSP, the recent impressive progress in SAT (Boolean Satisfiability Problem) has been achieved using nogood recording (clause learning) under a randomization and restart policy enhanced with a very efficient lazy data structure [24]. Indeed, the interest of clause learning has arisen with the availability of large instances (encoding practical applications) which contain some structures and exhibit heavy-tailed phenomenon. Learning in SAT is a typical successful technique obtained from the cross fertilization between CSP and SAT: nogood recording [9] and conflict directed backjumping [25] have been introduced for CSP and later imported into SAT solvers [2, 21].

Recently, a generalization of nogoods, as well as an elegant learning method, have been proposed in [18, 19] for CSP. While standard nogoods correspond to variable assignments, generalized nogoods also involve value refutations. These generalized nogoods benefit from nice features. For example, they can compactly capture large sets of standard nogoods and are proved to be more powerful than standard ones to prune the search space.

As the set of nogoods that can be recorded might be of exponential size, one needs to achieve some restrictions. For example, in SAT, learned nogoods are not minimal

and are limited in number using the First Unique Implication Point (First UIP) concept. Different variants have been proposed (e.g. relevance bounded learning [2]), all of them attempt to find the best trade-off between the overhead of learning and performance improvements. Consequently, the recorded nogoods can not lead to a complete elimination of redundancy in search trees. An original alternative [29] to combine search scattering and redundancy avoidance involves performing random jumps in the search space. It is particularly relevant when an allotted time is given.

In this paper, nogood recording is investigated within the randomization and restart framework. The principle of our approach is to learn nogoods from the last branch of the search tree before a restart, discarding already explored parts of the search tree in subsequent runs. Roughly speaking, we manage nogoods by introducing a global constraint with a dedicated filtering algorithm which exploits watched literals [24]. The worst-case time complexity of this propagation algorithm is O($n^2\gamma$) where n is the number of variables and γ the number of recorded nogoods. Besides, we know that γ is at most $nd\rho$ where d is the greatest domain size and ρ is the number of restarts already performed.

This approach, so-called *nogood recording from restarts*, can be seen as the CSP adaptation of the *search signature* technique [1] introduced for SAT. Indeed, this technique involves recording the explanations (as clauses) of the search path before restarting, while discarding all clauses inferred (if any) during the last run. *Nogood recording from restarts* presents some interesting features. First, when search is stopped before finding a solution, one can run later the CSP solver with the guarantee of not exploring the same portion of the search space. Secondly, it can be used as a complementary approach of the classical learning schemes which extract and record nogoods each time a conflict occurs.

2 Technical Background

A Constraint Network (CN) P is a pair $(\mathscr{X}, \mathscr{C})$ where $\mathscr{X}$ is a set of n variables and $\mathscr{C}$ a set of e constraints. Each variable $X \in \mathscr{X}$ has an associated domain, denoted *dom*(X), which contains the set of values allowed for X. Each constraint $C \in \mathscr{C}$ involves a subset of variables of $\mathscr{X}$, denoted *vars*(C), and has an associated relation, denoted *rel*(C), which contains the set of tuples allowed for *vars*(C).

A solution to a CN is an assignment of values to all the variables such that all the constraints are satisfied. A CN is said to be satisfiable iff it admits at least one solution. The Constraint Satisfaction Problem (CSP) is the NP-complete task of determining whether a given CN is satisfiable. A CSP instance is then defined by a CN, and solving it involves either finding one (or more) solution or determining its unsatisfiability. To solve a CSP instance, one can modify the CN by using inference or search methods [10].

The backtracking algorithm (BT) is a central algorithm for solving CSP instances. It performs a depth-first search in order to instantiate variables and a backtrack mechanism when dead-ends occur. Many works have been devoted to improve its forward and backward phases by introducing look-ahead and look-back schemes [10]. Today, MAC [26] is the (look-ahead) algorithm considered as the most efficient generic approach

to solve CSP instances. It maintains a property called Arc Consistency (AC) during search. When mentioning MAC, it is important to indicate which branching scheme is employed. Indeed, it is possible to consider binary (2-way) branching or non binary (d-way) branching. These two schemes are not equivalent as it has been shown that binary branching is more powerful (to refute unsatisfiable instances) than non-binary branching [17]. With binary branching, at each step of search, a pair (X,v) is selected where X is an unassigned variable and v a value in dom(X), and two cases are considered: the assignment $X = v$ and the refutation $X \neq v$. The MAC algorithm (using binary branching) can then be seen as building a binary tree. Classically, MAC always starts by assigning variables before refuting values. Generalized Arc Consistency (GAC) (e.g. [4]) extends AC to non binary constraints, and MGAC is the search algorithm that maintains GAC.

Although sophisticated look-back algorithms such as CBJ (Conflict Directed Backjumping) [25] and DBT (Dynamic Backtracking) [14] exist, it has been shown [3, 5, 20] that MAC combined with a good variable ordering heuristic often outperforms such techniques.

3 Reduced nld-Nogoods

From now on, we will consider a search tree built by a backtracking search algorithm based on the 2-way branching scheme (e.g. MAC), positive decisions being performed first. Each branch of the search tree can then be seen as a sequence of positive and negative decisions, defined as follows:

Definition 1. *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN and (X,v) be a pair such that $X \in \mathscr{X}$ and $v \in dom(X)$. The assignment $X = v$ is called a positive decision whereas the refutation $X \neq v$ is called a negative decision. $\neg(X = v)$ denotes $X \neq v$ and $\neg(X \neq v)$ denotes $X = v$.*

Definition 2. *Let $\Sigma = \langle \delta_1, \ldots, \delta_i, \ldots, \delta_m \rangle$ be a sequence of decisions where δ_i is a negative decision. The sequence $\langle \delta_1, \ldots, \delta_i \rangle$ is called a nld-subsequence (negative last decision subsequence) of Σ. The set of positive and negative decisions of Σ are denoted by $pos(\Sigma)$ and $neg(\Sigma)$, respectively.*

Definition 3. *Let P be a CN and Δ be a set of decisions. $P|_\Delta$ is the CN obtained from P s.t., for any positive decision ($X = v$) $\in \Delta$, dom(X) is restricted to $\{v\}$, and, for any negative decision ($X \neq v$) $\in \Delta$, v is removed from dom(X).*

Definition 4. *Let P be a CN and Δ be a set of decisions. Δ is a nogood of P iff $P|_\Delta$ is unsatisfiable.*

From any branch of the search tree, a nogood can be extracted from each negative decision. This is stated by the following property:

Proposition 1. *Let P be a CN and Σ be the sequence of decisions taken along a branch of the search tree. For any nld-subsequence $\langle \delta_1, \ldots, \delta_i \rangle$ of Σ, the set $\Delta = \{\delta_1, \ldots, \neg\delta_i\}$ is a nogood of P (called nld-nogood)*[1].

[1] The notation $\{\delta_1, \ldots, \neg\delta_i\}$ corresponds to $\{\delta_j \in \Sigma \mid j < i\} \cup \{\neg\delta_i\}$ reduced to $\{\neg\delta_1\}$ when $i = 1$.

Proof. As positive decisions are taken first, when the negative decision δ_i is encountered, the subtree corresponding to the opposite decision $\neg\delta_i$ has been refuted. □

These nogoods contain both positive and negative decisions and then correspond to the definition of generalized nogoods [12, 19]. In the following, we show that nld-nogoods can be reduced in size by considering positive decisions only. Hence, we benefit from both an improvement in space complexity and a more powerful pruning capability.

By construction, CSP nogoods do not contain two opposite decisions i.e. both $x = v$ and $x \neq v$. Propositional resolution allows to deduce the clause $r = (\alpha \vee \beta)$ from the clauses $x \vee \alpha$ and $\neg x \vee \beta$. Nogoods can be represented as propositional clauses (disjunction of literals), where literals correspond to positive and negative decisions. For example, a nogood $\Delta = \{X_1 = v_1, X_2 \neq, v_2, X_3 = v_3, X_4 \neq v_4\}$ can be represented by the clause $c = (X_1 \neq v_1 \vee X_2 = v_2 \vee X_3 \neq v_3 \vee X_4 = v_4)$. Consequently, we can extend resolution to deal directly with CSP nogoods (e.g. [23]), called Constraint Resolution (C-Res for short). It can be defined as follows:

Definition 5. *Let P be a CN, and $\Delta_1 = \Gamma \cup \{x_i = v_i\}$ and $\Delta_2 = \Lambda \cup \{x_i \neq v_i\}$ be two nogoods of P. We define Constraint Resolution as C-Res$(\Delta_1, \Delta_2) = \Gamma \cup \Lambda$.*

It is immediate that C-Res(Δ_1, Δ_2) is a nogood of P.

Proposition 2. *Let P be a CN and Σ be the sequence of decisions taken along a branch of the search tree. For any nld-subsequence $\Sigma' = \langle \delta_1, \ldots, \delta_i \rangle$ of Σ, the set $\Delta = pos(\Sigma') \cup \{\neg\delta_i\}$ is a nogood of P (called reduced nld-nogood).*

Proof. We suppose that Σ contains $k \geq 1$ negative decisions, denoted by $\delta_{g_1}, \ldots, \delta_{g_k}$, in the order that they appear in Σ. The nld-subsequence of Σ with k negative decisions is $\Sigma'_1 = \langle \delta_1, \ldots, \delta_{g_1}, \ldots, \delta_{g_k} \rangle$. Its corresponding nld-nogood is $\Delta_1 = \{\delta_1, \ldots, \delta_{g_1}, \ldots, \delta_{g_{k-1}}, \ldots, \neg\delta_{g_k}\}$, $\delta_{g_{k-1}}$ being now the last negative decision. The nld-subsequence of Σ with $k-1$ negative decisions is $\Sigma'_2 = \langle \delta_1, \ldots, \delta_{g_1}, \ldots, \delta_{g_{k-1}} \rangle$. Its corresponding nld-nogood is $\Delta_2 = \{\delta_1, \ldots, \delta_{g_1}, \ldots, \neg\delta_{g_{k-1}}\}$. We now apply C-Res between Δ_1 and Δ_2 and we obtain $\Delta'_1 =$ C-Res$(\Delta_1, \Delta_2) = \{\delta_1, \ldots, \delta_{g_1}, \ldots, \delta_{g_{k-2}}, \ldots, \delta_{g_{k-1}-1}, \delta_{g_{k-1}+1}, \ldots, \neg\delta_{g_k}\}$. The last negative decision is now $\delta_{g_{k-2}}$, which will be eliminated with the nld-nogood containing $k-2$ negative decisions. All the remaining negative decisions are then eliminated by applying the same process. □

One interesting aspect is that the space required to store all nogoods corresponding to any branch of the search tree is polynomial with respect to the number of variables and the greatest domain size.

Proposition 3. *Let P be a CN and Σ be the sequence of decisions taken along a branch of the search tree. The space complexity to record all nld-nogoods of Σ is O(n^2d^2) while the space complexity to record all reduced nld-nogoods of Σ is O(n^2d).*

Proof. First, the number of negative decisions in any branch is O(nd). For each negative decision, we can extract a (reduced) nld-nogood. As the size of any (resp. reduced) nld-nogood is O(nd) (resp. O(n) since it only contains positive decisions), we obtain an overall space complexity of O(n^2d^2) (resp. O(n^2d)). □

4 Nogood Recording from Restarts

In [15], it has been shown that the runtime distribution produced by a randomized search algorithm is sometimes characterized by an extremely long tail with some infinite moment. For some instances, this heavy-tailed phenomenon can be avoided by using random restarts, i.e. by restarting search several times while randomizing the employed search heuristic. For constraint satisfaction, restarts have been shown productive. However, when learning is not exploited (as it is currently the case for most of the academic and commercial solvers), the average performance of the solver is damaged (cf. Section 6).

Nogood recording has not yet been shown to be quite convincing for CSP (one noticeable exception is [19]) and, further, it is a technique that leads, when uncontrolled, to an exponential space complexity. We propose to address this issue by combining nogood recording and restarts in the following way: reduced nld-nogoods are recorded from the last (and current) branch of the search tree between each run. Our aim is to benefit from both restarts and learning capabilities without sacrificing solver performance and space complexity.

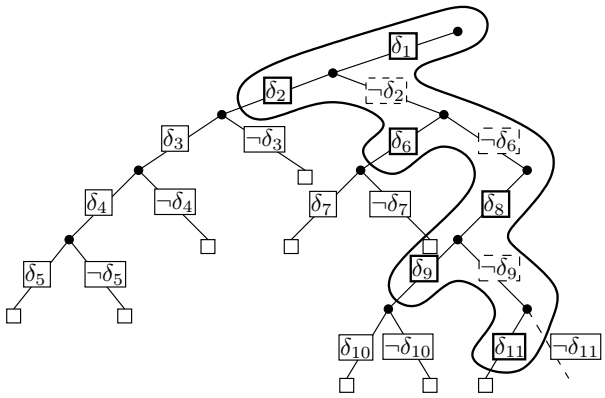


Fig. 1. Area of nld-nogoods in a partial search tree

Figure 1 depicts the partial search tree explored when the solver is about to restart. Positive decisions being taken first, a δ_i (resp. $\neg\delta_i$) corresponds to a positive (resp. negative) decision. Search has been stopped after refuting δ_{11} and taking the decision $\neg\delta_{11}$. The nld-nogoods of P are the following: $\Delta_1 = \{\delta_1, \neg\delta_2, \neg\delta_6, \delta_8, \neg\delta_9, \delta_{11}\}$, $\Delta_2 = \{\delta_1, \neg\delta_2, \neg\delta_6, \delta_8, \delta_9\}$, $\Delta_3 = \{\delta_1, \neg\delta_2, \delta_6\}$, $\Delta_4 = \{\delta_1, \delta_2\}$. The first reduced nld-nogood is obtained as follows:

$$\begin{aligned}\Delta'_1 &= \text{C-Res}(\text{C-Res}(\text{C-Res}(\Delta_1, \Delta_2), \Delta_3), \Delta_4)\\ &= \text{C-Res}(\text{C-Res}(\{\delta_1, \neg\delta_2, \neg\delta_6, \delta_8, \delta_{11}\}, \Delta_3), \Delta_4)\\ &= \text{C-Res}(\{\delta_1, \neg\delta_2, \delta_8, \delta_{11}\}, \Delta_4)\\ &= \{\delta_1, \delta_8, \delta_{11}\}\end{aligned}$$

Applying the same process to the other nld-nogoods, we obtain:
$\Delta'_2 =$ C-Res(C-Res$(\Delta_2, \Delta_3), \Delta_4) = \{\delta_1, \delta_8, \delta_9\}$.
$\Delta'_3 =$ C-Res$(\Delta_3, \Delta_4) = \{\delta_1, \delta_6\}$.
$\Delta'_4 = \Delta_4 = \{\delta_1, \delta_2\}$.

In order to avoid exploring the same parts of the search space during subsequent runs, recorded nogoods can be exploited. Indeed, it suffices to control that the decisions of the current branch do not contain all decisions of one nogood. Moreover, the negation of the last unperformed decision of any nogood can be inferred as described in the next section. For example, whenever the decision δ_1 is taken, we can infer $\neg\delta_2$ from nogood Δ'_4 and $\neg\delta_6$ from nogood Δ'_3.

Finally, we want to emphasize that *reduced nld-nogoods extracted from the last branch subsume all reduced nld-nogoods that could be extracted from any branch previously explored*.

5 Managing Nogoods

In this section, we show how to efficiently exploit reduced nld-nogoods by using the SAT technique of watched literals [24, 30, 11]. We present an efficient propagation algorithm enforcing GAC on all learned reduced nld-nogoods that can be collectively considered as a global constraint. It is important to note that, reduced nld-nogoods will be stored under the form of propositional clauses only involving negative literals.

5.1 Data structures

First, we introduce three basic types that will be useful for defining our data structures. The first one, denoted $Literal$, identifies any positive or negative decision (i.e. any variable assignment or value refutation). It then corresponds to a structure including three fields as follows:

- $variable$ is a reference to a variable
- $value$ is an integer that belongs to the initial domain of the variable
- $positive$ is a Boolean that indicates if the decision is positive ($true$) or not ($false$)

The second one, denoted $Element$, associates a nogood with two watched literals. It then corresponds to a structure including three fields as follows:

- $nogood$ is an array of $Literal$ references (whose size is at least 2)
- $watch1$ is an integer which gives the position of a first watched literal in $nogood$
- $watch2$ is an integer which gives the position of a second watched literal in $nogood$

The third one, denoted $Link$, allows to build linked lists of $Element$ references. These links are used to access nogoods recorded in the base. It corresponds to a structure including two fields as follows:

- $element$ is an $Element$ reference
- $next$ is a $Link$ reference (whose value is nil if it is not followed by another link)

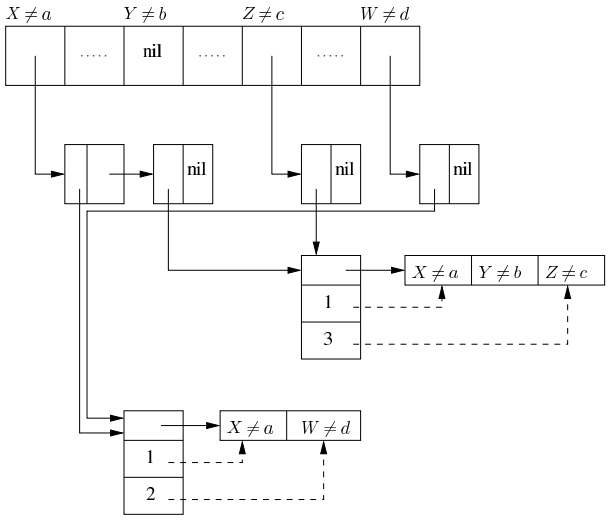


Fig. 2. Partial view of the nogood base

We can now introduce the following global structure:

– $watches$ is an array of $Link$ references, which gives, for each literal δ, the head of a list containing all nogoods Δ such that δ is watched in Δ.

We will consider that the indexing of any array t of size s ranges from 1 to s and that $size(t)$ denotes s. Also, remark that references must be considered as pointers (following the Java model), and that nil is used for empty references. Initially, $watches$ is an array such that, for each literal δ, we have $watches[\delta]$ initialized to nil. Here, to simplify the presentation and without any loss of generality, we consider $watches$ as a kind of associative array (map) which gives for each literal δ, the reference to the first nogood (via an $Element$ reference) currently involving δ as watched literal. In practice, to guarantee a constant time access to this first element, we can either use a three-dimensional array or some specific encoding of literals.

Figure 2 illustrates the data structures that we have introduced. In a partial view, one can observe two recorded nogoods. The first one contains the two watched literals $X \neq a$ and $Z \neq c$ whereas the second one contains $X \neq a$ and $W \neq d$.

5.2 Recording Nogoods

Nogoods derived from the current branch of the search tree (i.e. reduced nld-nogoods) when the current run is stopped can be recorded by calling the $storeNogoods$ function (see Algorithm 1). The parameter of this function is the sequence of literals labelling the current branch. As observed in Section 3, a reduced nld-nogood can be recorded from each negative decision occurring in this sequence. From the root to the leaf of the

Algorithm 1 storeNogoods(branch : array of Literal)

```
 1: positiveLiterals : array of size(branch) Literal
 2: nbPositiveLiterals ← 0
 3: for i ranging from 1 to size(branch) do
 4:    if branch[i].positive then
 5:       nbPositiveLiterals ← nbPositiveLiterals + 1
 6:       positiveLiterals[nbPositiveLiterals] ← branch[i]
 7:    else
 8:       if nbPositiveLiterals = 0 then
 9:          remove branch[i].value from branch[i].variable for all subsequent runs
10:       else
11:          nogood : array of nbPositiveLiterals + 1 Literal
12:          for j ranging from 1 to nbPositiveLiterals do
13:             nogood[j] ← positiveLiterals[j]
14:             nogood[j].positive ← false
15:          end for
16:          nogood[nbPositiveLiterals+1] ← branch[i]
17:          addNogood(nogood)
18:       end if
19:    end if
20: end for
```

Algorithm 2 addNogood(nogood : array of Literal)

```
1: element : Element
2: element.nogood ← nogood
3: element.watch1 ← 1
4: insertWatch(nogood[1],element)
5: element.watch2 ← size(nogood)
6: insertWatch(nogood[size(nogood)], element])
```

current branch, when a positive literal is encountered, it is recorded in an array (lines 5 and 6), and when a negative literal is encountered, we build a nogood from this literal and all recorded positive ones (lines 11 to 16). It is important to remark that, here, the nogood is considered as a clause (disjunction of literals), this is the reason why we modify the phase of the literals (see line 14). If the nogood is of size 1, it can be directly exploited by reducing the domain of the involved variable (line 9). Otherwise, it is recorded, by calling the $addNogood$ function, into the base (line 17).

To record a new nogood, the $addNogood$ function (see Algorithm 2) is called. We select as watched literals the first and last literal of the nogood. To do this, we have to call the function $insertWatch$ (see Algorithm 3). A new link is used to become the first link of the list of nogoods (via elements) involving the given literal as watched literal.

We can show that the worst-case time complexity of $storeNogoods$ is O($\lambda_p \lambda_n$) where λ_p and λ_n are the number of positive and negatives decisions on the current branch, respectively.

Algorithm 3 insertWatch(literal : Literal, element : Element)

```
1: link : Link
2: link.element ← element
3: link.next ← watches[literal]
4: watches[literal] ← link
```

5.3 Exploiting Nogoods

Inferences can be performed using reduced nld-nogoods while establishing (maintaining) Generalized Arc Consistency. We show it with a coarse-grained GAC algorithm based on a variable-oriented propagation scheme [22, 8, 6]. The Algorithm 4 can be applied to any CN (involving constraints of any arity) in order to establish GAC. At preprocessing, $propagate$ must be called with the set S of variables of the network whereas during search, S only contains the variable involved in the last positive or negative decision. At any time, the principle is to have in Q all variables whose domains have been reduced by propagation.

Algorithm 4 propagate(S : Set of variables) : Boolean

```
1: Q ← S
2: while Q ≠ ∅ do
3:    pick and delete X from Q
4:    if | dom(X) | = 1 then
5:       let a be the unique value in dom(X)
6:       if checkWatches(X ≠ a) = false then return false
7:    end if
8:    for each C | X ∈ vars(C) do
9:       for each Y ∈ Vars(C) | X ≠ Y do
10:         if revise(C,Y) then
11:            if dom(Y) = ∅ then return false
12:            else Q ← Q ∪ {Y}
13: end while
14: return true
```

Initially, Q contains all variables of the given set S (line 1). Then, iteratively, each variable X of Q is selected (line 3). If dom(X) corresponds to a singleton $\{v\}$ (lines 4 to 7), we can exploit recorded nogoods by checking the consistency of the nogood base. This is performed by the function $checkWatches$ (described below) by iterating all nogoods involving $X \neq v$ as watched literal. For each such nogood, either another literal not yet watched can be found, or an inference is performed (and the set Q is updated).

The rest of the algorithm (lines 8 to 12) corresponds to the body of a classical generic coarse-grained GAC algorithm. For each constraint C binding X, we perform the revision of all arcs (C, Y) with $Y \neq X$. A revision is performed by a call to the function $revise$, specific to the chosen coarse-grained arc consistency algorithm, and

Algorithm 5 checkWatches(literal : Literal) : Boolean

```
 1: previous ← nil
 2: current ← watches[literal]
 3: while current ≠ nil do
 4:    position ← canFindAnotherWatch(current.access)
 5:    if position ≠ -1 then
 6:       if previous = nil then watches[literal] ← watches[literal].next
 7:       else previous.next ← current.next
 8:       if literal = current.element.nogood[current.element.watch1] then
 9:         current.element.watch1 ← position
10:       else
11:         current.element.watch2 ← position
12:       let newWatchedLiteral be current.element.nogood[i]
13:       tmp ← current.next
14:       current.next ← watches[newWatchedLiteral]
15:       watches[newWatchedLiteral] ← current
16:       current ← tmp
17:    else
18:       if literal = current.element.nogood[current.element.watch1] then
19:         inferredLiteral ← current.element.nogood[current.element.watch2]
20:       else
21:         inferredLiteral ← current.element.nogood[current.element.watch1]
22:       let X be inferredLiteral.variable and v be inferredLiteral.value
23:       if v ∈ dom(X) then
24:          remove v from dom(X)
25:          if dom(X) = ∅ then return false
26:          else Q ← Q ∪ {X}
27:       end if
28:       previous ← current
29:       current ← current.next
30:    end if
31: end while
32: return true
```

entails removing values that became inconsistent with respect to C. When the revision of an arc (C, Y) involves the removal of some values in $dom(Y)$, $revise$ returns $true$ and the variable Y is added to Q. For more information about this algorithm and some of these optimizations, see [6] The algorithm loops until a fix-point is reached.

The principle of Algorithm 5 is to iterate the list of elements (nogoods) involving as watched literal the literal given in parameter. For each such element, denoted by $current$ at each turn of the main loop, we have to look for another watched literal. This is done by calling the function $canFindAnotherWatch$ (see Algorithm 6). If we can find a literal which is not currently watched (see line 2) and which can be watched then its position is returned. Otherwise, -1 is returned. When a new watched literal has been found, we have to update (i.e. remove an element) the list $watches[literal]$ (lines 6 and 7), update a watched literal position (lines 8 to 11) and update (i.e. add an element) the list $watches[newWatchedLiteral]$ (lines 13 to 15). When no other literal can be

Algorithm 6 canFindAnotherWatch(element : Element) : integer

```
1: for i ranging from 1 to size(element.nogood) do
2:     if element.watch1 = i or element.watch2 = i then continue
3:     let X be element.nogood[i].variable and v be element.nogood[i].value
4:     if v ∉ dom(X) or | dom(X) |> 1 then return i
5: end for
6: return −1
```

watched, we can then infer that the second watched literal must be verified. Remember that we only record reduced nld-nogoods. Hence, the inference is necessarily of the form $X \neq a$. Taking into account this inference when a still belongs to dom(X), we can remove a from dom(X), which can yield an inconsistency or an update of the set Q.

The worst-case time complexity of $checkWatches$ is O($n\gamma$) where γ is the number of reduced nld-nogoods stored in the base and n is the number of variables[2]. Indeed, in the worst case, each nogood is watched by the literal given in parameter, and the time complexity of dealing with a reduced nld-nogood in order to find another watched literal or make an inference is O(n). Then, the worst-case time complexity of $propagate$ is O($er^2d^r + n^2\gamma$) where r is the greatest constraint arity. More precisely, the cost of establishing GAC (using a generic approach) is O(er^2d^r) when an algorithm such as GAC2001 [4] is used and the cost of exploiting nogoods to enforce GAC is O($n^2\gamma$). Indeed, $checkWatches$ is O($n\gamma$) and it can be called only once per variable.

The space complexity of the structures introduced to manage reduced nld-nogoods in a backtracking search algorithm is O($n(d + \gamma)$). Indeed, we need to store γ nogoods of size at most n and we need to store watched literals which is O(nd).

6 Experiments

In order to show the practical interest of the approach described in this paper, we have conducted an extensive experimentation (on a PC Pentium IV 2.4GHz 512Mo under Linux). We have used the state-of-the-art algorithm MAC [26] and studied the impact of exploiting restarts (denoted by MAC+RST) and nogood recording from restarts (denoted by MAC+RST+NG). Concerning the restart policy, the initial number of allowed backtracks for the first run has been set to 10 and the increasing factor to 1.5 (i.e., at each new run, the number of allowed backtracks increases by a 1.5 factor). We used three different variable ordering heuristics: the classical $brelaz$ [7] and $dom/ddeg$ [3] as well as the adaptive $dom/wdeg$ that has been recently shown to be the most efficient generic heuristic [5, 20, 16, 28]. Importantly, when restarts are performed, randomization is introduced in $brelaz$ and $dom/ddeg$ to break ties. For $dom/wdeg$, the weight of constraints are preserved from each run to the next one, which makes randomization useless (weights are sufficiently discriminant).

In our first experimentation, we have tested the three algorithms on the full set of 1064 instances used as benchmarks for the first competition of CSP solvers [28]. The

[2] In practice, the size of reduced nld-nogoods can be far smaller than n (cf. Section 6).

time limit to solve an instance was fixed to 30 minutes. Table 1 provides an overview of the results in terms of the number of instances unsolved within the time limit ($\#timeouts$) and the average cpu time in seconds ($avg\ time$) computed from instances solved by all three methods. Figures 3 and 4 represent scatter plots displaying pairwise comparisons for $dom/ddeg$ and $dom/wdeg$. Finally, Table 2 focuses on the most difficult real-world instances of the Radio Link Frequency Assignment Problem (RLFAP). Performance is measured in terms of the cpu time in seconds (no timeout) and the number of visited nodes. An analysis of all these results reveals three main points.

Restarts (without learning) yields mitigated results. First, we observe an increased average cpu time for all heuristics and fewer solved instances for classical ones. However, a close look at the different series reveals that MAC+RST combined with $brelaz$ (resp. $dom/ddeg$) solved 27 (resp. 32) less instances than MAC on the series ehi. These instances correspond to random instances embedding a small unsatisfiable kernel. As classical heuristics do not guide search towards this kernel, restarting search tends to be nothing but an expense. Without these series, MAC+RST would have solved more instances than MAC (but, still, with worse performance). Also, remark that $dom/wdeg$ renders MAC+RST more robust than MAC (even on the ehi series).

Nogood recording from restarts improves MAC performance. Indeed, both the number of unsolved instances and the average cpu time are reduced. This is due to the fact that the solver never explores several times the same portion of the search space while benefiting from restarts.

Nogood recording from restarts applied to real-world instances pays off. When focusing to the hardest instances [28] built from the real-world RLFAP instance $scen$-11, we can observe in Table 2 that using a restart policy allows to be more efficient by almost one order of magnitude. When we further exploit nogood recording, the gain is about 10%.

		MAC		
			$+ RST$	$+ RST + NG$
$dom/ddeg$	$\#timeouts$	365	378	337
	$avg\ time$	125.0	159.0	109.1
$brelaz$	$\#timeouts$	277	298	261
	$avg\ time$	85.1	121.7	78.2
$dom/wdeg$	$\#timeouts$	140	123	121
	$avg\ time$	47.8	56.0	43.6

Table 1. Number of unsolved instances and average cpu time on the 2005 CSP competition benchmarks, given 30 minutes CPU.

Finally, we noticed that the number and the size of the reduced nld-nogoods recorded during search were always very limited. As an illustration, let us consider the hardest RLFAP instance $scen11 - f1$ which involves 680 variables and a greatest domain size of 43 values. MAC+RST+NG solved this instance in 36 runs while only 712 nogoods of average size 8.5 and maximum size 33 were recorded.

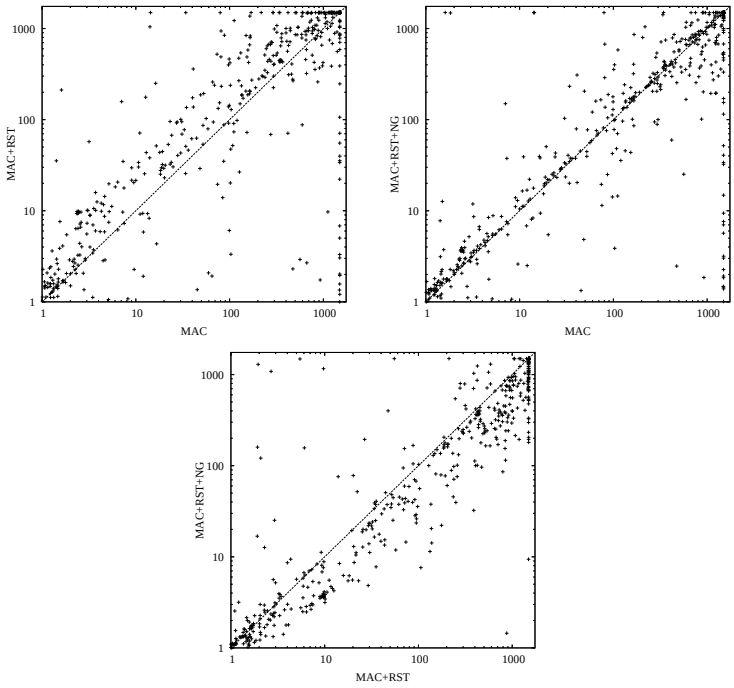


Fig. 3. Pairwise comparison (cpu time) on the 2005 CSP competition benchmarks using the dom/ddeg heuristic

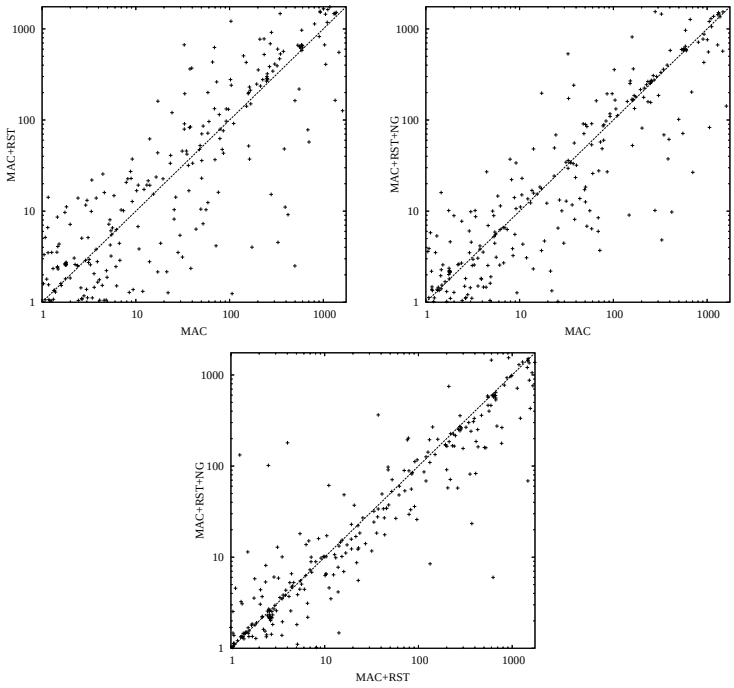


Fig. 4. Pairwise comparison (cpu time) on the 2005 CSP competition benchmarks using the dom/wdeg heuristic

		MAC		
			$+RST$	$+RST+NG$
scen11-f12	cpu	0.85	0.84	0.84
	$nodes$	695	477	445
scen11-f10	cpu	0.95	0.82	1.03
	$nodes$	862	452	636
scen11-f8	cpu	14.6	1.8	1.9
	$nodes$	14068	1359	1401
scen11-f7	cpu	185	9.4	8.4
	$nodes$	$207K$	9530	8096
scen11-f6	cpu	260	21.8	16.9
	$nodes$	$302K$	22002	16423
scen11-f5	cpu	1067	105	82.3
	$nodes$	$1327K$	$117K$	90491
scen11-f4	cpu	2494	367	339
	$nodes$	$2826K$	$419K$	$415K$
scen11-f3	cpu	9498	1207	1035
	$nodes$	$12M$	$1517K$	$1286K$
scen11-f2	cpu	$29K$	3964	3378
	$nodes$	$37M$	$5011K$	$4087K$
scen11-f1	cpu	$69K$	9212	8475
	$nodes$	$93M$	$12M$	$10M$

Table 2. Performance on hard RLFAP Instances using the $dom/wdeg$ heuristic (no timeout)

7 Conclusion

In this paper, we have studied the interest of recording nogoods in conjunction with a restart strategy. The benefit of restarting search is that the heavy-tailed phenomenon observed on some instances can be avoided. The drawback is that we can explore several times the same parts of the search tree. We have shown that it is quite easy to eliminate this drawback by recording a set of nogoods at the end of each run (similarly to the *search signature* technique proposed [1] for SAT). For efficiency reasons, nogoods are recorded in a base (and so do not correspond to new constraints) and propagation is performed using the 2-literal watching technique introduced for SAT. One can consider the base of nogoods as a unique global constraint with an efficient associated propagation algorithm.

Our experimental results show the effectiveness of our approach since the state-of-the-art generic algorithm MAC-dom/wdeg is improved. Our approach not only allows to solve more instances than the classical approach within a given timeout, but also is, on the average, faster on instances solved by both approaches.

References

1. L. Baptista, I. Lynce, and J. Marques-Silva. Complete search restart strategies for satisfiability. In *Proceedings of SSA'01 workshop held with IJCAI'01*, 2001.
2. R.J. Bayardo and R.C. Shrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of AAAI'97*, pages 203–208, 1997.
3. C. Bessiere and J. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Proceedings of CP'96*, pages 61–75, 1996.
4. C. Bessiere, J.C. Régin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

5. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.
6. F. Boussemart, F. Hemery, and C. Lecoutre. Revision ordering heuristics for the Constraint Satisfaction Problem. In *Proceedings of CPAI'04 workshop held with CP'04*, pages 29–43, 2004.
7. D. Brelaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.
8. A. Chmeiss and P. Jégou. Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(2):121–142, 1998.
9. R. Dechter. Enhancement schemes for constraint processing: backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.
10. R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.
11. N. Eén and N. Sorensson. An extensible sat-solver. In *Proceedings of SAT'03*, 2003.
12. F. Focacci and M. Milano. Global cut framework for removing symmetries. In *Proceedings of CP'01*, pages 77–92, 2001.
13. D. Frost and R. Dechter. Dead-end driven learning. In *Proceedings of AAAI'94*, pages 294–300, 1994.
14. M. Ginsberg. Dynamic backtracking. *Artificial Intelligence*, 1:25–46, 1993.
15. C.P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24:67–100, 2000.
16. T. Hulubei and B. O'Sullivan. Search heuristics and heavy-tailed behaviour. In *Proceedings of CP'05*, pages 328–342, 2005.
17. J. Hwang and D.G. Mitchell. 2-way vs d-way branching for CSP. In *Proceedings of CP'05*, pages 343–357, 2005.
18. G. Katsirelos and F. Bacchus. Unrestricted nogood recording in CSP search. In *Proceedings of CP'03*, pages 873–877, 2003.
19. G. Katsirelos and F. Bacchus. Generalized nogoods in CSPs. In *Proceedings of AAAI'05*, pages 390–396, 2005.
20. C. Lecoutre, F. Boussemart, and F. Hemery. Backjump-based techniques versus conflict-directed heuristics. In *Proceedings of ICTAI'04*, pages 549–557, 2004.
21. J.P. Marques-Silva and K.A. Sakallah. Conflict analysis in search algorithms for propositional satisfiability. Technical Report RT/4/96, INESC, Lisboa, Portugal, 1996.
22. J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19:229–250, 1979.
23. D.G. Mitchell. Resolution and constraint satisfaction. In *Proceedings of CP'03*, pages 555–569, 2003.
24. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of DAC'01*, pages 530–535, 2001.
25. P. Prosser. Hybrid algorithms for the constraint satisfaction problems. *Computational Intelligence*, 9(3):268–299, 1993.
26. D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP'94*, pages 10–20, 1994.
27. T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.
28. M.R.C. van Dongen, editor. *Proceedings of CPAI'05 workshop held with CP'05*, volume II, 2005.
29. H. Zhang. A random jump strategy for combinatorial search. In *Proceedings of AI&M'02*, 2002.
30. L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proceedings of CADE'02*, pages 295–313, 2002.

[blank page]

Automata for Nogood Recording in Constraint Satisfaction Problems *

Guillaume Richaud[1], Hadrien Cambazard[1], Barry O'Sullivan[2], and Narendra Jussien[1]

[1] École des Mines de Nantes – LINA CNRS FRE 2729
4 rue Alfred Kastler – BP 20722, F-44307 Nantes Cedex 3, France
{guillaume.richaud, hadrien.cambazard, narendra.jussien}@emn.fr

[2] Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
b.osullivan@cs.ucc.ie

Abstract. Nogood recording is a well known technique for reducing the thrashing encountered by tree search algorithms. One of the most significant disadvantages of nogood recording has been its prohibitive space complexity. In this paper we attempt to mitigate this by using an automaton to compactly represent a set of nogoods. We demonstrate how nogoods can be propagated using a known algorithm for achieving generalised arc consistency. Our experimental results on a number of benchmark problems demonstrate the utility of our approach.

1 Introduction

Nogood recording is a well known technique for reducing the level of thrashing experienced by tree search algorithms as they repeatedly rediscover the same inconsistencies. A nogood can be regarded as an assignment to a subset of the variables that cannot be extended to a solution [9]. Nogood learning was initially proposed as a Constraint Programming (CP) technique [9, 21], but without leading to significant performance improvements due to its worst-case exponential space complexity. However, it quickly became a successful technique in SAT [2]. SAT solvers seem to successfully manage the space and time requirements of nogoods. Inspired by success in SAT, several recent attempts have been made to reconsider nogoods in CP [13, 14].

There are two fundamental questions to be addressed in the context of discovering and exploiting nogoods. Firstly, *how should we compute nogoods?* Ideally one wishes to compute nogoods that rule out large parts of the future search space. Since nogoods have mostly been used to support intelligent backtracking and dynamic CSP [21], they always refer to the decision path and may be not very useful for filtering. A number of works, both in the SAT and CP community, provide some answers to this problem and we will review them in the next section. Secondly, *how should we process the nogoods we have learned?* In CP, nogoods have mostly be used to check whether the current search node can be extended to a solution or not. SAT solvers go a step further and use

* This work was supported by the Ulysses Ireland-France Travel Programme, funded in France by EGIDE (Grant Number 12476YG) and by Enterprise Ireland (Grant FR/2006/29).

a simple, but efficient, form of inference called unit propagation over the learned nogoods. Moreover, by keeping a nogood at each failure during search, one must address the problem of storing an exponential number of nogoods. The SAT community have designed clever and efficient data structures to store and propagate a very large number of clauses or nogoods. The *two watched literals* scheme [17] is one of the most successful schemes. It has also been applied in CP [13]. However the design of well suited data structures for recording nogoods in a CP context is still an open question. As a result, it is still not clear if nogood recording really pays off in CP.

We present a novel technique for storing nogoods in a compact way using an automaton. In Section 2 we recall how to compute nogoods and show how they can be propagated. In Section 3 we present our solution to store and to perform the propagation phase over a large set of tuples using an automaton. We present some preliminary experimental results in Section 4. Some concluding remarks are made in Section 5.

2 An Overview of Nogood Recording in CP

A constraint satisfaction problem (CSP) is defined by a triple $\langle X, D, C\rangle$ where $X = \{x_1, \ldots, x_n\}$ is a set of variables, $D = \{D_1, \ldots, D_n\}$ is the set of domains, where D_i is the domain of variable x_i and d is the maximum size of any domain. We denote by D_i^{orig} the original domain of x_i, and D_i is the current domain of x_i at a specific point of interest in the resolution process. Finally, C denotes the original set of constraints of the problem. Solving a CSP is achieved by interleaving search with propagation. Search can be regarded as the dynamic addition of constraints (decision constraints). For simplicity, we restrict ourselves to decision constraints that are of the form $x_i = a$, i.e. assignments of values to variables.

2.1 Computing Nogoods

A nogood is computed by analysing why a failure occurs during search. A nogood can be regarded as a subset of the assignments made so far that caused a failure. We introduce the basic definitions we require throughout the paper.

Definition 1 (Deduction) *A deduction $(x \neq a)$ is the determination that a value a should be removed from the domain of variable x.*

Definition 2 (Generalised Explanation) *A generalised explanation, $g\_expl(x_i \neq a)$ for the deduction $(x_i \neq a)$ is defined by two sets of constraints: $C' \subseteq C$ and Δ, a set of deductions, such that $C' \wedge \Delta \wedge (x_i = a)$ is globally inconsistent.*

The set Δ associated with a generalised explanation, $g\_expl(x_i \neq a)$, is denoted by $g\_expl_{\Delta}(x_i \neq a)$. Any deduction is itself, generally, due to others deductions. The inferences made during propagation can be traced back to the decision constraints added by the search algorithm. Therefore, we can compute explanations from generalised explanations. An empty Δ for a deduction $x_i \neq a$ represents the deduction that is either directly due to a decision performed on x_i such as $x_i = b$ or performed at the root node. Explaining a deduction only with respect to decisions made during search is the purpose of classical explanations.

Definition 3 (Explanation) *A (classical) explanation, $expl(x_i \neq a)$ for the deduction $x_i \neq a$ is defined by two sets of constraints: $C' \subseteq C$ and DC, a set of decision constraints (assignments), such that $C' \wedge DC \wedge (x_i = a)$ is globally inconsistent.*

Intelligent backtracking techniques usually store an explanation for each deduction [12]. Such explanations are computed on-the-fly by each constraint.

Example 1 (Explanations) *Let x and y be two variables such that $D_x = \{0, \ldots, 6\}$ and $D_y = \{0, 2, 3, 4, 6\}$. Values 1 and 5 were removed from the domain y so an explanation is already available for these deductions. Imagine that $expl(y \neq 1) = \{x_4 = 2\}$ and $expl(y \neq 5) = \{x_0 = 3, x_8 = 1\}$ and consider the constraint $|x - y| = 2$. The value 3 of x is removed by applying the filtering algorithm on $|x - y| = 2$. A generalised explanation is simply $g\_expl_{\Delta}(x \neq 3) = \{y \neq 1, y \neq 5\}$. An explanation is, for example, $expl(x \neq 3) = expl(y \neq 1) \cup expl(y \neq 5) = \{x_4 = 2, x_0 = 3, x_8 = 1\}$.* ▲

Explanations are designed for intelligent backtracking algorithms and, therefore, they always refer to a decision path. By explaining every value that is removed, one can explain a contradiction (an empty domain D_i) by computing the union of the explanations for each value removed from D_i^{orig}. The explanation that one obtains, $expl(D_i = \emptyset) = \bigcup_{j \in D_i^{orig}} expl(x_i \neq j)$, is often called a contradiction explanation and meets exactly the classical notion of nogood, i.e. *a set of assignments that cannot be extended to a solution*.

Generalised nogoods [14] enhance the pruning power of nogoods by keeping intermediate reasons for the removal of a value instead of always projecting them onto the current decision path and *postponing* the computation of the nogood when a failure occurs. Following [12, 14], one can define a generalised nogood as follows.

Definition 4 (Generalised Nogood) *A generalised nogood is a set of constraints C', a set of deductions Δ and a set of decision constraints DC such that $C' \wedge \Delta \wedge DC$ is globally inconsistent.*

By storing generalised explanations, one keeps in memory the logical chain of inferences made during search; in SAT this is referred to as the implication graph [4, 23]. From a contradiction due to an empty domain D_i of a variable x_i, one can compute several generalised nogoods (whereas only one nogood is available with the classical technique). The general scheme for computing a nogood[3] is given by Algorithm 1. Line 1 starts by computing the generalised nogood expressing the fact that the domain of variable x_i has been wiped-out and has raised a contradiction. Any deduction can be replaced by its generalised explanation to get a new (and maybe more informative) nogood. A generalised explanation whose Δ set is empty (line 6) is due to a decision made on that variable so we use the corresponding decision[4]. A generalised nogood is finally made of deductions as well as assignments. We can implement any SAT recording scheme by choosing the stopping criterion appropriately. For example, we can implement the Unique Implication Point [4] criterion if we wish to stop when we find a single reason that implies the conflict at current decision level.

[3] This is equivalent to the computation of a cut within the implication graph introduced in SAT. The implication graph is known in CP as a proof-tree [8].

[4] The explanation itself may be empty if the deduction is performed at the root node.

Algorithm 1 computeGeneralisedNogood(Var x_i)

1: GeneralisedContradictionExplanation $e \leftarrow \bigcup_{j \in D^{orig}} x_i \neq j$;
2: **while** stopping criterion not met **do**
3: $x_k \neq k \leftarrow$ choose a deduction from e;
4: **if** $g\_expl_{\Delta}(x_k \neq k)$ is not empty
5: $e \leftarrow e \cup g\_expl(x_k \neq k) - \{x_k \neq k\}$;
6: **else** $e \leftarrow e \cup expl(x_k \neq k) - \{x_k \neq k\}$;
7: **end while**
8: **return** e;

2.2 Propagating Nogoods

The propagation of nogoods is generally limited to the unit propagation approach used by SAT solvers. Consider as a literal, a variable/value pair (x_i, j). A positive literal will refer to $x_i = j$ whereas a negative literal refers to $x_i \neq j$. A positive (resp. negative) literal is said to be *satisfied* as soon as x_i is instantiated to j (resp. j removed from x_i), *falsified* in the opposite case and *free* otherwise. A generalised nogood (Δ, DC) can be seen as a constraint, i.e. a clause over the corresponding literals $(\bigvee_{x_k \neq j \in \Delta} x_k = j) \vee (\bigvee_{x_k = j \in DC} x_k \neq j)$, that must be satisfied in the remaining search. A nogood is free as long as two literals are free, satisfied as soon as one literal is satisfied and falsified once all literals are falsified. Moreover, the nogood is said to be *unit* when only one literal is free whereas all others are falsified. In this case, unit propagation enforces the free literal to be satisfied. The *two watched literals* scheme [4, 17] is recognised as the best way to propagate SAT clauses. We sketch this technique briefly here, since it is our baseline for nogood propagation.

The status of a nogood (free, satisfied, falsified or unit) can be determined by watching only two literals; each nogood is *watched* by two pointers on two free literals. Two lists of nogoods are, therefore, watched for each literal: the positive list, $pos\_watch(x_i, j)$ is the list of nogoods with the positive literal $x_i = j$, and the negative list, $neg\_watch(x_i, j)$ denotes the list of nogoods where $x_i \neq j$. The list $pos\_watch(x_i, j)$ is iterated once value j is removed from the domain of x_i (i.e., $x_i \neq j$) and $neg\_watch(x_i, j)$ is considered in case of an assignment (i.e. $x_i = j$). For each nogood in the list, the watched literal (now falsified) needs to be updated and several cases are considered:

1. The other watched literal is already satisfied, the pointer of the falsified literal is left unchanged;
2. Another free or satisfied literal is found and the watched list is updated accordingly;
3. Otherwise, all other literals are falsified. The nogood is unit and the other free literal is propagated. The falsified literal is left unchanged.

When using this scheme we leave the pointers to falsified literals that would remain valid upon backtracking. Indeed, the scheme ensures that as soon as a nogood is free, it is watched by two free literals. Adding a nogood dynamically at a leaf of the search tree is done by setting the pointers so that, again, the nogood will be watched correctly after backtracking. An advantage of the *watched literals* scheme is that it is well suited to applications where a large amount of memory is required for nogoods, since there is no need for complex data-structures that must be restored after backtracking.

Nogoods may still require an exponential amount of memory. It is, therefore, mandatory to forget some of the nogoods learned periodically during search. Several strategies have been introduced for forgetting nogoods such as *i-order bounded learning* [21] or *i-order relevance bounded learning* [1, 18]. Essentially these methods propose to only remember nogoods of a maximum size i or only those that are still relevant with the current decision path (that do not differ for more than i elements from the decision path). The space complexity of the previous schemes is $O(n \times d^i)$. However this tradeoff does not generally pay off [13, 15] and lots of nogoods may need to be recorded to make the learning worthwhile.

Efficient propagation schemes and optimised space management of nogoods are, therefore, the limiting factors of nogood recording techniques. To tackle the space bottleneck of nogoods, in this paper we propose to store them in a compiled form such as an automaton. Our assumption is that nogoods may share a lot of literals when they are learned in the same sub-tree. Based on the automaton representation we show that propagation can be performed efficiently. We discuss both of these issues below.

3 Encoding Nogoods using Automata

Generalised nogoods can be regarded as tuples. Specifically, since we consider finite domains, a deduction $(x_i \neq v_j)$ on domain D_i can be seen as $\bigvee_{v_k \in D_i \setminus \{v_j\}} (x_i = v_k)$. A set of tuples over n variables can be encoded in an acyclic automaton with $l = (n+1)$ layers corresponding to each variable and a final state F. A deterministic finite automaton is a tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states and $q_0 \in Q$ is a starting state. The alphabet Σ corresponds to the union of all domains of the variables and Σ^* to the set of all words. Each variable x_i is associated with the i^{th} layer of the automaton and outgoing transitions of nodes belonging to layer i are labelled with values of the domain D_i^{orig}. δ is a transition function from $Q \times \Sigma \mapsto Q$ and $\delta(q, val)$ denotes the state reached by applying the transition val in state q and the pair (q, val) denotes the corresponding edge. δ^* extends δ such that

$$\delta^*(q, w) = \begin{cases} \delta^*(\delta(q, x), y) & \text{if } w = xy \text{ with } x \in \Sigma \text{ and } y \in \Sigma^*; \\ \delta(q, w) & \text{if } w \in \Sigma. \end{cases}$$

We denote by $\gamma(q_1, q_2)$ the transition values that permit moves from state q_1 to state q_2. We will denote by $|A|$ the number of states of the automaton A and by $|A_i|$ the number of states of the i^{th} layer. An example of such an automaton is given in Figure 1.

This representation has already been used in the context of constraint satisfaction problems [22]. An automaton is a generic way of representing a set of tuples and, therefore, to define a constraint in an extensional manner. For a given set, S, of tuples over a finite sequence of variables X, we will refer to:

- A the automaton recognising the feasible tuples corresponding to S. That is to say, $\mathcal{L}(A) = \{w \in \Sigma^* / \delta^*(q_0, w) = F\} = S$.
- $\overline{A}$ the automaton recognising the infeasible tuples corresponding to S. In other words, $\mathcal{L}(\overline{A}) = \{w \in \Sigma^l / w \notin \mathcal{L}(A)\}$ where Σ^l denotes the words whose length is l.

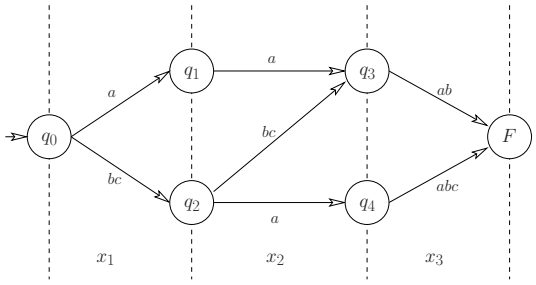


Fig. 1. An example of an automaton for three variables x_1, x_2, x_3 of domain $\{a, b, c\}$ encoding the tuples (a,a,a), (a,a,b), (b,b,a), (b,b,b), (b,c,a), (b,c,b), etc.

– $A(X)$ the automaton projected onto the current state of the domains of variables X *i.e.* that all edges (q, j) for a state q located on layer i such that $j \notin D_i$ are removed.

When the automaton is minimised (for a given ordering of the variables) it is unique, i.e. it has a canonical form. An automaton is minimal if there are no equivalent states. Two states are equivalent iff they define the same right language: $\overrightarrow{\mathcal{L}}$, i.e. they have the same set of strings that enable us to reach the final state. As we consider a layered automaton, we can efficiently minimise the automaton using bottom-to-top methods based on a recursive definition of right language of a state:

$$\overrightarrow{\mathcal{L}}(q) = \{a\,\overrightarrow{\mathcal{L}}(\delta(q,a))/a \in \Sigma \wedge \delta(q,a) \neq \bot\} \cup \begin{cases} \{\varepsilon\} \text{ if } q \in F; \\ \emptyset \quad \text{otherwise.} \end{cases}$$

Interestingly, in a minimal automaton, encoding the infeasible or feasible tuples does not matter in terms of the size of the automaton. One can easily prove that the numbers of states of A and $\overline{A}$ differs by at most l states.

Property 1 *If A and $\overline{A}$ are minimal then $abs(|A| - |\overline{A}|) < l$.*

Proof. (Sketch) One can show how to build $\overline{A}$ from A (see Figure 2). First, change the final state of A into a garbage state so that all valid tuples of A become forbidden. Second, invalid tuples of A have to be recognised and all missing transitions (those going implicitly towards the garbage state) have to be added (bold edges on Figure 2). At most $(l-1)$ states are removed (dashed edges on Figure 2). Indeed a state is removed if all its transitions lead to the old final state, and only one state per layer may have such a property (otherwise the two states would have been equivalent). Again, at most one state is added per layer (because, again, of minimality only one state may have all its outgoing transitions leading to the new final state). □

Considering a set of nogoods S (infeasible tuples), we choose to maintain the automaton $\overline{A}$ corresponding, therefore, to the set of feasible tuples. Adding a nogood within such an automaton means removing the corresponding word from the language recognised by the automaton. While this does not really matter for the automaton's size, it is easier to reason on $\overline{A}$ when propagating the automaton.

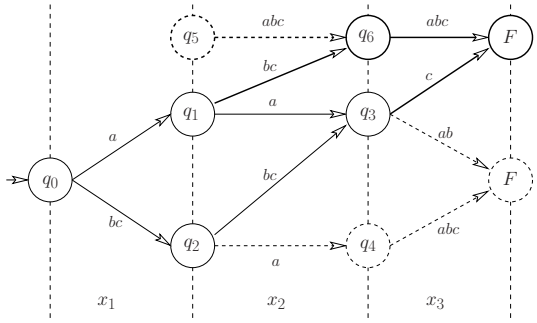


Fig. 2. Switching from A (normal and dashed edges) to $\overline{A}$ (normal and bold edges).

3.1 Incremental Minimisation of the Automaton

We briefly describe two strategies for incrementally minimising the automaton. The goal is to incrementally maintain the automaton of feasible tuples. We must be able to add nogoods (remove the corresponding word from the language recognised by the automaton) incrementally as we discover new ones [11].

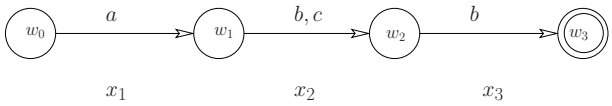


Fig. 3. Chain automaton recognising the generalised nogood $(x_1 = a) \wedge (x_2 \neq a) \wedge (x_3 = b)$ with $\Sigma = \{a, b, c\}$.

We consider a finite Σ so each deduction $(x \neq v)$ of the nogood can be replaced by $\{x \in \Sigma \backslash v\}$. We denote by w the word corresponding to the nogood to be removed from the allowed tuples of $\mathcal{L}(A)$. Removing w from $\mathcal{L}(A)$ involves building a new automaton $A \cap \overline{W}$ with W the chain-automaton recognising w (see Figure 3). W is built using the same variable order of A such $\delta^*(w_0, w)$ is the final state.

The algorithm proceeds in two steps (depicted Figure 4). Firstly, we compute $A \cap \overline{W}$: the main differences with other methods used to incrementally construct minimal acyclic automata is that we try to remove a string instead of adding it and that a generalised nogood can represent more than one string. Secondly, we incrementally minimise the new automaton by taking into account the new added states; using the fact that our automaton is layered we can minimise it efficiently. The time complexity of the removal and minimisation is $O(|W| + |\Sigma| \times |W|)$.

Adding a nogood w can add at most $|w|$ states to the automaton even if no minimisation occurs (see Step b of Figure 4). This is, however, not true for generalised nogoods. We find the incremental compilation of nogoods difficult for the following three reasons. Firstly, in the case of generalised nogoods, the automaton can be larger (in number of states) than the sum of the number of states of chain automata corresponding to the nogoods. This is due to the fact that a generalised nogood represents several tuples. A chain automaton is already a kind of compact representation. Moreover this

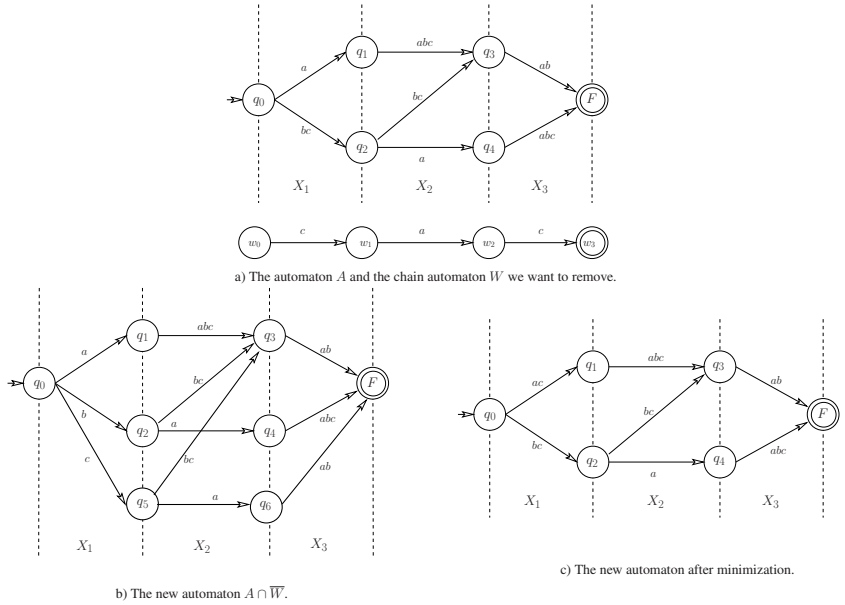


Fig. 4. The minimisation process.

behaviour is difficult to predict as it is hard to predict the size of the automaton for a given language. In the worst case, adding one generalised nogood to the automaton can add $\sum_{k=2}^{l-1} |A_k|$ new states. Secondly, the size of the automaton is related to the order of the variables. As tuples are discovered during search, the dynamic computation of the automaton would imply we re-order it dynamically. Thirdly, by adding the nogoods dynamically one by one in the order in which failures are encountered in search, the size of the automaton may increase quickly and will only decrease when enough nogoods have been learned so that they share a sufficient number of assignments.

We are currently investigating how to delay the compilation of nogoods in the hope that, as the nogoods are known, it would be possible to: select a subset of nogoods that may give a compact automaton; find a good order on the variables by applying some heuristics (based on similar ideas than for ROBDD [5]) before compilation; optimise the order in which nogoods are added to the automaton during the incremental minimisation phase to avoid large intermediate sizes [16].

3.2 The Filtering Algorithm

Pesant [19] provides a filtering algorithm for a global constraint defined by a regular language. We use this algorithm to enforce arc-consistency using our automaton A. The idea behind the algorithm is to maintain the set Q_{ij} of states acting as supports for each variable-value pair (x_i, v_j). A state q of the ith layer is considered as a support of (x_i, v_j) as long as there exists a path from q_0 to q and from $\delta(q, v_j)$ to F in A. Once Q_{ij} is empty, value j is removed from the domain of variable x_i.

In Figure 1, for example, we have $Q_{1a} = Q_{1b} = Q_{1c} = \{q_0\}$, $Q_{2a} = \{q_1, q_2\}$, $Q_{2b} = Q_{2c} = \{q_2\}$, $Q_{3a} = Q_{3b} = \{q_3, q_4\}$ and $Q_{3c} = \{q_4\}$. Incremental propagation is performed by storing in a "backtrackable" data-structure the incoming and outgoing edges of each node as well as their in and out-degree. Each time a value j is removed from the domain of variable i, the degree of the states within Q_{ij} are decremented accordingly. If some degree reaches zero, this information is propagated to all connected nodes (predecessors if the out-degree is null and successors in case of the in-degree) by decrementing their degree and maintaining the Q_{ij} lists accordingly.

In the example in Figure 1, if values a and b are removed from the domain of x_3, the out-degree of q_3 falls to zero, so its ingoing edges are considered. The states q_1 and q_2 are removed from Q_{2a}, Q_{2b} and Q_{2c} while iterating over the ingoing edges. As Q_{2b} and Q_{2c} are updated to $\emptyset$, values b and c are removed from x_2. Moreover, the out-degrees of q_1 and q_2 are decremented and the process continues as the degree of q_1 reaches zero so that value a is finally removed from x_1.

Explaining Automaton-based Filtering. As stated in Section 2.1, each filtering algorithm has to be *explained* in order to be able to generate generalised nogoods. Each time a value is removed, a generalised explanation must be associated with the deduction. It is, therefore, mandatory to explain the pruning that comes from the nogoods compiled in the automaton. Explaining the pruning of the automaton is done by, firstly, explaining why a state cannot reach F (Algorithm 2) and, secondly, explaining why a state can not be reached from q_0 (Algorithm 3). An explanation $expl(q)$ and a back-

Algorithm 2 explainOut(State q, int i)

```
1: Explanation e ← ∅;
2: if is_explained(q) is false then
3:     for all j such that δ(q, j) ≠ null do
4:         if j ∈ D_i then e ← e∪expl(δ(q, j));
5:         else e ← e ∪ expl(x_i ≠ j);
6:     end for
7:     is_explained(q) ← true;
8:     expl(q) ← e;
9: end if
```

Algorithm 3 explainIn(State q, int i)

```
1: Explanation e ← ∅;
2: if is_explained(q) is false then
3:     for all (p, j) such that δ(p, j) == q do
4:         if j ∈ D_{i−1} then e ← e ∪ expl(p);
5:         else e ← e ∪ expl(x_{i−1} ≠ j);
6:     end for
7:     is_explained(q) ← true;
8:     expl(q) ← e;
9: end if
```

trackable boolean, $is\_explained(q)$, are associated with each state q of the original automaton. $expl(q)$ records why q is invalid, i.e why it cannot be on a path from q_0 to F. $is\_explained(q)$ is true if the invalidity of q has already been explained and a valid $expl(q)$ is available in the current branch of the tree. Since many explanations exist it is mandatory to avoid overriding an existing valid explanation, because the explanation itself is not restorable upon backtracking.

A value j from a variable x_i is pruned because Q_{ij} is empty. We can explain the pruning because for each state q that was part of the original list of supports of (x_i, v_j) (denoted $Qinit_{ij}$), either q is itself invalid or $\delta(q, j)$ is invalid (Algorithm 4).

Algorithm 4 prune(int i, int j)

1: Explanation $e \leftarrow \emptyset$;
2: **for all** q in $Qinit_{ij}$ **do**
3: **if** is_explained(q) **then** $e \leftarrow e \cup expl(q)$;
4: **else** $e \leftarrow e \cup expl(\delta(q, j))$;
5: **end for**
6: remove value j from x_i due to e;

$expl(q)$ is computed for each state q in the following way. Firstly, to explain why a state q_k at layer i cannot be reached from q_0, we divide its predecessors into two sets $rpred$ and $\overline{rpred}$. The predecessors $rpred$, that can be reached from q_0, and those, $\overline{rpred}$, that are unreachable. For each predecessor p of q_k, either it belongs to $\overline{rpred}$ and we use the explanation $expl(p)$ attached to p, or it belongs to $rpred$ and the values of transitions leading to q_k from p ($\gamma(p, q_k)$) have been removed from the domain of x_{i-1}. Algorithm 3 is called each time the in-degree of q_k reaches zero and computes $expl(q_k)$:

$$expl(q_k) = expl(q_0 \nRightarrow q_k) = \bigcup_{p \in rpred} expl(x_{i-1} \neq \gamma(p, q_k)) \cup \bigcup_{p \in \overline{rpred}} expl(p).$$

Secondly, in a similar way, the state q_k cannot reach F because either its successor cannot reach F or the value leading to a state that could reach F is missing. Algorithm 2 is called each time the out-degree of q_k reaches zero and computes $expl(q_k)$:

$$expl(q_k) = expl(q_k \nRightarrow F) = \bigcup_{s \in rsucc} expl(x_i \neq \gamma(q_k, s)) \cup \bigcup_{s \in \overline{rsucc}} expl(s).$$

Lightweight Filtering Algorithms. The aim of the automaton is to compile large sets of nogoods and, therefore, to be able to mitigate the large space consumption of classical approaches. The incremental propagation algorithm is, in a sense, very greedy in memory as it needs two doubly-linked lists (incoming and outgoing arcs) and two integers (in-degree and out-degree) per state that are restorable upon backtracking. It also uses a backtrackable list Q_{ij} of states per variable-value pair. First, we give up maintaining doubly-linked lists for ingoing and outgoing edges. If the number of outgoing edges is bounded by the alphabet size (the maximum domain size), the number of ingoing edges can be equal to the number of states of the previous layer which seems unreasonable in our case. This algorithm is denoted by $Aut0$ in the following. Moreover, we investigate the following tradeoff which looses the constant time update at each variable-value removal: firstly, explained by Pesant [19], one does not need all the state-supports and only one can be kept in memory; secondly, one does not really need the exact degree of each state but only whether the degree is null or not.

One strength of watched literals precisely lies in the fact that nothing needs to be restored upon backtracking. We tried, based on this principle, to spare memory by keeping an outgoing and ingoing edge per state that are updated only when the edge is lost instead of storing the degree. A valid edge at depth k in the tree search is also valid

at depths less than k. The filtering based on $Aut0$ with the previous improvement is denoted $Aut1$. Finally, we store only one support-state for each value (x_i, v_j). When this support become invalid, we look for an other one among edges of $Q_{init_{ij}}$. $Aut1$ combined to this improvement is called $Aut2$.

4 Empirical Evaluation

We present experiments that study two aspects of the problem studied in this paper. In Section 4.1 we investigate the value of storing a large table of tuples in an automaton to perform filtering compared to generalised arc-consistency [3]. In Section 4.2 we report our experience of nogood recording with watched literals. For the reasons presented in Section 3.1, the dynamic compilation of nogoods was far too costly to be competitive. Experiments for the automaton remain to be done once the questions raised in Section 3.1 have been addressed. Crossword puzzles and RLFAP are our benchmark problems. All experiments are performed on a Pentium 4 3GHz with 1 GB of RAM under Linux with the choco constraint solver (*choco-solver.net*).

4.1 The Automaton: Storing and Filtering

Crossword puzzles problems involve filling a given grid using words from a reference dictionary such that each word is used at most once. Our interest here is that constraints have to store large tables of tuples corresponding to allowed words of the dictionary.

A variable x_i with domain $D(x_i) = \{a, b, \ldots, z\}$ is associated with each free square of the puzzle. A constraint is stated per word, i.e., per contiguous sequence of letters in the puzzle. The allowed tuples of the constraint are defined by all words of the corresponding length from a reference dictionary. A word can only be used once in the puzzle so a not-equal constraint is also added between any pairs of words with the same size. We studied two approaches to enforcing GAC on the problem:

1. The propagation scheme described above. Each dictionary of size k (all words of size k) is compiled within a minimal automaton called $auto_k$ (Q^k_{ij} denote the set Q_{ij} for $auto_k$).
2. The GAC schema introduced by [3][5]. A direct access to the supports of each variable-value pair is given within a shared data-structure among constraints. Linked-lists of words of size k that have a letter l at a given position p are stored in a three dimensional array called `supports[l][p][k]`. GAC is then achieved with a GAC2001 algorithm by storing the current support (an integer restorable upon backtracking denotes the index of the word in the `supports` data structure).

We considered the benchmark of [7] which is made of instances from size 5×5 to 23×23 and comes from the Herald Tribune Crosswords. We use the dictionary *words* that collects 45000 words. Table 1 summarises the results (time limit is set to 1 hour).

The initial propagation of the automaton is costly (initialisation of the Q_{ij} lists). GAC2001 is, therefore, faster on instances that are solved in a few nodes. However,

[5] Multidirectionality is not implemented in our GAC schema.

Table 1. Automaton and GAC filtering for crossword puzzles.

	Mac-Aut0		Mac-Aut1		Mac-Aut2		Mac-GAC	
Instances	**Time(s)**	**Node**	**Time(s)**	**Node**	**Time(s)**	**Node**	**Time(s)**	**Node**
05.01(dico:words)	**0,2**	30	0,2	30	0,3	30	0,3	30
15.01(dico:words)	1,3	75	1,3	75	1,3	75	**0,9**	75
15.02(dico:words)	**12,1**	872	13,7	872	16,6	872	25,5	872
15.07(dico:words)	**321,2**	22859	366,4	22859	554,1	22859	923,4	22859
19.02(dico:words)	**83,8**	17511	95,2	17511	214,6	17511	253,2	17511
19.05(dico:words)	$>$ 1h	**1213314**	$>$ 1h	1066428	$>$ 1h	728579	$>$ 1h	500871
21.03(dico:words)	**63,6**	13017	73,2	13017	203,5	13017	248,7	13017
21.06(dico:words)	$>$ 1h	**494848**	$>$ 1h	416718	$>$ 1h	261107	$>$ 1h	131916
21.07(dico:words)	**20,2**	1825	23,4	1825	34,5	1825	49,5	1825
23.07(dico:words)	$>$ 1h	**256456**	$>$ 1h	227168	$>$ 1h	118092	$>$ 1h	102668

on hard instances, the automaton tends to be between two and three times faster than GAC2001. This is due to the fact that words are a "structured" set of tuples (that share a lot assignments), i.e $|Q_{pl}^k| < |\texttt{supports[l][p][k]}|$. The result is that the function `seekNextSupport` which is the basis of any GAC algorithm is faster on Q_{pl}^k than on `supports[l][p][k]`.

Table 2. Average memory consumption (in Mbytes) for the four approaches.

instance	$Aut0$	$Aut1$	$Aut2$	GAC
15.07(dico:words)	31.8	21.1	16.1	19.7
19.02(dico:words)	37.2	25.2	17.6	19.1
21.03(dico:words)	51.6	33.9	22.2	23.7

In terms of memory, the automaton is more compact (1,70 Mbytes) than the `supports[l][p][k]` data structure (3,98 Mbytes) for storing all words of size 8 of the dictionary *words*. However, the data structure needed to filter the automaton consumes more memory than the GAC. Every 500 backtracks we measure the amount of memory used for the four approaches and report the average (Table 2). Among the three versions, the best compromise for space and time requirements seems to be $Aut1$. Notice that $Aut2$ is a little faster than GAC but requires less memory.

4.2 Nogood Recording

We studied the following three approaches on the Crossword puzzles problems and RLFAP (Radio Link Frequency Allocation Problems): MAC-CBJ [20] is an intelligent backtracking technique that involves in backtracking to the latest decision involved in the conflict when a failure occurs; MAC-CBJ + S is MAC-CBJ combined to standard nogood propagated by watched literals; MAC-CBJ + G is MAC-CBJ combined to generalised nogoods propagated by watched literals. The variable ordering heuristic used was min(dom/deg).

Crosswords puzzles. We again study crossword puzzles. The time limit was set up to 2 hours and $aut1$ is used for the constraints stated per word. Results are reported

Table 3. The use of watched literals implies a little overhead so MAC-CBJ remains faster on easy instances. However, we were able to find the results reported in [14], so generalised nogoods do effectively pay off on this problem. The next step will be to see the utility of the automaton for storing such nogoods.

Table 3. Nogood recording for crossword puzzles.

	MAC-CBJ		**MAC-CBJ + S**		**MAC-CBJ + G**	
	tps (s)	**node**	**tps (s)**	**node**	**tps (s)**	**node**
15.02(dico:words)	**29,7**	314	46,2	314	43,8	**303**
15.07(dico:words)	**841,6**	18182	1255,6	17918	894,3	**11172**
19.02(dico:words)	**10,8**	264	15,6	264	14,1	**219**
19.05(dico:words)	110,6	2104	64,2	1182	**47,9**	**727**
21.03(dico:words)	**11,8**	292	18,4	292	18,3	**281**
21.06(dico:words)	177,0	2707	265,8	2677	**164,8**	**1879**
21.07(dico:words)	**44,9**	1168	72,6	1157	63,5	**957**
21.04(dico:words)	$>$ 2h	87677	$>$ 2h	73342	$>$ 2h	47527
23.07(dico:words)	**45,3**	954	74,7	892	61,7	**584**
21.05(dico:words)	$>$ 2h	176226	$>$ 2h	102057	**722,8**	9407
21.10(dico:words)	$>$ 2h	74718	$>$ 2h	47346	$>$ 2h	43440
15.04(dico:words)	$>$ 2h	171029	$>$ 2h	105069	**1268,6**	**16315**
15.06(dico:words)	$>$ 2h	150552	$>$ 2h	94936	**5529,1**	**50554**
15.10(dico:words)	$>$ 2h	153260	$>$ 2h	99929	**1634,6**	**14153**
19.03(dico:words)	$>$ 2h	123770	$>$ 2h	88461	$>$ 2h	58632
19.04(dico:words)	$>$ 2h	303559	$>$ 2h	222910	**55,2**	**2294**
19.07(dico:words)	$>$ 2h	462297	$>$ 2h	289966	**131,9**	**4312**
21.01(dico:words)	$>$ 2h	86857	$>$ 2h	53418	$>$ 2h	36336
23.03(dico:words)	$>$ 2h	80810	$>$ 2h	50998	$>$ 2h	42919
23.04(dico:words)	$>$ 2h	74641	$>$ 2h	37265	$>$ 2h	23723
23.05(dico:words)	**11,7**	239	17,1	254	15,7	**226**

Radio Link Frequency Allocation Problems. Our second experiment is based on real world frequency allocation problems coming from the FullRLFAP archive [6]. The problem involves finding frequencies (f_i) for different channels of communication so that interferences are minimised. We followed the approach described in [10] to generate hard satisfaction instances. Therefore, `scenXX-wY-fZ` corresponds to the original instance `scenXX` where constraints with a weight greater than `Y` are removed, as well as the `Z` highest frequencies. Generalised nogoods are much more effective on those problems again (Table 4) however the results of [10], using a different conflict-based heuristic to ours, remain better.

Figure 5 shows the size of the automaton after the addition of each of the first 300 generalised nogoods taken from `scen03-05-11`. The basic compilation approach is dyn in which nogoods are compiled as they are discovered, using an ordering for the variables in the automaton based on the variable ordering from the nogoods themselves. We study two independent enhancements to the basic scheme. Firstly, denoted by $+ls$, we use a local search to optimise the variable ordering before compiling a set of nogoods; for example, when compiling k nogoods we try to find an ordering for the automaton that minimises its size having compiled the k nogoods. Secondly, denoted by $+s$, we compile the k nogoods in lexicographically order (instead of compiling them in the order given by the failures during search). Notice that $+ls$ affects the final size

Table 4. Nogood recording for RLFAP problems.

	MAC-Cbj		MAC-Cbj + S		MAC-Cbj + G	
	tps (s)	**node**	**tps (s)**	**node**	**tps (s)**	**node**
scen02-05-24	**0,3**	104	0,9	104	0,4	104
scen02-05-25	**3,0**	610	5,2	610	3,1	**360**
scen03-05-10	1659,2	572507	$>$ 2h	343927	**123,8**	**11575**
scen03-05-11	$>$ 2h	3506415	$>$ 2h	776095	$>$ 2h	155008
scen11-05-00	6,4	1207	8,3	1207	**3,8**	**622**
scen06-02-00	73,9	68669	164,2	61866	**5,3**	**1854**
scen07-01-04	**0,1**	202	0,2	202	0,2	**201**
scen07-01-05	**0**	26	0,1	26	0,1	26
graph08-05-10	$>$ 2h	1722485	$>$ 2h	491970	$>$ 2h	175079
graph08-05-11	$>$ 2h	1300390	$>$ 2h	494286	**46,6**	**6906**

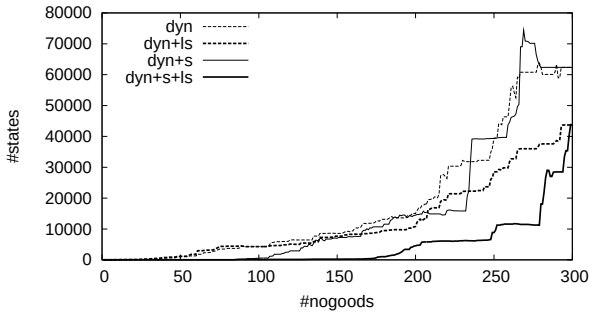


Fig. 5. Size of the automaton (#states) after the addition of each nogoods for scen03-05-11.

of the automaton, since it affects the variable ordering, whereas $+s$ affects only its intermediate size.

In Figure 5 we see two pairs of curves: one pair corresponds to the nogood-based variable ordering, while the other used local search. Within each pair of curves, while each one converges on the same size automaton having compiled all the nogoods we considered, the intermediate size is determined by the order in which the nogoods were compiled. Interestingly, a lexicographic ordering of the nogoods does not ensure that the automaton is more compact than the ordering based on how the nogoods themselves were discovered. Clearly, the variable ordering in the automaton is critical to ensure an overall compact representation, but the order in which nogoods are incrementally compiled is also important in order to avoid a large intermediate automaton.

5 Conclusion

This paper investigates a novel approach to storing and propagating nogoods. The approach uses an automaton to overcome the exponential memory requirements of nogood recording. We demonstrate the advantages and limitations of the approach. We show that the dynamic compilation of nogoods is certainly very difficult to achieve in practice but we show interesting computational results using an automaton to achieve arc-consistency over large and structured tables of tuples.

References

1. R. J. Bayardo and D. P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *AAAI-1996*, pages 298–304, 1996.
2. R. J. Bayardo and R. Schrag. Using CSP look-back techniques to solve exceptionally hard SAT instances. In *Proceedings CP 1996*, pages 46–60, 1996.
3. C. Bessière and J.-C Régin. Arc consistency for general constraint networks: Preliminary results. In *IJCAI'97*, pages 398–404, 1997.
4. L. Bordeaux, Y. Hamadi, and L. Zhang. Propositional satisfiability and constraint programming: A comparative survey. Technical Report MSR-TR-2005-124, 2005.
5. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
6. B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners. Radio link frequency assignment. *Constraints*, 4(1):79–89, 1999.
7. X. Chen and P. Beek. Conflict-directed backjumping revisited. *Journal of Artificial Intelligence Research*, 14:53–81, 2001.
8. R. Debruyne and al. Correctness of constraint retraction algorithms. In *FLAIRS'03*, pages 172–176, 2003.
9. R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.
10. C. Lecoutre F. Boussemart, F. Hemery and L. Sais. Boosting systematic search by weighting constraints. In *ECAI'04*, pages 482–486, 2004.
11. B. Watson J. Daciuk, S. Mihov and R. Watson. Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1):3–16, 2000.
12. N. Jussien and P. Boizumault. Dynamic backtracking with constraint propagation – application to static and dynamic CSPs. In *CP Workshop: Theory and Practice of Dynamic Constraint Satisfaction*, 1997.
13. G. Katsirelos and F. Bacchus. Unrestricted nogood recording in CSP search. In *Proceedings CP 2003*, pages 873–877, 2003.
14. G. Katsirelos and F. Bacchus. Generalized nogoods in CSPs. In *National Conference on Artificial Intelligence (AAAI-2005)*, pages 390–396, 2005.
15. I. Lynce and J. Marques-Silva. The effect of nogood recording in MAC-CBJ SAT algorithms. Technical Report RT/04/2002., 2002.
16. S. Mihov. Direct building of minimal automaton for given list. In *Annuaire de l'Université de Sofia St. Kl. Ohridski*, volume 91. Sofia, Bulgaria, 1998.
17. M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of DAC'01*, 2001.
18. S. Ouis, N. Jussien, and P. Boizumault. k-relevant explanations for constraint programming. In *FLAIRS'03*, pages 192–196, St. Augustine, Florida, USA, 2003.
19. G. Pesant. A regular language membership constraint for finite sequences of variables. In *CP 2004*, volume LNCS 3258, 2004.
20. P. Prosser. MAC-CBJ: maintaining arc consistency with conflict-directed backjumping. Technical Report /95/177, University of Strathclyde, 1995.
21. T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problem. *IJAIT*, 3(2):187–207, 1994.
22. N.R. Vempaty. Solving constraint satisfaction problems using finite state automata. In *AAAI*, pages 453–458, 1992.
23. L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

[blank page]